

Gerth Stølting Brodal
Stefano Leonardi (Eds.)

LNCS 3669

Algorithms – ESA 2005

13th Annual European Symposium
Palma de Mallorca, Spain, October 2005
Proceedings



 Springer

Commenced Publication in 1973

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison

Lancaster University, UK

Takeo Kanade

Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler

University of Surrey, Guildford, UK

Jon M. Kleinberg

Cornell University, Ithaca, NY, USA

Friedemann Mattern

ETH Zurich, Switzerland

John C. Mitchell

Stanford University, CA, USA

Moni Naor

Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz

University of Bern, Switzerland

C. Pandu Rangan

Indian Institute of Technology, Madras, India

Bernhard Steffen

University of Dortmund, Germany

Madhu Sudan

Massachusetts Institute of Technology, MA, USA

Demetri Terzopoulos

New York University, NY, USA

Doug Tygar

University of California, Berkeley, CA, USA

Moshe Y. Vardi

Rice University, Houston, TX, USA

Gerhard Weikum

Max-Planck Institute of Computer Science, Saarbruecken, Germany

Gerth Stølting Brodal Stefano Leonardi (Eds.)

Algorithms – ESA 2005

13th Annual European Symposium
Palma de Mallorca, Spain, October 3-6, 2005
Proceedings

Volume Editors

Gerth Stølting Brodal
University of Aarhus
BRICS, Department of Computer Science
IT-parken, Åbogade 34, 8200 Århus N, Denmark
E-mail: gerth@daimi.au.dk

Stefano Leonardi
Università di Roma "La Sapienza"
Dipartimento di Informatica e Sistemistica
Via Salaria 113, 00198 Rome, Italy
E-mail: Stefano.Leonardi@dis.uniroma1.it

Library of Congress Control Number: 2005932856

CR Subject Classification (1998): F.2, G.1-2, E.1, F.1.3, I.3.5, C.2.4, E.5

ISSN 0302-9743
ISBN-10 3-540-29118-0 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-29118-3 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

Springer is a part of Springer Science+Business Media

springeronline.com

© Springer-Verlag Berlin Heidelberg 2005
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11561071 06/3142 5 4 3 2 1 0

Preface

This volume contains the 75 contributed papers and the abstracts of the three invited lectures presented at the 13th Annual European Symposium on Algorithms (ESA 2005), held in Palma de Mallorca, Spain, October 3–6, 2005. The three distinguished invited speakers were Giuseppe F. Italiano, Cristopher Moore and Joseph (Seffi) Naor.

Since 2002, ESA has consisted of two tracks, with separate program committees, which dealt respectively with

- the design and mathematical analysis of algorithms (the “Design and Analysis” track);
- real-world applications, engineering and experimental analysis of algorithms (the “Engineering and Applications” track).

Previous ESAs in the current two track format were held in Rome, Italy (2002); Budapest, Hungary (2003); and Bergen, Norway (2004). The proceedings of these symposia were published as Springer’s LNCS volumes 2461, 2832, and 3221 respectively.

Papers were solicited in all areas of algorithmic research, including but not limited to algorithmic aspects of networks, approximation and on-line algorithms, computational biology, computational geometry, computational finance and algorithmic game theory, data structures, database and information retrieval, external memory algorithms, graph algorithms, graph drawing, machine learning, mobile computing, pattern matching and data compression, quantum computing, and randomized algorithms. The algorithms could be sequential, distributed, or parallel. Submissions were especially encouraged in the area of mathematical programming and operations research, including combinatorial optimization, integer programming, polyhedral combinatorics, and semidefinite programming.

Each extended abstract was submitted to one of the two tracks. The extended abstracts were read by at least three referees each, and evaluated on their quality, originality, and relevance to the symposium. The program committee of the Design and Analysis track met at the “Universitat Politècnica de Catalunya” on June 4–5, 2005. The Engineering and Applications track held an electronic program committee meeting. The Design and Analysis track selected 55 out of 185 submissions. The Engineering and Applications track selected 20 out of 59 submissions.

The program committees of the two tracks consisted of:

Design and Analysis track

Dimitris Achlioptas	Microsoft Research
Michael Bender	Stony Brook
Alberto Caprara	Bologna
Friedrich Eisenbrand	MPI Saarbrücken
Luisa Gargano	Salerno
Andrew Goldberg	Microsoft Research
Haim Kaplan	Tel-Aviv
Jochen Könemann	Waterloo
Stefano Leonardi (Chair)	Rome “La Sapienza”
Kirk Pruhs	Pittsburgh
Edgar Ramos	Urbana-Champaign
Adi Rosèn	The Technion, Haifa
Maria Serna	Barcelona
Christian Sohler	Paderborn
Emo Welzl	ETH Zürich
Berthold Vöcking	RWTH Aachen

Engineering and Applications Track

Jon Bentley	Avaya
Gerth Stølting Brodal (Chair)	Aarhus
Hervé Brönnimann	Brooklyn Poly
Adam L. Buchsbaum	AT&T
Riko Jacob	ETH Zürich
Richard Ladner	Washington
Sotiris Nikolettseas	Patras
Jordi Petit	Barcelona
Bettina Speckmann	Eindhoven
Subhash Suri	Santa Barbara
Sivan Toledo	Tel-Aviv

ESA 2005 was held along with the fifth Workshop on Algorithms in Bioinformatics (WABI 2005), the third workshop on Algorithmic MeThods and Models for Optimization of RailwayS (ATMOS 2005), and the third Workshop on Approximation and Online Algorithms (WAOA 2005) in the context of the combined conference ALGO 2005. The conference started right after the Solar Eclipse of October 3rd, 2005. The organizing committee of ALGO 2005 consisted of:

Carme Alvarez
 Josep Lluís Ferrer (Co-chair)
 Llorenç Huguet
 Magdalena Payeras

Jordi Petit
Maria Serna (Co-chair)
Oriol Serra
Dimitrios M. Thilikos

all from the Universitat Politècnica de Catalunya, Barcelona.

ESA 2005 was sponsored by EATCS (the European Association for Theoretical Computer Science), the Universitat Politècnica de Catalunya, the Universitat de les Illes Balears, and the Ministerio de Educación y Ciencia. The EATCS sponsorship included an award of EUR 500 for the author of the best student paper at ESA 2005. The winner of this prize was Piotr Sankowski for his paper *Shortest Paths in Matrix Multiplication Time*. The best student paper was jointly awarded by both Track A and Track B program committees.

Gerth Stølting Brodal and Stefano Leonardi would like to thank Guido Schäfer for his assistance in handling the submitted papers and these proceedings.

We hope that this volume offers the reader a selection of the best current research on algorithms.

July 2005

Gerth Stølting Brodal and Stefano Leonardi

Organization

Referees

Manuel Abellanas
Ittai Abraham
Marcel Ackermann
Micah Adler
Jochen Alber
Susanne Albers
Helmut Alt
Carme Alvarez
Christoph Ambühl
Nir Andelman
Reid Andersen
Sigrun Andradottir
Shoshana Anily
Jan Arne Telle
Tetsuo Asano
Albert Atserias
Enzo Auletta
Franz Aurenhammer
Giorgio Ausiello
Yossi Azar
Helge Bals
Evrripides Bampis
Nikhil Bansal
Amotz Bar-Noy
Lali Barriere
Paul Beame
Luca Becchetti
Richard Beigel
Robert Berke
Marcin Bienkowski
Philip Bille
Markus Bläser
Maria Blesa
Johannes Blömer
Christian Blum
Vincenzo Bonifaci
Olaf Bonorden
Stephen Boyd
Andreas Brandstadt
Joshua E. Brody
Peter Brucker
Cristoph Buchheim
Luciana Salete Buriol
Gruia Calinescu
Ioannis Caragiannis
Sunil Chandran
Shuchi Chawla
Joseph Cheriyan
Steve Chien
Jana Chlebikova
Marek Chrobak
Ken Clarkson
Jens Clausen
Andrea Clementi
Graham Cormode
Derek Corneil
Péter Csorba
Artur Czumaj
Peter Damaschke
Valentina Damerow
Gabriel de Dietrich
Roberto De Prisco
Mauro Dell'Amico
Erik Demaine
Xiaotie Deng
Olivier Devillers
Luc Devroye
Miriam Di Ianni
Zanoni Dias
Josep Diaz
Adrian Dumitrescu
Christian Duncan
Miroslaw Dynia
Jeff Edmonds
Pavlos Efraimidis
Arno Eigenwillig

Ioannis Emiris	Magns Halldorsson
David Eppstein	Dan Halperin
Leah Epstein	Mikael Hammar
Jeff Erickson	Sariel Har-Peled
Thomas Erlebach	Jason Hartline
Guy Even	Nick Harvey
Qizhi Fang	Rafael Hassin
Martin Farach-Colton	Herman Haverkort
Sándor Fekete	Laura Heinrich-Litan
Jon Feldman	Pavol Hell
Wenceslas Fernandez de la Vega	Christoph Helmberg
Paolo Ferragina	Kris Hildrum
Guillaume Fertin	Thanh Minh Hoang
Amos Fiat	Michael Hoffmann
Faith Fich	Cor Hurkens
Simon Fischer	John Iacono
Aleksei Fishkin	Nicole Immorlica
Rudolf Fleischer	Sandy Irani
Fedor Fomin	Alon Itai
Dean Foster	Kazuo Iwama
Dimitris Fotakis	Kamal Jain
Gereon Frahling	Jeannette Jansen
Gianni Franceschini	Klaus Jansen
Thomas Franke	Wojciech Jawor
Leonor Frias	Wojtek Jawor
Stefan Funke	David Johnson
Martin Furer	Sham Kakade
Bernd Gärtner	Christos Kaklamanis
Martin Gairing	Bahman Kalantari
David Gamarnik	Iyad Kanj
Naveen Garg	Howard Karloff
Leszek Gasieniec	Marek Karpinski
Cyril Gavoille	Andreas Karrenbauer
Joachim Gehweiler	Hans Kellerer
Panos Giannopoulos	Claire Kenyon
Joachim Giesen	Lutz Kettner
Seth Gilbert	Tracy Kimbrel
Ioannis Giotis	Ralf Klasing
Andreas Goerdt	Jon Kleinberg
Jens Gramm	Christian Knauer
Fabrizio Grandoni	Stephen Kobourov
Sudipto Guha	Stavros Kolliopoulos
Jens Gustedt	Spyros Kontogiannis
Michel Habib	Mirosław Korzeniowski
Torben Hagerup	Elias Koutsoupias

Dariusz Kowalski	Joseph O'Rourke
Evangelos Kranakis	Yoshio Okamoto
Dieter Kratsch	Martin Otto
Shankar Krishnan	Leonidas Palios
Alexander Kulikov	Belen Palop
Piyush Kumar	Alessandro Panconesi
Jaroslav Kutylowski	Mihai Patrascu
Eduardo Sany Laber	Christophe Paul
Gad Landau	Andrzej Pelc
Andre Lanka	David Peleg
Lap Chi Lau	Warren Perger
Sören Laue	Julian Pfeifle
Adam Letchford	Sylvain Pion
Asaf Levin	David Pisinger
Ami Litman	Greg Plaxton
Alex Lopez-Ortiz	Valentin Polishchuk
Zvi Lotker	Sheung-Hung Poon
Christos Makris	H. Prodinger
Gregorio Malajovich	Yuval Rabani
Dahlia Malkhi	Harald Räcke
Carlo Mannino	Dror Rawitz
Giovanni Manzini	Andreas Razen
Gitta Marchand	Yossi Richter
Alberto Marchetti-Spaccamela	Romeo Rizzi
Vangelis Markakis	Liam Roddity
Chip Martel	Heiko Röglin
Conrado Martinez	Udi Rotics
Domagoj Matijević	Salvador Roura
Alexander May	Stefan Rührup
Ernst Mayr	Leo Rüst
Kurt Mehlhorn	Wojciech Rytter
Joao Meidanis	Cenk Sahinalp
Jiantao Meng	Carla Savage
Uli Meyer	Gabriel Scalosub
Daniele Micciancio	Guido Schäfer
Tova Milo	Christian Scheideler
Joe Mitchell	Baruch M. Schieber
Dieter Mitsche	Christian Schindelhauer
Michael Molloy	Stefan Schirra
Shlomo Moran	Falk Schreiber
S. Muthukrishnan	Rüdiger Schultz
Umberto Nanni	Andreas S. Schulz
Giri Narasimhan	Marinella Sciortino
Alantha Newman	Raimund Seidel
Sotiris Nikolettseas	Sandeep Sen

Maria Serna
Jay Sethuraman
Jiri Sgall
Hadas Shachnai
Nira Shafir
Ori Shalev
Micha Sharir
Bruce Shepherd
Gennady Shmonin
Riccardo Silvestri
Dan Simon
Steve Skiena
Martin Skutella
Sagi Snir
Aravind Srinivasan
Miloš Stojaković
Ileana Streinu
C. R. Subramanian
Maxim Sviridenko
Chaitanya Swamy
Tibor Szabó
Annon Ta-Shama
Kunal Talwar
Hisao Tamaki
Arik Tamir
Monique Teillaud
Kavitha Telikepalli
Prasad Tetali
Dimitrios M. Thilikos
Robin Thomas
Karsten Tiemann
Sivan Toledo
Enrico Tronci
Elias Tsigaridas
Levent Tuncel
Walter Unger

Ugo Vaccaro
Jan Vahrenhold
Gabriel Valiente
Marc van Kreveld
René van Oostrum
Rob van Stee
Lieven Vandenbergh
Suresh Venkatasubramanian
Elad Verbin
Kiem-Phong Vo
Klaus Volbert
Tjark Vredeveld
Uli Wagner
Tandy Warnow
Birgitta Weber
Oren Weimann
Ron Wein
Matthias Westermann
Udi Wieder
Gordon Wilfong
David Williamson
Gerhard Woeginger
Nicola Wolpert
Fatos Xhafa
Mutsunori Yagiura
Koichi Yamazaki
Baiyu Yang
Chee Yap
Martin Zachariasen
Giacomo Zambelli
Christos Zaroliagis
Lisa Zhang
Martin Ziegler
Philipp Zumein
Uri Zwick

Table of Contents

Designing Reliable Algorithms in Unreliable Memories <i>Irene Finocchi, Fabrizio Grandoni, Giuseppe F. Italiano</i>	1
From Balanced Graph Partitioning to Balanced Metric Labeling <i>Joseph Naor</i>	9
Fearful Symmetries: Quantum Computing, Factoring, and Graph Isomorphism <i>Cristopher Moore</i>	10
Exploring an Unknown Graph Efficiently <i>Rudolf Fleischer, Gerhard Trippen</i>	11
Online Routing in Faulty Meshes with Sub-linear Comparative Time and Traffic Ratio <i>Stefan Rührup, Christian Schindelhauer</i>	23
Heuristic Improvements for Computing Maximum Multicommodity Flow and Minimum Multicut <i>Garima Batra, Naveen Garg, Garima Gupta</i>	35
Relax-and-Cut for Capacitated Network Design <i>Georg Kliewer, Larissa Timajev</i>	47
On the Price of Anarchy and Stability of Correlated Equilibria of Linear Congestion Games <i>George Christodoulou, Elias Koutsoupias</i>	59
The Complexity of Games on Highly Regular Graphs <i>Konstantinos Daskalakis, Christos H. Papadimitriou</i>	71
Computing Equilibrium Prices: Does Theory Meet Practice? <i>Bruno Codenotti, Benton McCune, Rajiv Raman, Kasturi Varadarajan</i>	83
Efficient Exact Algorithms on Planar Graphs: Exploiting Sphere Cut Branch Decompositions <i>Frederic Dorn, Eelko Penninx, Hans L. Bodlaender, Fedor V. Fomin</i>	95

An Algorithm for the SAT Problem for Formulae of Linear Length <i>Magnus Wahlström</i>	107
Linear-Time Enumeration of Isolated Cliques <i>Hiro Ito, Kazuo Iwama, Tsuyoshi Osumi</i>	119
Finding Shortest Non-separating and Non-contractible Cycles for Topologically Embedded Graphs <i>Sergio Cabello, Bojan Mohar</i>	131
Delineating Boundaries for Imprecise Regions <i>Iris Reinbacher, Marc Benkert, Marc van Kreveld, Joseph S.B. Mitchell, Alexander Wolff</i>	143
EXACUS: Efficient and Exact Algorithms for Curves and Surfaces <i>Eric Berberich, Arno Eigenwillig, Michael Hemmer, Susan Hert, Lutz Kettner, Kurt Mehlhorn, Joachim Reichel, Susanne Schmitt, Elmar Schömer, Nicola Wolpert</i>	155
Min Sum Clustering with Penalties <i>Refael Hassin, Einat Or</i>	167
Improved Approximation Algorithms for Metric Max TSP <i>Zhi-Zhong Chen, Takayuki Nagoya</i>	179
Unbalanced Graph Cuts <i>Ara Hayrapetyan, David Kempe, Martin Pál, Zoya Svitkina</i>	191
Low Degree Connectivity in Ad-Hoc Networks <i>Luděk Kučera</i>	203
5-Regular Graphs are 3-Colorable with Positive Probability <i>J. Díaz, G. Grammatikopoulos, A.C. Kaporis, L.M. Kirousis, X. Pérez, D.G. Sotiropoulos</i>	215
Optimal Integer Alphabetic Trees in Linear Time <i>T.C. Hu, Lawrence L. Larmore, J. David Morgenthaler</i>	226
Predecessor Queries in Constant Time? <i>Marek Karpinski, Yakov Nekrich</i>	238
An Algorithm for Node-Capacitated Ring Routing <i>András Frank, Zoltán Király, Balázs Kotnyek</i>	249
On Degree Constrained Shortest Paths <i>Samir Khuller, Kwangil Lee, Mark Shayman</i>	259

A New Template for Solving p -Median Problems for Trees in Sub-quadratic Time <i>Robert Benkoczi, Binay Bhattacharya</i>	271
Roll Cutting in the Curtain Industry <i>Arianna Alfieri, Steef L. van de Velde, Gerhard J. Woeginger</i>	283
Space Efficient Algorithms for the Burrows-Wheeler Backtransformation <i>Ulrich Lauther, Tamás Lukovszki</i>	293
Cache-Oblivious Comparison-Based Algorithms on Multisets <i>Arash Farzan, Paolo Ferragina, Gianni Franceschini, J. Ian Munro</i>	305
Oblivious vs. Distribution-Based Sorting: An Experimental Evaluation <i>Geeta Chaudhry, Thomas H. Cormen</i>	317
Allocating Memory in a Lock-Free Manner <i>Anders Gidenstam, Marina Papatriantafidou, Philippos Tsigas</i>	329
Generating Realistic Terrains with Higher-Order Delaunay Triangulations <i>Thierry de Kok, Marc van Kreveld, Maarten Löffler</i>	343
I/O-Efficient Construction of Constrained Delaunay Triangulations <i>Pankaj K. Agarwal, Lars Arge, Ke Yi</i>	355
Convex Hull and Voronoi Diagram of Additively Weighted Points <i>Jean-Daniel Boissonnat, Christophe Delage</i>	367
New Tools and Simpler Algorithms for Branchwidth <i>Christophe Paul, Jan Arne Telle</i>	379
Treewidth Lower Bounds with Brambles <i>Hans L. Bodlaender, Alexander Grigoriev, Arie M.C.A. Koster</i>	391
Minimal Interval Completions <i>Pinar Heggernes, Karol Suchan, Ioan Todinca, Yngve Villanger</i>	403
A 2-Approximation Algorithm for Sorting by Prefix Reversals <i>Johannes Fischer, Simon W. Ginzinger</i>	415
Approximating the 2-Interval Pattern Problem <i>Maxime Crochemore, Danny Hermelin, Gad M. Landau, Stéphane Vialette</i>	426

A Loopless Gray Code for Minimal Signed-Binary Representations <i>Gurmeet Singh Manku, Joe Sawada</i>	438
Efficient Approximation Schemes for Geometric Problems? <i>Dániel Marx</i>	448
Geometric Clustering to Minimize the Sum of Cluster Sizes <i>Vittorio Bilò, Ioannis Caragiannis, Christos Kaklamani, Panagiotis Kanellopoulos</i>	460
Approximation Schemes for Minimum 2-Connected Spanning Subgraphs in Weighted Planar Graphs <i>André Berger, Artur Czumaj, Michelangelo Grigni, Hairong Zhao</i>	472
Packet Routing and Information Gathering in Lines, Rings and Trees <i>Yossi Azar, Rafi Zachut</i>	484
Jitter Regulation for Multiple Streams <i>David Hay, Gabriel Scalosub</i>	496
Efficient c -Oriented Range Searching with DOP-Trees <i>Mark de Berg, Herman Haverkort, Micha Streppel</i>	508
Matching Point Sets with Respect to the Earth Mover's Distance <i>Sergio Cabello, Panos Giannopoulos, Christian Knauer, Günter Rote</i>	520
Small Stretch Spanners on Dynamic Graphs <i>Giorgio Ausiello, Paolo G. Franciosa, Giuseppe F. Italiano</i>	532
An Experimental Study of Algorithms for Fully Dynamic Transitive Closure <i>Ioannis Krommidas, Christos Zaroliagis</i>	544
Experimental Study of Geometric t -Spanners <i>Mohammad Farshi, Joachim Gudmundsson</i>	556
Highway Hierarchies Hasten Exact Shortest Path Queries <i>Peter Sanders, Dominik Schultes</i>	568
Preemptive Scheduling of Independent Jobs on Identical Parallel Machines Subject to Migration Delays <i>Aleksei V. Fishkin, Klaus Jansen, Sergey V. Sevastyanov, René Sitters</i>	580

Fairness-Free Periodic Scheduling with Vacations <i>Jiří Sgall, Hadas Shachnai, Tami Tamir</i>	592
Online Bin Packing with Cardinality Constraints <i>Leah Epstein</i>	604
Fast Monotone 3-Approximation Algorithm for Scheduling Related Machines <i>Annamária Kovács</i>	616
Engineering Planar Separator Algorithms <i>Martin Holzer, Grigorios Prasinos, Frank Schulz, Dorothea Wagner, Christos Zaroliagis</i>	628
STXXL : Standard Template Library for XXL Data Sets <i>Roman Dementiev, Lutz Kettner, Peter Sanders</i>	640
Negative Cycle Detection Problem <i>Chi-Him Wong, Yiu-Cheong Tam</i>	652
An Optimal Algorithm for Querying Priced Information: Monotone Boolean Functions and Game Trees <i>Ferdinando Cicalese, Eduardo Sany Laber</i>	664
Online View Maintenance Under a Response-Time Constraint <i>Kamesh Munagala, Jun Yang, Hai Yu</i>	677
Online Primal-Dual Algorithms for Covering and Packing Problems <i>Niv Buchbinder, Joseph Naor</i>	689
Efficient Algorithms for Shared Backup Allocation in Networks with Partial Information <i>Yigal Bejerano, Joseph Naor, Alexander Sprintson</i>	702
Using Fractional Primal-Dual to Schedule Split Intervals with Demands <i>Reuven Bar-Yehuda, Dror Rawitz</i>	714
An Approximation Algorithm for the Minimum Latency Set Cover Problem <i>Refael Hassin, Asaf Levin</i>	726
Workload-Optimal Histograms on Streams <i>S. Muthukrishnan, M. Strauss, X. Zheng</i>	734
Finding Frequent Patterns in a String in Sublinear Time <i>Petra Berenbrink, Funda Ergun, Tom Friedetzky</i>	746

Online Occlusion Culling <i>Gereon Frahling, Jens Krokowski</i>	758
Shortest Paths in Matrix Multiplication Time <i>Piotr Sankowski</i>	770
Computing Common Intervals of K Permutations, with Applications to Modular Decomposition of Graphs <i>Anne Bergeron, Cedric Chauve, Fabien de Montgolfier, Mathieu Raffinot</i>	779
Greedy Routing in Tree-Decomposed Graphs <i>Pierre Fraigniaud</i>	791
Making Chord Robust to Byzantine Attacks <i>Amos Fiat, Jared Saia, Maxwell Young</i>	803
Bucket Game with Applications to Set Multicover and Dynamic Page Migration <i>Marcin Bienkowski, Jarosław Byrka</i>	815
Bootstrapping a Hop-Optimal Network in the Weak Sensor Model <i>Martín Farach-Colton, Rohan J. Fernandes, Miguel A. Mosteiro</i>	827
Approximating Integer Quadratic Programs and MAXCUT in Subdense Graphs <i>Andreas Björklund</i>	839
A Cutting Planes Algorithm Based Upon a Semidefinite Relaxation for the Quadratic Assignment Problem <i>Alain Faye, Frédéric Roupin</i>	850
Approximation Complexity of min-max (Regret) Versions of Shortest Path, Spanning Tree, and Knapsack <i>Hassene Aissi, Cristina Bazgan, Daniel Vanderpooten</i>	862
Robust Approximate Zeros <i>Vikram Sharma, Zilin Du, Chee K. Yap</i>	874
Optimizing a 2D Function Satisfying Unimodality Properties <i>Erik D. Demaine, Stefan Langerman</i>	887
Author Index	899

Designing Reliable Algorithms in Unreliable Memories^{*}

Irene Finocchi¹, Fabrizio Grandoni¹, and Giuseppe F. Italiano²

¹ Dipartimento di Informatica, Università di Roma “La Sapienza”,
Via Salaria 113, 00198 Roma, Italy
{finocchi, grandoni}@di.uniroma1.it

² Dipartimento di Informatica, Sistemi e Produzione,
Università di Roma “Tor Vergata”, Via del Politecnico 1, 00133 Roma, Italy
italiano@disp.uniroma2.it

Abstract. Some of today’s applications run on computer platforms with large and inexpensive memories, which are also error-prone. Unfortunately, the appearance of even very few memory faults may jeopardize the correctness of the computational results. An algorithm is resilient to memory faults if, despite the corruption of some memory values before or during its execution, it is nevertheless able to get a correct output at least on the set of uncorrupted values. In this paper we will survey some recent work on reliable computation in the presence of memory faults.

1 Introduction

The inexpensive memories used in today’s computer platforms are not fully secure, and sometimes the content of a memory unit may be temporarily or permanently lost or damaged. This may depend on manufacturing defects, power failures, or environmental conditions such as cosmic radiation and alpha particles [15,23,30,32]. Unfortunately, even very few memory faults may jeopardize the correctness of the underlying algorithms, and thus the quest for reliable computation in unreliable memories arises in an increasing number of different settings.

Many large-scale applications require the processing of huge data sets that can easily reach the order of Terabytes: for instance, NASA’s Earth Observing System generates Petabytes of data per year, while Google currently reports to be indexing and searching over 8 billion Web pages. In all such applications processing massive data sets, there is an increasing demand for large, fast, and inexpensive memories, at any level of the memory hierarchy: this trend is witnessed, e.g., by the large popularity achieved in recent years by commercial Redundant Arrays of Inexpensive Disks (RAID) systems [7,18], which offer enhanced I/O bandwidths, large capacities, and low cost. As the memory size becomes larger, however, the mean time between failures decreases considerably: assuming standard soft error rates for the internal memories currently on the market [30],

* Work supported by the Italian MIUR Project ALGO-NEXT “Algorithms for the Next Generation Internet and Web: Methodologies, Design and Experiments”.

a system with Terabytes of memory (e.g., a cluster of computers with a few Gigabytes per node) would experience one bit error every few minutes. Error checking and correction circuitry added at the board level could contrast this phenomenon, but would also impose non-negligible costs in performance and money: hence, it is not a feasible solution when speed and cost are both at prime concern.

A different application domain for reliable computation is fault-based crypt-analysis. Some recent optical and electromagnetic perturbation attacks [4,29] work by manipulating the non-volatile memories of cryptographic devices, so as to induce very timing-precise controlled faults on given individual bits: this forces the devices to output wrong ciphertexts that may allow the attacker to determine the secret keys used during the encryption. Differently from the almost random errors affecting the behavior of large size memories, in this context the errors are introduced by a malicious adversary that can assume some knowledge of the algorithm's behavior.

In order to protect the computation against destructive memory faults, data replication would be a natural approach. However, it can be very inefficient in highly dynamic contexts or when the objects to be managed are large and complex: copying such objects can indeed be very costly, and in some cases we might not even know how to do this (for instance, when the data is accessed through pointers, which are moved around in memory instead of the data itself, and the algorithm relies on user-defined access functions). In these cases, we can assume neither the existence of *ad hoc* functions for data replication nor the definition of suitable encoding mechanisms to maintain a reasonable storage cost. Instead, it makes sense to assume that it is the algorithms themselves in charge of dealing with memory faults.

Informally, we have a *memory fault* when the correct value that should be stored in a memory location gets altered because of a failure. We say that an algorithm is *resilient to memory faults* if, despite the corruption of some memory values before or during its execution, the algorithm is nevertheless able to get a correct output (at least) on the set of uncorrupted values. In this paper we survey some recent work on the design and analysis of resilient algorithms by focusing on the two basic problems of sorting and searching, which are fundamental in many large scale applications. For instance, the huge data sets processed by Web search engines are typically stored in low cost memories by means of inverted indices which have to be maintained sorted for fast document access: for such large data structures, even a small failure probability can result in few bit flips in the index, that may become responsible of erroneous answers to keyword searches [16]. In Section 2 we will discuss different models and approaches proposed in the literature to cope with unreliable information. In Section 3 we will highlight a faulty memory model and overview known results and techniques.

2 Models and Related Work

The problem of computing with unreliable information or in the presence of faulty components dates back to the 50's [33]. Due to the heterogeneous nature

of faults (e.g., permanent or transient) and to the large variety of components that may be faulty in computer platforms (e.g., processors, memories, network nodes or links), many different models have been proposed in the literature. In this section we will briefly survey only those models that appear to be most relevant to the problem of computing with unreliable memories.

The Liar Model. Two-person games in the presence of unreliable information have been the subject of extensive research since Rényi [28] and Ulam [31] posed the following “twenty questions problem”:

Two players, Paul and Carole, are playing the game. Carole thinks of a number between one and one million, which Paul has to guess by asking up to twenty questions with binary answers. How many questions does Paul need to get the right answer if Carole is allowed to lie once or twice?

Many subsequent works have addressed the problem of searching by asking questions answered by a possibly lying adversary [1,6,11,12,19,24,25,26]. These works consider questions of different formats (e.g., comparison questions or general yes-no questions such as “Does the number belong to a given subset of the search space?”) and different degrees of interactivity between the players (in the adaptive framework, Carole must answer each question before Paul asks the next one, while in the non-adaptive framework all questions must be issued in one batch). We remark that the problem of finding optimal searching strategies has strong relationships with the theory of error correcting codes. Furthermore, different kinds of limitations can be posed on the way Carole is allowed to lie: e.g., fixed number of errors, probabilistic error model, or linearly bounded model in which the number of lies can grow proportionally with the number of questions. Even in the very difficult linearly bounded model, searching is now well understood and can be solved to optimality: Borgstrom and Kosaraju [6], improving over [1,11,25], designed an $O(\log n)$ searching algorithm. We refer to the excellent survey [26] for an extensive bibliography on this topic.

Problems such as sorting and selection have instead drastically different bounds. Lakshmanan *et al.* [20] proved that $\Omega(n \log n + k \cdot n)$ comparisons are necessary for sorting when at most k lies are allowed. The best known $O(n \log n)$ algorithm tolerates only $O(\log n / \log \log n)$ lies, as shown by Ravikumar in [27]. In the linearly bounded model, an exponential number of questions is necessary even to test whether a list is sorted [6]. Feige *et al.* [12] studied a probabilistic model and presented a sorting algorithm correct with probability at least $(1 - q)$ that requires $\Theta(n \log(n/q))$ comparisons. Lies are well suited at modeling transient ALU failures, such as comparator failures. Since memory data get never corrupted in the liar model, fault-tolerant algorithms may exploit query replication strategies. We remark that this is not the case in faulty memories.

Fault-Tolerant Sorting Networks. Destructive faults have been first investigated in the context of fault-tolerant sorting networks [2,21,22,34], in which comparators can be faulty and can possibly destroy one of the input values. Asaf and Upfal [2] present an $O(n \log^2 n)$ -size sorting network tolerant (with high probability) to random destructive faults. Later, Leighton and Ma [21] proved

that this bound is tight. The Assaf-Upfal network makes $\Theta(\log n)$ copies of each item, using data replicators that are assumed to be fault-free.

Parallel Computing with Memory Faults. Multiprocessor platforms are even more prone to hardware failures than uniprocessor computers. A lot of research has been thus devoted to deliver general simulation mechanisms of fully operational parallel machines on their faulty counterparts. The simulations designed in [8,9,10,17] are either randomized or deterministic, and operate with constant or logarithmic slowdown, depending on the model (PRAM or Distributed Memory Machine), on the nature of faults (static or dynamic, deterministic or random), and on the number of available processors. Some of these works also assume the existence of a special fault-detection register that makes it possible to recognize memory errors upon access to a memory location.

Implementing Data Structures in Unreliable Memory. In many applications such as file system design, it is very important that the implementation of a data structure is resilient to memory faults and provides mechanisms to recover quickly from erroneous states. Unfortunately, many pointer-based data structures are highly non-resilient: losing a single pointer makes the entire data structure unreachable. This problem has been addressed in [3], providing fault-tolerant versions of stacks, linked lists, and binary search trees: these data structures have a small time and space overhead with respect to their non-fault-tolerant counterparts, and guarantee that only a limited amount of data may be lost upon the occurrence of memory faults.

Blum et al. [5] considered the following problem: given a data structure residing in a large unreliable memory controlled by an adversary and a sequence of operations that the user has to perform on the data structure, design a checker that is able to detect any error in the behavior of the data structure while performing the user's operations. The checker can use only a small amount of reliable memory and can report a buggy behavior either immediately after an errant operation (on-line checker) or at the end of the sequence (off-line checker). Memory checkers for random access memories, stacks and queues have been presented in [5], where lower bounds of $\Omega(\log n)$ on the amount of reliable memory needed in order to check a data structure of size n are also given.

3 Designing Resilient Algorithms

Memory faults alter in an unpredictable way the correct values that should be stored in memory locations. Due to such faults, we cannot assume that the value read from a memory location is the same value that has been previously written in that location. If the algorithm is not prepared to cope with memory faults, it may take wrong steps upon reading of corrupted values and errors may propagate over and over. Consider for instance mergesort: during the merge step, errors may propagate due to corrupted keys having value larger than the correct one. Even in the presence of very few faults, in the worst case as many as $\Theta(n)$ keys may be out of order in the output sequence, where n is the number of keys to be

merged. There are even more subtle problems with recursive implementations: if some parameter or local variable in the recursion stack (e.g., an array index) gets corrupted, the mergesort algorithm may recurse on wrong subarrays and entire subsequences may remain unordered.

3.1 The Faulty Memory Model

Memory faults may happen at any time during the execution of an algorithm, at any memory location, and even simultaneously. The last assumption is motivated by the fact that an entire memory bank may dismiss to work properly, and thus all the data contained in it may get lost or corrupted at the same time. In order to model this scenario, in [14] we introduced a *faulty-memory random access machine*, i.e., a random access machine whose memory locations may experience memory faults which corrupt their content. In this model corrupted values are indistinguishable from correct ones. We also assumed that the algorithms can exploit only $O(1)$ *reliable* memory words, whose content gets never corrupted: this is not restrictive, since at least registers can be considered fault-free.

Let δ denote an upper bound on the total number of memory faults that can happen during the execution of an algorithm (note that δ may be a function of the instance size). We can think of the faulty-memory random access machine as controlled by a malicious adaptive adversary: at any point during the execution of an algorithm, the adversary can introduce an arbitrary number of faults in arbitrary memory locations. The only adversary's constraint is not to exceed the upper bound δ on the number of faults. If the algorithm is randomized, we assume that the adversary has no information about the sequence of random values.

In the faulty-memory model described above, if each value were replicated k times, by majority techniques we could easily tolerate up to $(k - 1)/2$ faults; however, the algorithm would present a multiplicative overhead of $\Theta(k)$ in terms of both space and running time. This implies, for instance, that in order to be resilient to $O(\sqrt{n})$ faults, a sorting algorithm would require $O(n^{3/2} \log n)$ time and $O(n^{3/2})$ space. The space may be improved using error-correcting codes, but at the price of a higher running time.

3.2 Resilient Sorting and Searching

It seems natural to ask whether it is possible to design algorithms that do not exploit data replication in order to achieve resilience to memory faults: i.e., algorithms that do not wish to recover corrupted data, but simply to be correct on uncorrupted data, without incurring any time or space overhead. More formally, in [14] we considered the following resilient sorting and searching problems.

- *Resilient sorting*: we are given a set of n keys that need to be sorted. The values of some keys may be arbitrarily corrupted during the sorting process. The problem is to order correctly the set of uncorrupted keys.
- *Resilient searching*: we are given a sequence of n keys on which we wish to perform membership queries. The keys are stored in increasing order, but

some keys may be corrupted and thus may occupy wrong positions in the sequence. Let x be the key to be searched for. The problem is either to find a key equal to x , or to determine that there is no correct key equal to x .

In both cases, this is the best that we can achieve in the presence of memory faults. For sorting, we cannot indeed prevent keys corrupted at the very end of the algorithm execution from occupying wrong positions in the output sequence. For searching, memory faults can make the searched key x appear or disappear in the sequence at any time.

We remark that one of the main difficulties in designing efficient resilient sorting and searching algorithms derives from the fact that positional information is no longer reliable in the presence of memory faults: for instance, when we search an array whose correct keys are in increasing order, it may be still possible that a faulty key in position i is smaller than some correct key in position j , $j < i$, thus guiding the search towards a wrong direction.

Known Results and Techniques. In [14] we proved both upper and lower bounds on resilient sorting and searching. These results are shown for deterministic algorithms that do not make use of data replication and use only $O(1)$ words of reliable memory. We remark that a constant-size reliable memory may be even not sufficient for a recursive algorithm to work properly: parameters, local variables, return addresses in the recursion stack may indeed get corrupted. This is however not a problem if the recursion can be simulated by an iterative implementation using only a constant number of variables.

With respect to sorting, we proved that any resilient $O(n \log n)$ comparison-based deterministic algorithm can tolerate the corruption of at most $O(\sqrt{n \log n})$ keys. This lower bound implies that, if we have to sort in the presence of $\omega(\sqrt{n \log n})$ memory faults, then we should be prepared to spend more than $O(n \log n)$ time. We also designed a resilient $O(n \log n)$ comparison-based sorting algorithm that tolerates $O((n \log n)^{1/3})$ memory faults. The algorithm is based on a bottom-up iterative implementation of mergesort and hinges upon the combination of two merging subroutines with quite different characteristics. The first subroutine requires optimal linear time, but it may be unable to sort correctly all the uncorrupted keys (i.e., some correct elements may be in a wrong position in the output sequence). Such errors are recovered by using the second subroutine: this procedure may require more than linear time in general, but is especially efficient when applied to unbalanced sequences. The crux of the approach is therefore to make the disorder in the sequence returned by the first subroutine proportional to the number of memory faults that happened during its execution: this guarantees that the shorter sequence received as input by the second subroutine will have a length proportional to the actual number of corrupted keys, thus limiting the total running time. The gap between the upper and lower bounds for resilient sorting has been recently closed in [13], by designing an optimal resilient comparison-based sorting algorithm that can tolerate up to $O(\sqrt{n \log n})$ faults in $O(n \log n)$ time. In [13] we also prove that, in the special case of integer sorting, there is a randomized algorithm that can tolerate

up to $O(\sqrt{n})$ faults in expected linear time. No lower bound is known up to this time for resilient integer sorting.

With respect to searching, in [14] we designed an $O(\log n)$ time searching algorithm that can tolerate up to $O(\sqrt{\log n})$ memory faults and we proved that any $O(\log n)$ time deterministic searching algorithm can tolerate at most $O(\log n)$ memory faults. The lower bound has been extended to randomized algorithms in [13], where we also exhibit an optimal randomized searching algorithm and an almost optimal deterministic searching algorithm that can tolerate up to $O((\log n)^{1-\epsilon})$ memory faults in $O(\log n)$ worst-case time, for any small positive constant ϵ .

References

1. J. A. Aslam and A. Dhagat. Searching in the presence of linearly bounded errors. *Proc. 23rd ACM Symp. on Theory of Computing (STOC'91)*, 486–493, 1991.
2. S. Assaf and E. Upfal. Fault-tolerant sorting networks. *SIAM J. Discrete Math.*, 4(4), 472–480, 1991.
3. Y. Aumann and M. A. Bender. Fault-tolerant data structures. *Proc. 37th IEEE Symp. on Foundations of Computer Science (FOCS'96)*, 580–589, 1996.
4. J. Blömer and J.-P. Seifert. Fault based cryptanalysis of the Advanced Encryption Standard (AES). *Proc. 7th International Conference on Financial Cryptography (FC'03)*, LNCS 2742, 162–181, 2003.
5. M. Blum, W. Evans, P. Gemmell, S. Kannan and M. Naor. Checking the correctness of memories. *Proc. 32th IEEE Symp. on Foundations of Computer Science (FOCS'91)*, 1991.
6. R. S. Borgstrom and S. Rao Kosaraju. Comparison based search in the presence of errors. *Proc. 25th ACM Symp. on Theory of Computing (STOC'93)*, 130–136, 1993.
7. P. M. Chen, E. L. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2), 145–185, 1994.
8. B. S. Chlebus, A. Gambin and P. Indyk. PRAM computations resilient to memory faults. *Proc. 2nd Annual European Symp. on Algorithms (ESA'94)*, LNCS 855, 401–412, 1994.
9. B. S. Chlebus, A. Gambin and P. Indyk. Shared-memory simulations on a faulty-memory DMM. *Proc. 23rd International Colloquium on Automata, Languages and Programming (ICALP'96)*, 586–597, 1996.
10. B. S. Chlebus, L. Gasieniec and A. Pelc. Deterministic computations on a PRAM with static processor and memory faults. *Fundamenta Informaticae*, 55(3-4), 285–306, 2003.
11. A. Dhagat, P. Gacs, and P. Winkler. On playing “twenty questions” with a liar. *Proc. 3rd ACM-SIAM Symp. on Discrete Algorithms (SODA'92)*, 16–22, 1992.
12. U. Feige, P. Raghavan, D. Peleg, and E. Upfal. Computing with noisy information. *SIAM Journal on Computing*, 23, 1001–1018, 1994.
13. I. Finocchi, F. Grandoni and G. F. Italiano. Optimal resilient sorting and searching in the presence of memory faults. Manuscript, 2005.
14. I. Finocchi and G. F. Italiano. Sorting and searching in the presence of memory faults (without redundancy). *Proc. 36th ACM Symposium on Theory of Computing (STOC'04)*, 101–110, 2004.

15. S. Hamdioui, Z. Al-Ars, J. Van de Goor, and M. Rodgers. Dynamic faults in Random-Access-Memories: Concept, faults models and tests. *Journal of Electronic Testing: Theory and Applications*, 19, 195–205, 2003.
16. M. Henzinger. The past, present and future of Web Search Engines. Invited talk. *31st Int. Coll. Automata, Languages and Programming*, Turku, Finland, July 12–16 2004.
17. P. Indyk. On word-level parallelism in fault-tolerant computing. *Proc. 13th Annual Symp. on Theoretical Aspects of Computer Science (STACS'96)*, 193–204, 1996.
18. R. H. Katz, D. A. Patterson and G. A. Gibson, *Disk system architectures for high performance computing*, Proceedings of the IEEE, 77(12), 1842–1858, 1989.
19. D. J. Kleitman, A. R. Meyer, R. L. Rivest, J. Spencer, and K. Winklmann. Coping with errors in binary search procedures. *Journal of Computer and System Sciences*, 20:396–404, 1980.
20. K. B. Lakshmanan, B. Ravikumar, and K. Ganesan. Coping with erroneous information while sorting. *IEEE Trans. on Computers*, 40(9):1081–1084, 1991.
21. T. Leighton and Y. Ma. Tight bounds on the size of fault-tolerant merging and sorting networks with destructive faults. *SIAM Journal on Computing*, 29(1):258–273, 1999.
22. T. Leighton, Y. Ma and C. G. Plaxton. Breaking the $\Theta(n \log^2 n)$ barrier for sorting with faults. *Journal of Computer and System Sciences*, 54:265–304, 1997.
23. T. C. May and M. H. Woods. Alpha-Particle-Induced Soft Errors In Dynamic Memories. *IEEE Trans. Elect. Dev.*, 26(2), 1979.
24. S. Muthukrishnan. On optimal strategies for searching in the presence of errors. *Proc. 5th ACM-SIAM Symp. on Discrete Algorithms (SODA'94)*, 680–689, 1994.
25. A. Pelc. Searching with known error probability. *Theoretical Computer Science*, 63, 185–202, 1989.
26. A. Pelc. Searching games with errors: Fifty years of coping with liars. *Theoretical Computer Science*, 270, 71–109, 2002.
27. B. Ravikumar. A fault-tolerant merge sorting algorithm. *Proc. 8th Annual Int. Conf. on Computing and Combinatorics (COCOON'02)*, LNCS 2387, 440–447, 2002.
28. A. Rényi. *A diary on information theory*, J. Wiley and Sons, 1994. Original publication: *Napló az információelméletéről*, Gondolat, Budapest, 1976.
29. S. Skorobogatov and R. Anderson. Optical fault induction attacks. *Proc. 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'02)*, LNCS 2523, 2–12, 2002.
30. Tezzaron Semiconductor. *Soft errors in electronic memory - a white paper*, URL: <http://www.tezzaron.com/about/papers/Papers.htm>, January 2004.
31. S. M. Ulam. *Adventures of a mathematician*. Scribners (New York), 1977.
32. A.J. Van de Goor. *Testing semiconductor memories: Theory and practice*, ComTex Publishing, Gouda, The Netherlands, 1998.
33. J. Von Neumann, Probabilistic logics and the synthesis of reliable organisms from unreliable components. In *Automata Studies*, C. Shannon and J. McCarty eds., Princeton University Press, 43–98, 1956.
34. A. C. Yao and F. F. Yao. On fault-tolerant networks for sorting. *SIAM Journal on Computing*, 14, 120–128, 1985.

From Balanced Graph Partitioning to Balanced Metric Labeling

Joseph Naor

Computer Science Department, Technion, Haifa 32000, Israel and Microsoft Research,
One Microsoft Way, Redmond, WA 98052
naor@cs.technion.ac.il

Abstract. The Metric Labeling problem is an elegant and powerful mathematical model capturing a wide range of classification problems that arise in computer vision and related fields. In a typical classification problem, one wishes to assign labels to a set of objects to optimize some measure of the quality of the labeling. The metric labeling problem captures a broad range of classification problems where the quality of a labeling depends on the pairwise relations between the underlying set of objects, as described by a weighted graph. Additionally, a metric distance function on the labels is defined, and for each label and each vertex, an assignment cost is given. The goal is to find a minimum-cost assignment of the vertices to the labels. The cost of the solution consists of two parts: the assignment costs of the vertices and the separation costs of the edges (each edge pays its weight times the distance between the two labels to which its endpoints are assigned). Metric labeling has many applications as well as rich connections to some well known problems in combinatorial optimization.

The *balanced metric labeling* problem has an additional constraint requiring that at most ℓ vertices can be assigned to each label, i.e., labels have *capacity*. We focus on the case where the given metric is uniform and note that it already captures various well-known balanced graph partitioning problems. We discuss (pseudo) approximation algorithms for the balanced metric labeling problem, and focus on several important techniques used for obtaining the algorithms. Spreading metrics have proved to be very useful for balanced graph partitioning and our starting point for balanced metric labeling is a linear programming formulation that combines an embedding of the graph in a simplex together with spreading metrics and additional constraints that strengthen the formulation. The approximation algorithm is based on a novel randomized rounding that uses both a randomized metric decomposition technique and a randomized label assignment technique. At the heart of our approach is the fact that only limited dependency is created between the labels assigned to different vertices, allowing us to bound the expected cost of the solution and the number of vertices assigned to each label, simultaneously.

Fearful Symmetries: Quantum Computing, Factoring, and Graph Isomorphism

Cristopher Moore

Computer Science Department and Department of Physics and Astronomy,
University of New Mexico, Albuquerque NM 87131
`moore@cs.unm.edu`

Historically, the idea of symmetry has played a much greater role in physics than in computer science. While computer scientists typically work against an adversary, physicists are brought up to believe in the benevolence of nature, and to believe that the answers to natural questions are often as simple—and as symmetric—as they possibly could be. Indeed, symmetry is intimately linked to every branch of physics, from classical conservation laws to elementary particles to special and general relativity.

Recently, symmetry has appeared in a new branch of computer science, namely quantum computing. The mathematical embodiment of symmetry is a *group*, and Shor’s celebrated factoring algorithm works by looking for symmetries in a particular function defined on the multiplication group \mathbb{Z}_n^* of integers mod n . Similarly, if we could find symmetries in functions on the group S_n of permutations of n objects, we could solve the Graph Isomorphism problem.

Without relying on any of the physics behind quantum computing, I will briefly describe Shor’s algorithm, and how it uses the quantum Fourier transform to find periodicities in a function. I will then explain how both factoring and Graph Isomorphism can be reduced to the *Hidden Subgroup Problem*, and what it means to take the Fourier transform of a function on a nonabelian group (that is, a group where ab and ba are not necessarily the same) such as S_n . This leads us to the rich and beautiful world of *representation theory*, which has been used to prove that random walks on groups—such as shuffling a pack of cards—are rapidly mixing.

After we carry out the quantum Fourier transform, there are various types of measurements we can perform: weak measurement, where we learn just the “name” of a representation; strong measurement, where we observe a vector inside this representation; and even *entangled* measurements over the tensor product of states resulting from multiple queries. An enormous amount of progress has been made over the past year on how much computational power these types of measurement give us, and which cases of the Hidden Subgroup Problem they allow us to solve. Much work remains to be done, and I will conclude by trying to convey the flavor of this frontier in quantum algorithms.

Exploring an Unknown Graph Efficiently*

Rudolf Fleischer¹ and Gerhard Trippen²

¹ Fudan University, Shanghai Key Laboratory of Intelligent Information Processing,
Department of Computer Science and Engineering, Shanghai, China
`fleischer@acm.org`

² The Hong Kong University of Science and Technology, Hong Kong,
Department of Computer Science
`trippen@cs.ust.hk`

Abstract. We study the problem of exploring an unknown, strongly connected directed graph. Starting at some node of the graph, we must visit every edge and every node at least once. The goal is to minimize the number of edge traversals. It is known that the competitive ratio of online algorithms for this problem depends on the deficiency d of the graph, which is the minimum number of edges that must be added to make the graph Eulerian. We present the first deterministic online exploration algorithm whose competitive ratio is polynomial in d (it is $O(d^8)$).

1 Introduction

Exploring an unknown environment is a fundamental problem in online robotics. Exploration means to draw a complete map of the environment. Since all decisions where to proceed with the exploration are based on local (or partial) information, the exploration problem falls naturally into the class of *online algorithms* [7,12]. The quality of an online algorithm is measured by the *competitive ratio* which is the worst-case quotient of the length of the path traveled by the online algorithm and the length of the shortest path that can visit the entire environment.

The exploration problem has been studied for various restrictions on the online algorithm (power of local sensors, fuel consumption, etc.) and many types of environment: directed graphs with a single robot [1,10,16] or with cooperating robots [4], planar graphs when we only need to visit all nodes [15], undirected graphs when we must regularly return to the starting point [2], and polygonal environments with obstacles [9] or without obstacles [14]. Similarly, the search problem has been studied for some of these environments [3,5,6,18].

In this paper, we study the online exploration problem for strongly connected directed graphs. We denote a graph by $G = (V, E)$, where V is the set of n nodes

* The work described in this paper was partially supported by the RGC/HKUST Direct Allocation Grant DAG03/04.EG05 and by a grant from the German Academic Exchange Service and the Research Grants Council of Hong Kong Joint Research Scheme (Project No. G-HK024/02-II).

and E is the set of m edges. The *deficiency* d of the graph is the minimum number of edges that must be added to make the graph Eulerian. A node with more outgoing than incoming edges is a *source*, and a node with more incoming than outgoing edges is a *sink*.

Initially, we only know the starting node s . We do not know n , m , d , or any part of G except the outdegree of s . We can traverse edges only from tail to head. We say an edge (node) becomes *visited* when we traverse (discover) it for the first time. At any time, we know all the visited nodes and edges, and for any visited node we know the number of unvisited edges leaving the node, but we do not know the endpoints of the unvisited edges.

The cost of an exploration tour is defined as the number of traversed edges. The Chinese Postman Problem describes the offline version of the graph exploration problem [11]. For either undirected or directed graphs the problem can be solved in polynomial time. For mixed graphs with both undirected and directed edges the problem becomes NP-complete [17]. We always need at least m edge traversals to visit all edges (and nodes) of a graph, and there exist graphs with deficiency d that require $\Omega(dm)$ edge traversals [1].

In the online problem, we may have the choice of leaving the current node on a visited or an unvisited edge. If there is no outgoing unvisited edge, we say we are *stuck* at the current node.

For undirected graphs, the online exploration problem can easily be solved by DFS, which is optimally 2-competitive [10]. The problem becomes more difficult for strongly connected directed graphs. Deng and Papadimitriou [10] gave an online exploration algorithm that may need $d^{O(d)}m$ edge traversals. They also gave an optimal 4-competitive algorithm for the special case $d = 1$, and they conjectured that in general there might be a $poly(d)$ -competitive online exploration algorithm. For large d , i.e., $d = \Omega(n^c)$ for some $c > 0$, this is true because DFS never uses more than $O(\min\{nm, m + dn^2\})$ edge traversals [16] (as actually do most natural exploration algorithms [1]). The best known lower bounds are $\Omega(d^2m)$ edge traversals for deterministic and $\Omega(d^2m/\log d)$ edge traversals for randomized online algorithms [10].

Albers and Henzinger [1] gave a first improvement to the Deng and Papadimitriou algorithm. They presented the **Balance** algorithm which can explore a graph of deficiency d with $d^{O(\log d)}m$ edge traversals. They also showed that this bound is tight for their algorithm. To show the difficulty of the problem they also gave lower bounds of $2^{\Omega(d)}m$ edge traversals for several natural exploration algorithms as **Greedy**, **Depth-First**, and **Breadth-First**. For **Generalized Greedy** they gave a lower bound of $d^{\Omega(d)}m$ edge traversals.

No randomized online graph exploration algorithm has ever been analyzed, but in an earlier paper we presented an experimental study of all known deterministic and randomized algorithms [13]. The experiments basically show that on random graphs the simple greedy strategies work very well, in particular much better than the deterministic algorithms with better worst-case bounds.

In this paper we give the first $poly(d)$ -competitive online graph exploration algorithm. The main idea of our algorithm is to finish chains (i.e., newly discovered

paths) in a breadth-first-search manner, and to recursively split off subproblems that can be dealt with independently because they contain new cycles which we can use to relocate without using other edges. We prove that our algorithm needs at most $O(d^8 m)$ edge traversals.

This paper is organized as follows. In Section 2, we review some of the old results which we use for our new algorithm in Section 3. We analyze the competitive ratio of the algorithm in Section 4, and we close with some remarks in Section 5.

2 Basics

If a node has no unvisited outgoing edges, we call it *finished*. Similarly, a path is *finished* if it contains no unfinished nodes. Note that we must traverse any path at least twice. The first time when we *discover* it, and then a second time to finish all nodes on the path. But some paths must be traversed more often. If we get stuck while exploring some new path or after just finishing a path, we must *relocate* to another unfinished path. Bounding these relocation costs is the main difficulty in the design of an efficient online exploration algorithm.

A graph is *Eulerian* if there exists a round trip visiting each edge exactly once. The simple greedy algorithm used for testing whether a graph is Eulerian [8, Problem 22-3] will explore any unknown Eulerian graph with at most $2m$ edge traversals. In fact, this algorithm is optimal [10]. We simply take an unvisited edge whenever possible. If we get stuck (i.e., reach a node with no outgoing unvisited edges), we consider the closed walk (it must be a cycle) just visited and retrace it, stopping at nodes that have unvisited edges, applying this algorithm recursively from each such node. This algorithm has no relocation costs (because we never get stuck). In fact, it is easy to see that no edge will be traversed more than twice. The reason for this is that the recursive explorations always take place in completely new parts of the graph. Our new algorithm for arbitrary graphs will use a similar feature.

For non-Eulerian graphs, when exploring a new path the start node will become a new source of the currently known subgraph and the node where we get stuck is either a newly discovered sink or a source of the currently known subgraph (which therefore is not a source of G). Therefore, at any time the explored subgraph has an equal number of sources and sinks (counting multiple sources and sinks with their respective multiplicity). We maintain a matching of the sources and sinks in the form of *tokens* [1]. Initially, each sink of G has a token uniquely describing the sink (for example, we could use numbers $1, \dots, d$, because the number of tokens must equal the deficiency of G [1]). We can only get stuck when closing a cycle (as in the Eulerian case), or at a node with a token. In the former case, we proceed as in the Eulerian greedy algorithm. In the latter case, we say we *trigger* the token and we move it to the start node of the path we just followed before getting stuck (which is a new sink of the explored subgraph).

The concatenation of all paths triggering a token τ , which is a path from the current token node to its corresponding sink, is denoted by P_τ . Initially, P_τ is

an empty path. Similarly, τ_P denotes the token at the start node of a path P (if there is a token). We note that tokens are not invariably attached to a certain path. Instead, we will often rearrange tokens and their paths.

Although G is strongly connected, it is unavoidable that in the beginning our explored subgraph is not strongly connected. Albers and Henzinger described in [1, Section 3.3] a general technique that allows any exploration algorithm to assume w.l.o.g. that at any time the explored subgraph of G is strongly connected. In particular in the beginning we know an initial cycle C_0 containing s . For this assumption, we have to pay a penalty of a factor of d^2 in the competitive ratio. Basically, the algorithm must be restarted d times at a different start node, each time with a penalty factor of d for not knowing a strongly connected subgraph, before we can guarantee that our known subgraph is strongly connected. We will also make this assumption. To be more precise, we assume that in the beginning we know an initial cycle C_0 containing the start node s , and we can reach s from any token that we may discover.

3 The Algorithm

3.1 High-Level Description

We divide the graph into *clusters*. Intuitively, a cluster is a strongly connected subgraph of G that can be explored independently of the rest of G (although clusters created at different stages of the algorithm may overlap). If a cluster L was created while we were working on cluster K , then we say L is a *subcluster* of K , and L is a child of K in the *cluster tree* S . Each cluster K has an *initial cycle* C_K . The root cluster K_0 of S corresponds to the initial cycle C_0 containing the start node s . Fig. 1 shows a generic partition of a graph into recursive subclusters.

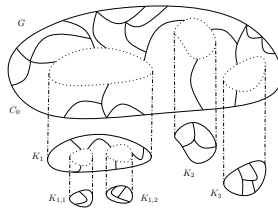


Fig. 1. A generic partition of a graph into recursive subclusters

From [1] we borrow the concept of a *token tree*. Actually, we need a token forest. For each cluster K there is a token tree T_K , with the initial cycle C_K of the cluster as root (this node does not correspond to a token). We denote the top-level token tree corresponding to K_0 by T_0 . For each token there will be one node in one of the token trees of the token forest. Also, for each cluster there will be one node, called a *cluster node*, in one token tree.

When we are currently in the process of finishing a path P_τ and trigger another token π in the same token tree, then we say τ becomes the *owner* of π . The token trees represent this ownership relation: a parent token is the owner of all its child tokens. The children are always ordered from left to right in the order in which they appear on the parent token path. If π belongs to a different token tree than τ , then π will become the child of some cluster node containing τ , but the token will as usually move to v_τ , the node where we started exploring the new path that ended in π .

If π is a predecessor of τ in the token tree we cannot simply change the ownership. Instead, we create a new subcluster L whose initial cycle C_L intuitively spans the path from π to τ in the token tree. If C_L uses the initial part of a token path P_γ (i.e., γ is lying on C_L) at the time L is created as a subcluster of K (there will always be at least one such token), γ becomes an *active member* of L and its corresponding token tree node will move from T_K to T_L . In K , γ will become an *inactive member*. Also, T_K will get a new cluster node representing L containing all the new active member tokens of L . Thus, a token can be member of several clusters, but only the most recently created of these clusters will contain a node for the token in its token tree. Initially, all tokens are active members of the initial cluster K_0 .

The number of tokens in a cluster node is the *weight* of this node. A cluster node is always connected to its parent by an edge of length equal to its weight.

We will maintain the invariant that any token path P_τ always consists of an unfinished subpath followed by a finished subpath, where either subpath could be empty. Since the finished subpath is not relevant for our algorithm or the analysis, we will from now on use P_τ to denote only the unfinished subpath which we call the *token path* of τ . We say a token is *finished* if its token path is empty. When we start finishing a path, called the *current chain*, we always finish it completely before doing something else, i.e., if we explore a new path and get stuck we immediately return to the current chain and continue finishing it. We say a cluster is *finished* if all nodes of its token tree are finished, otherwise it is *active*.

The clusters will be processed recursively, i.e., if a new cluster L is created while we are working in cluster K we will immediately turn to L after finishing the current chain (in K) and only resume the work in K after L has been finished. Thus, in S exactly the clusters on the rightmost path down from the root are active, all other clusters are finished.

When we start working on a new cluster K , we process its token tree T_K BFS-like. Usually, the initial cycle (i.e., the root of T_K) will be partially unfinished when we enter K , so this is usually the first chain to finish. When the next token in T_K (according to BFS) is τ and P_τ is unfinished, P_τ becomes the current chain that we are going to finish next. Then we continue with the next token to the right, or the leftmost token on the next lower level if τ was the rightmost token on its level. T_K will dynamically grow while we are working on K , but only below the level of the current chain.

We classify the edges of a cluster as native, adopted, or inherited. If a cluster K is created, it *inherits* the edges on the initial cycle C_K from its parent cluster. An edge is *native* to cluster K if it was first visited while traversing a new path that triggered a token whose token tree node was then moved to T_K (note that we can trigger tokens in a cluster K while working in some other cluster). Only active clusters can acquire new native edges, and edges are always finished while we are working in the cluster to which they are native. If a cluster is finished, all its native edges are *adopted* by its parent cluster.

A token π is a *forefather* of a token τ if there exists a cluster L such that π and τ are members of L , and either π or a cluster node containing π lies on the path from the root to τ or a cluster node containing τ . Note that an ordinary ancestor of a node in a token tree is also a forefather.

3.2 Finishing a Chain

Assume we are currently working in cluster K . To finish the current chain $C = P_\tau$ we relocate to the node currently holding the token τ and then move along C . Whenever we reach an unfinished node v we explore a new path P by repeatedly choosing arbitrary unvisited outgoing edges until we get stuck at some node w . We distinguish three cases.

- (1) If $v = w$, then we cut C at v , add P to C (see Fig. 2), and continue finishing C at v , first finishing the subpath P . This is like the greedy Eulerian algorithm.

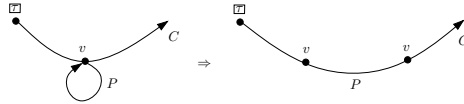


Fig. 2. Case (1): we found a loop P in C , and we extend C by P

- (2) If $v \neq w$ and w holds a token π which is not a forefather of τ (if w holds several such tokens, we choose an arbitrary one), we extend the token path P_π by prepending the new path P and moving π to v (see Fig. 3).

For the update of the token trees we have to consider a few cases. Let L be the lowest ancestor (in S) of the current cluster K in which both π and τ are members. If τ is an active member of L , let z_τ be the node τ in L ; otherwise, let z_τ be the cluster node in L containing τ . If π is an active member of L , then we move in T_L the node π together with its subtree as a new child below z_τ ; otherwise, let z_π be the cluster node in L containing π corresponding to some subcluster L_π . We *destroy* L_π by rearranging the tokens and token paths in L_π such that π becomes the root of the token tree of L_π (instead of the initial cycle). Then we can move this new token tree as a new child below z_τ . All native edges of L_π are adopted by L .

In all cases, if z_τ is a cluster node, we also keep as additional information in node π its true ownership (in case the subcluster z_τ needs to be destroyed later, see below).

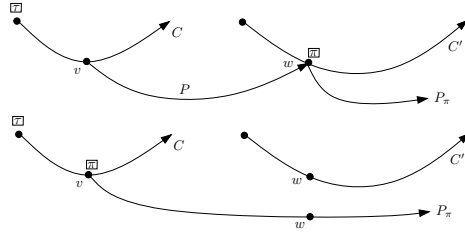


Fig. 3. Case (2): we extend the token path P_π starting in w by prepending P to P_π . The token π moves from w to v , and C becomes the new owner of π .

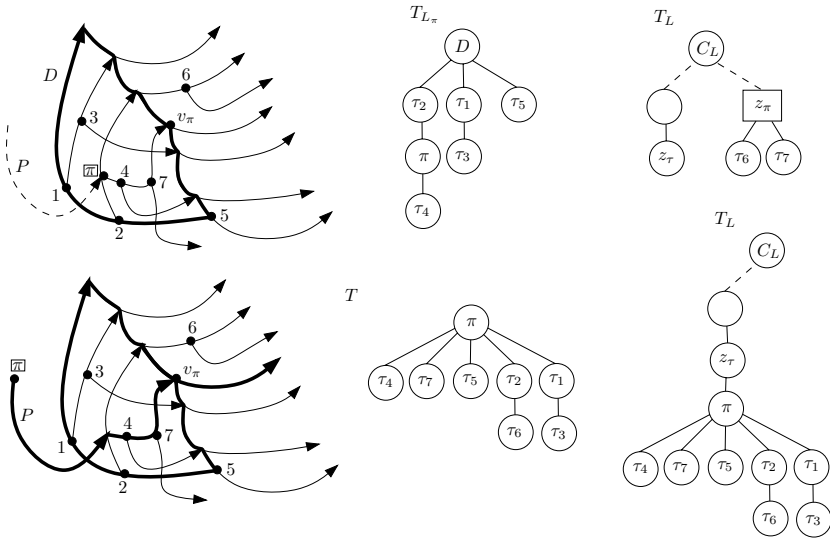


Fig. 4. How to destroy a cluster. **Top:** The numbers $1, \dots, 7$ denote the tokens τ_1, \dots, τ_7 . Since τ_6 and τ_7 correspond to paths leading outside L_π , they are not members of L_π ; instead, they are members of L and children of the cluster node z_π (which contains π) in some ancestor cluster L . A new path P from somewhere outside L_π triggers π in L_π . v_π is the position of π at the time L_π was created with initial cycle D (in bold). **Bottom:** π has moved to the start point of P . L_π was rearranged to yield a new tree T with root π . Note that this tree now contains the tokens τ_6 and τ_7 . Finally, T becomes a new child of z_τ in T_L .

The crucial observation when destroying L_π is that the token path P_π was crossing the initial cycle D of L_π in some node v_π at the time L_π was created. But then we can include the cycle D into P_π and create a new token tree T for L_π with root π by cutting off the subtree rooted at π , moving all children of D as children below π , and deleting the node D . We can do this because there exists a path only using edges of L_π (and its recursive subclusters) from D to every member of L_π . If z_π happens to have children, we must also

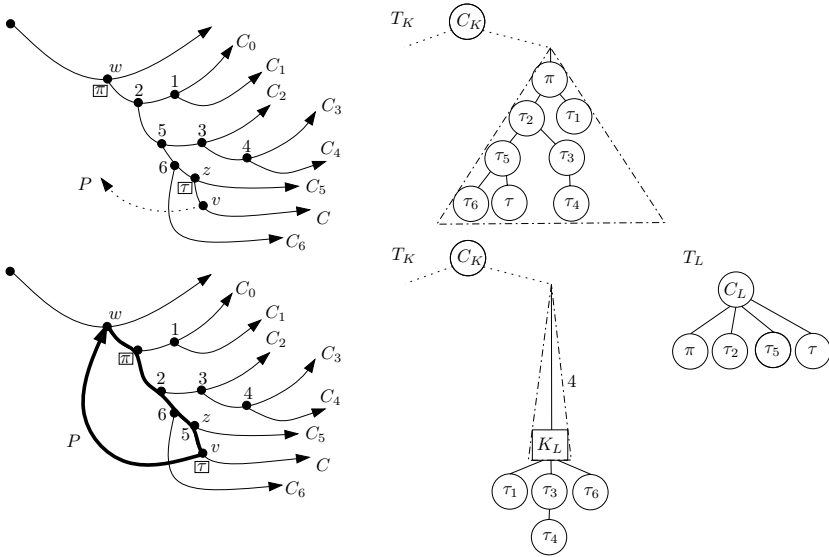


Fig. 5. Creating a cluster. **Top:** The new path P starting at v triggers π , which is an ancestor of τ in T_K . **Bottom:** We delete π and τ from T_K and create a subcluster L with initial cycle $D = (w, \tau_2, \tau_5, z, v, w)$ (in bold). The cluster node K_L in T_K has weight 4.

include these children in T by adding them as children of their owners. See Fig. 4 for an example.

Then we relocate to v and continue finishing the current chain C .

- (3) If $v \neq w$ and w holds a token π which is a forefather of τ (if w holds several such tokens, we choose an arbitrary one), we cannot move the subtree rooted at π below τ . Instead, we either create a new cluster or extend the current cluster. Note that it can happen that $\pi = \tau$.

We distinguish a few cases. If π is an ancestor of τ in T_K , then we just closed a cycle D containing v and w . The part of D from w to v consists of edges that are initial parts of the token paths of all predecessors of τ up to π , see Fig. 5. We shorten all these token paths by moving the respective tokens along D to the node that originally held the next token. Thus, all these token paths now start on D . We create a new subcluster L with initial cycle $C_L = D$. All the member tokens of L become children of D . Note that these tokens lie on a path from π to τ in T_K . We cut off the subtree rooted at π and replace it by a new cluster node K_L containing the member tokens of L . The edge connecting this new node to T_K has length equal to its weight (i.e., the number of member tokens of L). Thus, it is on the same level as previously τ . If any of the member tokens of L had children in T_K that did not become member of L , we move these subtrees as children below K_L . The reason for this is that these subtrees correspond to token paths that do not end in L , so we should finish these paths within K and not within L (otherwise, relocation might become too expensive).

The description above assumed that none of the nodes in T_K between π and τ are cluster nodes. If there are cluster nodes on this path, we must first destroy these clusters (similarly as in case (2)) before we can create the new subcluster. Fig. 6 shows an example.

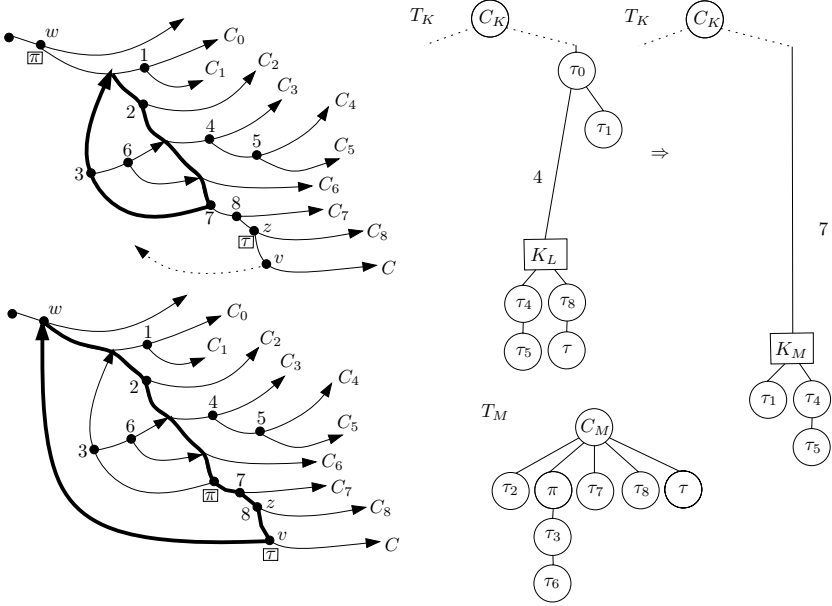


Fig. 6. Creating a cluster where the initial cycle contains a cluster node K_L of weight 4. After destroying L , we can create a new cluster M of weight 7.

Even more complicated is the case if π is contained in a cluster node in T_K , or if π is an active member of a cluster higher up in the recursion than the current cluster. In the latter case we actually extend the current cluster by including the forefather tokens up to π . Details of these constructions are left to the full version of this paper.

Before we start working recursively on the new subcluster L we must first finish the current chain C (in the current cluster K), i.e., we relocate to v . If we trigger a token that is a forefather of K_D or a token contained in K_D , we must *extend* L , see Fig. 7 for an example of the latter case. In that case, the chain that just triggered the token will be extended so that it starts on C_L , i.e., it will now contain some part of C . Note that these edges must have been recently discovered in K , i.e., originally they have been native to K , but now they become native to L instead of K . This is equivalent to what would happen if we discovered this new chain P_{τ_2} while finishing D in L .

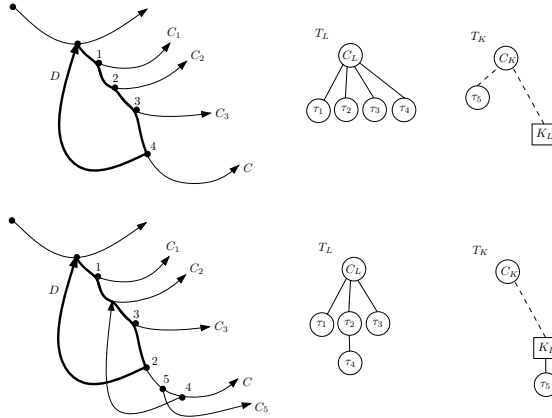


Fig. 7. Top: While finishing C (with token τ_4), we just created a subcluster L with initial chain D (in bold). **Bottom:** Before working on L we must finish C , where we trigger τ_5 and then τ_2 on D . We switch τ_2 and τ_4 , making the subpath of C between the two nodes part of P_{τ_2} and its edges native to L . Then we continue finishing C which now starts at τ_4 .

4 The Analysis

To analyze the total relocation cost, we count how often an edge can be used in a relocation.

Lemma 1. *At any time, any token tree T_K has at most d levels below the root, and each level can contain at most d nodes.* □

Lemma 2. *If we are currently working at a node on level h , then all nodes above level h are finished. In particular, we never move an unfinished node onto level h or above.* □

Lemma 3. *A cluster can have at most $O(d^3)$ subclusters.*

Proof. We finish at most d^2 chains by Lemmas 1 and 2. When we have finished a chain, this chain may have induced at most one subcluster. This subcluster can be extended at most $d - 1$ times. □

Lemma 4. *A cluster can only inherit native or adopted edges from its parent cluster.*

Proof. Edges on the initial cycle are inherited from the parent, but never inherited by a subcluster. □

Lemma 5. *An edge can be native, inherited, or adopted in at most $O(d^3)$ clusters.*

Proof. An edge e can be native to only one cluster. e can only be adopted if its cluster is finished or destroyed. If all member tokens of a cluster K move into a subcluster L , K can never be destroyed. As soon as L is finished or destroyed, K is also finished, i.e., K adopts L 's edges but it will not use them for relocations in K . Thus, each time e is adopted by an active cluster, this new cluster must have more member tokens than the previous one, i.e., e can be adopted by at most d active clusters higher up in the recursion tree. In each of these d clusters, e can be inherited by at most $O(d^2)$ subclusters because an edge can only appear in one subcluster on each of the d levels of the token tree, which then may be extended $d-1$ times. It cannot be inherited a second time by any sub-subclusters by Lemma 4. \square

Lemma 6. *Not counting relocations within subclusters, at most $O(d^3)$ relocations suffice to finish a cluster.*

Proof. We finish the token tree using BFS. We have d^2 relocations after having finished a chain. While we are finishing a chain, we may trigger at most d different tokens. It may happen that the same token is triggered several times in which case it moves along the current chain. The cost of all these relocations is the same as if we had only triggered the token once, namely at the last instance. Thus, we have at most d^3 relocations due to getting stuck after exploring a new path. By Lemma 3, we must also relocate to and from $O(d^3)$ subclusters. \square

Lemma 7. *Each edge is used in at most $O(d^6)$ relocations.*

Proof. This follows basically from Lemmas 5 and 6, except for those relocations where we trigger a token outside of the current cluster. In that case, we must follow some path down the cluster tree to relocate to the current chain. However, this happens only once for each token because afterwards the token is in the current cluster and we can relocate locally (and these costs are already accounted for). All the relocations then form an inorder traversal of the cluster tree, i.e., each edge in each cluster is used only a constant number of times. Since an edge can appear in $O(d^3)$ clusters by Lemma 5 and we have d tokens, each edge is used at most $O(d^4)$ times for these relocations. \square

Theorem 1. *Our algorithm is $O(d^8)$ -competitive.*

Proof. The assumption that we always know a strongly connected explored subgraph costs us another factor of $O(d^2)$ in the competitive ratio [1]. \square

5 Conclusions

We presented the first $poly(d)$ -competitive algorithm for exploring strongly connected digraphs. The bound of $O(d^8)$ is not too good and can probably be improved. For example, the d^2 -penalty for the start-up phase seems too high because some of the costs are already accounted for in other parts of our analysis. Lemma 5 seems overly pessimistic. If a cluster is extended, it does not really

incur additional costs for an edge, we could just make the cluster larger and continue as if nothing had happened. Also, destroying a cluster when a member gets triggered seems to be unnatural.

References

1. S. Albers and M. R. Henzinger. Exploring unknown environments. *SIAM Journal on Computing*, 29(4):1164–1188, 2000.
2. B. Awerbuch, M. Betke, R. Rivest, and M. Singh. Piecemeal graph exploration by a mobile robot. *Information and Computation*, 152(2):155–172, 1999.
3. E. Bar-Eli, P. Berman, A. Fiat, and P. Yan. Online navigation in a room. *Journal of Algorithms*, 17(3):319–341, 1994.
4. M. A. Bender and D. K. Slonim. The power of team exploration: Two robots can learn unlabeled directed graphs. In *Proceedings of the 35th Symposium on Foundations of Computer Science (FOCS'94)*, pages 75–85, 1994.
5. P. Berman, A. Blum, A. Fiat, H. Karloff, A. Rosén, and M. Saks. Randomized robot navigation algorithms. In *Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms (SODA'96)*, pages 75–84, 1996.
6. A. Blum, P. Raghavan, and B. Schieber. Navigating in unfamiliar geometric terrain. *SIAM Journal on Computing*, 26(1):110–137, 1997.
7. A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, Cambridge, England, 1998.
8. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, and London, England, 2. edition, 2001.
9. X. Deng, T. Kameda, and C. H. Papadimitriou. How to learn an unknown environment. *Journal of the ACM*, 45:215–245, 1998.
10. X. Deng and C. H. Papadimitriou. Exploring an unknown graph. *Journal of Graph Theory*, 32:265–297, 1999.
11. J. Edmonds and E. L. Johnson. Matching, Euler tours and the Chinese postman. *Mathematical Programming*, 5:88–124, 1973.
12. A. Fiat and G. Woeginger, editors. *Online Algorithms — The State of the Art*. Springer Lecture Notes in Computer Science 1442. Springer-Verlag, Heidelberg, 1998.
13. R. Fleischer and G. Trippen. Experimental studies of graph traversal algorithms. In *Proceedings of the 2nd International Workshop on Experimental and Efficient Algorithms (WEA'03)*. Springer Lecture Notes in Computer Science 2647, pages 120–133, 2003.
14. F. Hoffmann, C. Icking, R. Klein, and K. Kriegel. The polygon exploration problem. *SIAM Journal on Computing*, 31(2):577–600, 2001.
15. B. Kalyanasundaram and K. R. Pruhs. Constructing competitive tours from local information. *Theoretical Computer Science*, 130:125–138, 1994.
16. S. Kwek. On a simple depth-first search strategy for exploring unknown graphs. In *Proceedings of the 5th Workshop on Algorithms and Data Structures (WADS'97)*. Springer Lecture Notes in Computer Science 1272, pages 345–353, 1997.
17. C. H. Papadimitriou. On the complexity of edge traversing. *Journal of the ACM*, 23(3):544–554, 1976.
18. C. H. Papadimitriou and M. Yannakakis. Shortest paths without a map. *Theoretical Computer Science*, 84:127–150, 1991.

Online Routing in Faulty Meshes with Sub-linear Comparative Time and Traffic Ratio

Stefan Rührup * and Christian Schindelhauer **

Heinz Nixdorf Institute, University of Paderborn, Germany
{sr, schindel}@uni-paderborn.de

Abstract. We consider the problem of routing a message in a mesh network with faulty nodes. The number and positions of faulty nodes is unknown. It is known that a flooding strategy like expanding ring search can route a message in the minimum number of steps h while it causes a traffic (i.e. the total number of messages) of $\mathcal{O}(h^2)$. For optimizing traffic a single-path strategy is optimal producing traffic $\mathcal{O}(p + h)$, where p is the perimeter length of the barriers formed by the faulty nodes. Therefore, we define the comparative traffic ratio as a quotient over $p + h$ and the competitive time ratio as a quotient over h . Optimal algorithms with constant ratios are known for time and traffic, but not for both. We are interested in optimizing both parameters and define the combined comparative ratio as the maximum of competitive time ratio and comparative traffic ratio. Single-path strategies using the right-hand rule for traversing barriers as well as multi-path strategies like expanding ring search have a combined comparative ratio of $\Theta(h)$. It is an open question whether there exists an online routing strategy optimizing time and traffic for meshes with an unknown set of faulty nodes. We present an online strategy for routing with faulty nodes providing sub-linear combined comparative ratio of $h^{\mathcal{O}\left(\sqrt{\frac{\log \log h}{\log h}}\right)}$.

1 Introduction

The problem of routing in mesh networks has been extensively studied. Even if the mesh contains faulty nodes, standard routing techniques can be applied. But as an *online problem*, where the location of faulty nodes is unknown, this task becomes challenging, because some communication overhead has to be invested for the exploration of the network and it is not clear, how much exploration is worth the effort. With this formulation of the problem, online route discovery gets related to graph exploration and motion planning problems. The definition of an appropriate measure for analyzing the efficiency and the design of an algorithm that beats known routing techniques is the contribution of this paper.

* DFG Graduiertenkolleg 776 “Automatic Configuration in Open Systems”.

** Supported by the DFG Sonderforschungsbereich 376: “Massive Parallelität: Algorithmen, Entwurfsmethoden, Anwendungen.” and by the EU within the 6th Framework Programme under contract 001907 “Dynamically Evolving, Large Scale Information Systems” (DELIS).

We consider the problem of routing (i.e. route discovery) in a mesh network with faulty nodes. If a node fails, then only the direct neighbors can detect this failure. The information about a failure can be spread in the network, but this produces unnecessary communication overhead, if there is no demand for routing a message. Therefore we concentrate on a reactive route discovery, which implies that the locations of the faulty nodes are not known in advance: The routing algorithm has no global knowledge and must solve the problem *online*, i.e. it can send a message in a promising direction but it does not know whether the target can be reached or the message gets stuck in a labyrinth of faulty nodes. As groups of faulty nodes can obstruct a path leading to the target, we call them *barriers*. If there are barriers, the routing algorithm has to perform some kind of exploration, i.e. some paths are possibly dead ends so that other paths have to be investigated. This exploration can be done sequentially which is time-consuming or in parallel which increases the traffic (i.e. the total number of messages used).

An example for a sequential strategy is the following *barrier traversal* algorithm: (1.) Follow the straight line connecting source and target node. (2.) If a barrier is in the way, then traverse the barrier, remember all points where the straight line is crossed, and resume step 1 at that crossing point that is nearest to the target. This algorithm needs $\mathcal{O}(h + p)$ steps, where h is the length of the shortest barrier-free path and p the sum of the perimeter lengths of all barriers. This bound holds also for the traffic, because this is a *single-path* strategy. One can show, that this traffic bound is optimal. An example for a *multi-path* strategy is *expanding ring search*, which is nothing more than to start flooding with a restricted search depth and repeat flooding while doubling the search depth until the destination is reached. This strategy is asymptotically time-optimal, but it causes a traffic of $\mathcal{O}(h^2)$, regardless of the presence of faulty nodes.

We observe that both approaches are efficient with respect to *either* time or traffic. The problem is to find an appropriate measure to address *both* the time and the traffic efficiency. We introduce the *combined comparative ratio*, which is motivated by the following two observations: The first observation is that the shortest barrier-free path can be lengthened by adding faulty nodes that cause a detour; this affects the time behavior. We use the so-called *competitive ratio* [4] which compares the performance of the algorithm with the optimal offline solution: The *competitive time ratio* is the ratio of routing time and optimal time. The second observation is that additional barriers increase the exploration costs; this affects the traffic efficiency. Here, by a comparison with the best offline strategy the exploration costs would be disregarded. So we use a *comparative ratio* which compares the traffic caused by the algorithm with the online lower bound for traffic of $\mathcal{O}(h + p)$. Finally, the *combined comparative ratio* is the maximum of time and traffic ratio. These ratios are defined in Section 3.

Under this measure the disadvantages of the two strategies described above, namely that time is optimized at the expense of the traffic or vice versa, can be expressed: Both strategies have a linear combined comparative ratio. In this paper we present an algorithm that has a sub-linear combined comparative ratio. The algorithm is described and analyzed in Section 4.

1.1 Related Work

The routing problem in computer networks, even if restricted to two-dimensional meshes, can be investigated under various aspects, like fault-tolerance, reliability of the message delivery, message size, complexity of a pre-routing stage etc. In this paper we focus on fault-tolerance, routing as an online problem, competitive analysis, and traffic efficiency with respect to the difficulty of the scenario. These aspects are not only regarded in the field of networking.

A similar model is used by Zakrevski and Karpovski [19]. They also investigate the routing problem for two-dimensional meshes under the store-and-forward model and present a routing algorithm that is based on constructing fault-free rectangular clusters in an offline pre-routing stage. Connections between these clusters are stored in a graph which is used in the routing stage.

Wu [17] presents algorithms for two-dimensional meshes which use only local information and need no pre-routing stage. The faulty regions in the mesh are assumed to be rectangular blocks. In [18] Wu and Jiang present a distributed algorithm that constructs convex polygons from arbitrary fault regions by excluding nodes from the routing process. This is advantageous in the wormhole routing model, because it helps to reduce the number of virtual channels. We will not deal with virtual channels and deadlock-freedom as we consider the store-and-forward model. Faulty mesh networks have been studied in the field of parallel computing, e.g. by Cole et al. [6]. Here, the ability of a network to tolerate faults and emulate the original network is studied. The general goal is the construction of a routing scheme rather than performing an online path selection.

The problem of finding a target in an unknown environment has been investigated in position-based routing as well as in online robot motion planning.

Position-based routing is a reactive routing used in wireless networks, where the nodes are equipped with a positioning system, such that a message can be forwarded in the direction of the target (see [12] for a survey). Due to the limited range of the radio transceivers, there are local minima and messages have to be routed around void regions (an analog to the fault regions in the mesh network). There are various single-path strategies, e.g. [8,5,9]. Position-based strategies have been mainly analyzed in a worst case setting, i.e. the void regions have been constructed such that the connections form a labyrinth. In this case the benefit of a single-path strategy, namely the traffic efficiency compared to flooding, ceases. The algorithm presented in this paper is a compromise of single-path routing and flooding. By analyzing the time in a competitive manner and by including the perimeters of fault regions we can express performance beyond the worst case point of view. This paper improves the results of the authors presented in [15] where a combined comparative ratio of $\mathcal{O}(h^{1/2})$ has been shown.

In online robot motion planning, the task is to guide a robot from a start point to a target in a scenario with unknown obstacles. This is analogous to the position-based routing problem, except for the possibility to duplicate messages in networks. The motion planning problem for an unknown environment is also known as “online searching” or “online navigation” (see [2] for a survey). It has

been addressed by Lumelsky and Stepanov [11] that the performance of navigation strategies depends on the obstacles in the scenario. The proposed strategies are also suitable for traversing mazes. In such cases, and also in some position-based routing strategies, the well known *right-hand rule* is used: By keeping the right hand always in touch of the wall, one will find the way out of the maze. See [10,14] for an overview of maze traversal algorithms. Analog to the network model with rectangular fault blocks described in [17], there are robot navigation strategies for obstacles of rectangular or polygonal shape. A competitive analysis of such algorithms is presented by Papadimitriou and Yannakakis [13] and Blum, Raghavan and Schieber [3], where the ratio of the length of the path chosen by the algorithm and the shortest barrier-free path is considered as performance measure. This complies with the most common definition of the competitive ratio. Lumelsky and Stepanov [11] as well as Angluin, Westbrook and Zhu [1] use a modified competitive measure that uses the sum of the perimeters of the obstacles in the scene instead of the optimal path length as a benchmark. In this paper we use the competitive ratio for the length of the routing path — which is, here, regarded as routing time. For the induced traffic, we use a comparative measure that credits the cost of exploring new obstacles (which in fact every online algorithm has to pay) to the algorithm (cf. Section 3).

The problem studied in this paper has a strong relation to online search and navigation problems. However, it is not clear how unbounded parallelism can be modelled for robot navigation problems in a reasonable way — usually navigation strategies are only considered for a constant number of robots. Therefore, we consider a mesh network with faulty parts as underlying model, which enables us to study the impact of parallelism on the time needed for reaching the destination.

2 Barriers, Borders and Traversal

In this paper we consider a two-dimensional mesh network with faulty nodes. The network is defined by a set of nodes $V \subseteq \mathbb{N} \times \mathbb{N}$ and a set of edges $E := \{(v, w) : v, w \in V \wedge |v_x - w_x| + |v_y - w_y| = 1\}$. A node v is identified by its position $(v_x, v_y) \in \mathbb{N} \times \mathbb{N}$ in the mesh. There is no restriction on the size of the network, because we analyze time and traffic with respect to the position of the given start and target node in the network. We will see that the major impact on the efficiency of the routing algorithm is not given by the size of the network.

We assume a synchronized communication: Each message transmission to a neighboring node takes one *time step*. For multi-hop communication we assume the messages to be transported in a store-and-forward fashion. We also assume that the nodes do not fail while a message is being transported — otherwise a node could take over a message and then break down. However, there is no global knowledge about faulty nodes. Only adjacent nodes can determine whether a node is faulty.

Important Terms and Definitions: The network contains *active* (functioning) and *faulty* nodes. Faulty nodes which are orthogonally or diagonally neigh-

boring form a *barrier*. A barrier consists only of faulty nodes and is not connected to or overlapping with other barriers. Active nodes adjacent to faulty nodes are called *border nodes*. All the nodes in the neighborhood (orthogonally or diagonally) of a barrier B form the *perimeter* of B . A path around a barrier in (counter-)clockwise order is called a *right-hand (left-hand) traversal path*, if every border node is visited and only nodes in the perimeter of B are used. The *perimeter size* $p(B)$ of a barrier B is the number of directed edges of the traversal path. The *total perimeter size* is $p := \sum_{i \in \mathbb{N}} p(B_i)$. The perimeter size is the number of steps required to send a message from a border node around the barrier and back to the origin, whereby each border node of the barrier is visited. It reflects the time consumption of finding a detour around the barrier.

3 Competitive and Comparative Ratios

When designing online algorithms one typically asks for the best solution an algorithm can provide online or even offline. The comparison of the performance of an (online) algorithm with the performance of an optimal offline algorithm is called *competitive analysis* (see [4]). In faulty mesh networks the offline algorithm has global knowledge and can deliver the message on the shortest barrier-free path (we denote the length of this path with h). Therefore, both the offline traffic and the offline time bound is h . Comparing the traffic of an online algorithm with this lower bound yields a competitive ratio that is not very informative because it disregards the cost of exploration. In fact every online algorithm produces traffic of $\Omega(h + p)$ in some worst case situation: Consider a scenario where the faulty nodes form long corridors (containing active nodes). The source node sees only the entrances of these corridors. Yet, only one corridor leads to the target; the others are dead ends. Every online routing strategy has to examine the corridors (i.e. explore all the barriers), because in the worst case the exit is at the corridor which is examined as the last. This consideration leads to a lower traffic bound of $\Omega(h + p)$, regardless whether the exploration is done sequentially or in parallel.

The optimal time behavior depends on the type of strategy: A single-path strategy has to traverse the barriers sequentially. This leads to a lower time bound of $\Omega(h + p)$ for all single-path strategies. A multi-path strategy like expanding ring search (repeated flooding with increasing search depth) can do this exploration in parallel. The trivial lower bound of $\Omega(h)$ can be achieved using such a strategy.

	Time	Traffic
Online lower bound, single-path	$\Omega(h + p)$	$\Omega(h + p)$
Online lower bound, multi-path	$\Omega(h)$	$\Omega(h + p)$
Best offline solution	h	h

For the time analysis, we use the performance of the best offline algorithm as a benchmark. Regarding time, the best offline performance is of the same order as the online lower bound.

Definition 1. An algorithm A has a competitive ratio of c , if $\forall x \in \mathcal{I} : C_A(x) \leq c \cdot C_{\text{opt}}(x)$, where \mathcal{I} is the set of all instances of the problem, $C_A(x)$ the cost of algorithm A on input x and $C_{\text{opt}}(x)$ the cost of an optimal offline algorithm on the same input.

Definition 2. Let h be the length of the shortest barrier-free path between source and target. A routing algorithm has competitive time ratio $\mathcal{R}_t := T/h$ if the message delivery is performed in T steps.

Regarding traffic, a comparison with the best offline behavior would be unfair, because no online algorithm can reach this bound. So, we define a *comparative ratio* based on a *class* of instances of the problem, which is a modification of the comparative ratio introduced by Koutsoupias and Papadimitriou [7]: In this paper, we compare the algorithm A w.r.t. the competing algorithm B using the cost of each algorithm in the worst case of a class of instances instead of comparing both algorithms w.r.t. a particular instance that causes the worst case ratio. The reason is the following: On the one hand for every online algorithm A there is a placement of barriers with perimeter size p such that A is forced to pay extra cost $\mathcal{O}(p)$ (cf. the labyrinth scenario described above). On the other hand in the class of online algorithms there is always one algorithm B that uses the path which is the shortest path in this particular scenario. Therefore B has extra knowledge of the scenario.

Definition 3. An algorithm A has a comparative ratio $f(P)$, if

$$\forall p_1 \dots p_n \in P : \max_{x \in \mathcal{I}_P} C_A(x) \leq f(P) \cdot \min_{B \in \mathcal{B}} \max_{x \in \mathcal{I}_P} C_B(x),$$

where \mathcal{I}_P is the set of instances which can be described by the parameter set P , $C_A(x)$ the cost of algorithm A and $C_B(x)$ the cost of an algorithm B from the class of online algorithms \mathcal{B} .

With this definition we address the difficulty that is caused by a certain class of scenarios that can be described in terms of the two parameters h and p . For any such instance the online traffic bound is $\min_{B \in \mathcal{B}} \max_{x \in \mathcal{I}_{\{h,p\}}} C_B(x) = \Theta(h+p)$. Note, that for any choice of a scenario one can find an optimal offline algorithm: $\max_{x \in \mathcal{I}_{\{h,p\}}} \min_{B \in \mathcal{B}} C_B(x) = h$. This requires the modification of the comparative ratio in [7] in order to obtain a fair measure. So, we use the online lower bound for traffic to define the *comparative traffic ratio*.

Definition 4. Let h be the length of the shortest barrier-free path between source and target and p the total perimeter size. A routing algorithm has comparative traffic ratio $\mathcal{R}_{Tr} := M/(h+p)$ if the algorithm needs altogether M messages.

Under these ratios we can formalize the intuition telling that flooding as a multi-path strategy performs well in mazes and badly in open space, while some single-path strategy performs well in open space, but bad in mazes. The following table shows the competitive time ratio \mathcal{R}_t and the comparative traffic ratio \mathcal{R}_{Tr} of a single-path and a multi-path strategy. We consider the barrier

traversal algorithm described in Section 1 as traffic-optimal single-path strategy, and expanding ring search as time-optimal multi-path strategy.

Multi-path	\mathcal{R}_t	\mathcal{R}_{Tr}	Single-path	\mathcal{R}_t	\mathcal{R}_{Tr}
General	$\frac{\mathcal{O}(h)}{h}$	$\frac{\mathcal{O}(h^2)}{h+p}$	General	$\frac{\mathcal{O}(h+p)}{h}$	$\frac{\mathcal{O}(h+p)}{h+p}$
Open space ($p < h$)	$\mathcal{O}(1)$	$\mathcal{O}(h)$	Open space ($p < h$)	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Maze ($p = h^2$)	$\mathcal{O}(1)$	$\mathcal{O}(1)$	Maze ($p = h^2$)	$\mathcal{O}(h)$	$\mathcal{O}(1)$

This comparison shows that by these basic strategies the time is optimized at the expense of the traffic or vice versa. To address both the time and the traffic efficiency, we define the *combined comparative ratio*:

Definition 5. *The combined comparative ratio is the maximum of the competitive time ratio and the comparative traffic ratio: $\mathcal{R}_c := \max\{\mathcal{R}_t, \mathcal{R}_{Tr}\}$*

For both strategies compared above this ratio is linear, i.e. $\mathcal{R}_c = \mathcal{O}(h)$.

4 The Algorithm

The basic idea of the routing algorithm is to use flooding only if necessary. Therefore we identify regions where flooding is affordable. This is done by a subdivision (partitioning) of a quadratic search area into smaller squares. Messages are sent in parallel to each square using only paths on the borders of the squares. If a message on such a path encounters a barrier, it tries to circumvent the barrier in the interior of a square. However, if too many barrier nodes are detected while traversing the border of the barrier, then the algorithm floods the whole square.

4.1 Incremental BFS

We observe that flooding defines a bread-first search (BFS) tree by the message paths. The BFS tree contains all the active nodes that are reachable from the root node. Obviously, both the size and the depth of the BFS tree are bounded by the number of nodes in the network. We use an incremental BFS which works in $\log d(T)$ iterations, where $d(T)$ is the depth of the BFS tree T : In the i -th iteration BFS is started with a search depth restricted to 2^i .

In every iteration i we observe *open paths* and *closed paths* in the tree. Open paths are paths with length larger than 2^i where the exploration is stopped after 2^i steps because of the search depth restriction. Closed paths end in a leaf of the tree. In the next iteration (assumed that the target is not found in this iteration) we do not need to investigate nodes on closed paths again. Thus, we have to remember and revisit only open paths for continuing the search. The following lemma shows that the steps needed for this BFS strategy are linear in the size of the tree.

Lemma 1. *Given a tree T , let T_i denote a sub-graph of T which contains all paths from the root to a leaf with length greater than 2^i , which are pruned at depth 2^{i-1} . Then for all trees T with depth $d(T)$ it holds $\sum_{i=0}^{\log d(T)} s(T_i) \leq 3s(T)$, where $s(T)$ denotes the size of the tree, i.e. the number of nodes.*

A proof is given in [16].

4.2 The Online Frame Multicast Problem

In the following we define a multicast problem for a quadratic mesh network. The solution of this problem leads to the solution of the routing problem.

Definition 6. *The frame of a $g \times g$ mesh is the set of framing nodes $F = \{v \in V_{g \times g} : v_x \in \{1, g\} \vee v_y \in \{1, g\}\}$. For a $g \times g$ mesh and a set of entry nodes $s_1, \dots, s_k \in F$, the frame multicast problem is to multicast a message (starting from the entry nodes) to all nodes u on the frame which are reachable, i.e. $u \in F \wedge \exists s_i : \text{dist}(s_i, u) \leq g^2$ where $\text{dist}(s_i, u)$ denotes the length of the shortest barrier-free path from s_i to u .*

Definition 7. *Given a $g \times g$ mesh with entry nodes s_1, \dots, s_k which are triggered at certain time points t_1, \dots, t_k . A routing scheme is called c -time-competitive if for each active node u on the frame of the mesh a message is delivered to u in at most $\min_{i \in [k]} \{t_i + c \cdot \text{dist}(s_i, u)\}$ steps where $\text{dist}(s_i, u)$ denotes the length of the shortest barrier-free path from s_i to u .*

Obviously, a simple flooding algorithm is 1-time-competitive. But it needs $\mathcal{O}(g^2)$ messages, regardless whether few or many barrier nodes are involved. With the following algorithm we can reduce the traffic with only constant factor slow down.

The Traverse and Search Algorithm: The basic idea of the Traverse and Search algorithm for the online frame multicast problem is to traverse the frame and start a breadth-first search if a barrier obstructs the traversal. The algorithm works as follows:

The message delivery is started at each of the entry nodes as soon as they are triggered. An entry node s sends two messages in both directions along the frame, i.e. a message is forwarded to both neighboring frame nodes (if present) that is in turn forwarded to other frame nodes (*frame traversal*). If the message is stopped because of a barrier (i.e. a frame node is faulty) then a flooding process is started in order to circumvent the barrier and return to the frame. We call the nodes that coordinate the flooding process *exploration nodes* (cf. Figure 2).

For the flooding process we use the idea of the incremental BFS described in Section 4.1, i.e. flooding is started in consecutive rounds until the newly found barriers are small or until the flooded area cannot be extended anymore. In the i -th round the search depth is restricted to 2^{i+1} . This is sufficient to circumvent a single faulty frame node in the first round ($i = 1$). If in round i the number of border nodes b_i is greater than a constant fraction of the search depth $\alpha 2^i$ with $0 < \alpha < 1$ then flooding is re-started in the next round with doubled search depth. Therefore it is necessary to report the number of flooded border nodes (i.e. active nodes adjacent to barrier nodes) to the entry node. These reply messages are sent to the entry node using the reverse of the flooding paths, which are defined by the BFS tree. At the branching points of the BFS tree the replies from all branches are collected and merged. This collection process behaves like the BFS in reverse time order. So, the traffic for collecting information corresponds to the forward search of BFS and is increased only by a factor of 2.

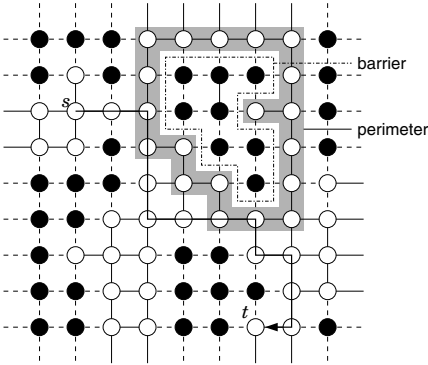


Fig. 1. Optimal routing path from s to t in a mesh network with faulty nodes (black)

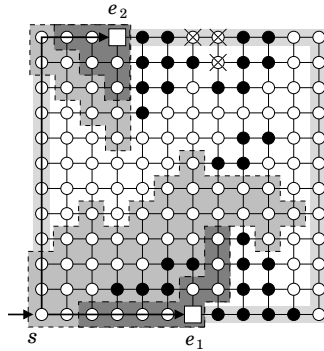


Fig. 2. Execution of the Traverse and Search Algorithm. Entry node s , exploration nodes e_1, e_2 , explored nodes: light background, occupied nodes: dark background.

The open paths (see Section 4.1) in the BFS tree are used for flooding a greater area in the next round. If only small barriers are discovered then the flooding process is aborted. Then the flooded frame nodes at which the search depth is reached continue the frame traversal. In some situations the flooding process is started concurrently from different locations. So we have to ensure that the same regions are not flooded too often by different entry nodes. Otherwise the traffic would disproportionately increase. But if we refrain from flooding a region twice then the flooded regions of one entry node can block the flooding process of another entry node, which would affect the time behavior. Our solution to this problem is as follows: When a node is flooded for the first time, it is marked as *explored*. When it is flooded for the second time (because it is part of an open path in the BFS tree) it is marked as *occupied*. Explored nodes may be explored again by other exploration nodes, but it is forbidden to flood any occupied node again. These two marks are only valid for a specific round of the BFS, that corresponds to a specific search depth. This way we only forbid to flood a region twice with the same search depth.

Lemma 2. *The Traverse and Search algorithm is $\mathcal{O}(1)$ -time-competitive.*

Lemma 3. *Given a $g \times g$ mesh with total perimeter size p . The Traverse and Search algorithm produces traffic $\mathcal{O}(g + \min\{g^2 \log g, p^2 \log g\})$.*

Proof sketch: The time and traffic analysis is based on the following ideas: If we cannot approximate the shortest path by simply traversing the frame then we use the incremental BFS which has a linear asymptotic behavior. Thus, we achieve a constant competitive time ratio. The bound for the number of messages is derived as follows: We allow to flood an area that is quadratic in the number of discovered border nodes. But this area is also bounded by the size of the $g \times g$

mesh. The search from concurrent entry points costs a logarithmic factor. The traversal of the frame costs additional g messages. Complete proofs for Lemma 2 and Lemma 3 are given in [16].

4.3 Grid Subdivision

In order to reduce traffic asymptotically while keeping a constant competitive time behavior, we subdivide an area into smaller squares and apply the Traverse and Search algorithm in each square of this grid subdivision. This technique can be used instead of flooding, which is used by the incremental BFS.

Lemma 4. *Given a $g_1 \times g_1$ mesh with total perimeter size p . For all g_0 with $1 \leq g_0 \leq g_1$ there is a $\mathcal{O}(1)$ -time-competitive algorithm for the online frame multicast problem with traffic $\mathcal{O}(\frac{g_1^2}{g_0} + p \cdot g_0 \log g_0)$.*

A proof is given in [16].

This leads to a modified Traverse and Search algorithm, which is similar to the algorithm described in Section 4.2 except for the BFS: The multicast process is started with the level-1 square and begins with the frame traversal. If a barrier prevents a frame node from proceeding with the frame traversal, the frame node becomes exploration node and starts the modified BFS, which we call level-1 BFS. The flooding process now uses only the nodes on the grid, i.e. only the frame nodes of level-0 squares are “flooded”, whereas the interior of the frames is controlled by the Traverse and Search algorithm on level 0.

The level-1 BFS tree which is defined by the flooding process consists only of level-0 frame nodes. Especially the leaves of the tree are frame nodes, namely the entry nodes of level-0 squares, because at such points the control is given to the Traverse and Search algorithm on level 0. The BFS-tree will be used for gathering information (newly found border nodes and open paths) for the coordinating level-1 entry nodes.

4.4 Recursive Subdivision

A further reduction of the traffic can be achieved, if we use the subdivision technique to replace the flooding phase of the Traverse and Search Algorithm and apply this technique recursively.

Let ℓ be the number of recursive subdivisions, g_ℓ the edge length of a top level square. Beginning with the top-level squares, each $g_i \times g_i$ square is subdivided into squares with edge length g_{i-1} ($i = 1, \dots, \ell$). On the lowest level the Traverse and Search algorithm is applied, which produces a traffic of $g_0 + \min\{g_0^2 \log g_0, p^2 \log g_0\}$, where p is the perimeter size in the $g_0 \times g_0$ square. Traffic in the $g_i \times g_i$ square is caused by the trials of surrounding the square and — if the number of thereby observed border nodes is too large — by covering the corresponding area with $g_{i-1} \times g_{i-1}$ sub-squares.

We use the following notation: \mathbf{tr}_ℓ denotes the traffic induced by a single $g_\ell \times g_\ell$ square; $\mathbf{Tr}_\ell(a)$ denotes the traffic of an area a which is dissected by level- ℓ squares. Define $G(\ell) := \sum_{i=1}^{\ell} \frac{g_i}{g_{i-1}} \prod_{j=i}^{\ell} \log g_j + g_0 \prod_{i=0}^{\ell} \log g_i$.

Theorem 1. *Given a $g \times g$ mesh with total perimeter size p . For all g_i with $1 \leq g_0 \leq \dots \leq g_\ell$ there is a $\mathcal{O}(c^\ell)$ -time-competitive algorithm for the online frame multicast problem with traffic $\mathcal{O}(\frac{g^2}{g_\ell} + p \cdot G(\ell))$.*

A proof is given in [16]. Note, that the recursive subdivision of an area a with ℓ levels produces traffic $\mathbf{Tr}_\ell(a) = \mathcal{O}(\frac{a}{g_\ell} + p \cdot G(\ell))$.

4.5 Expanding Grid

The solution of the frame multicast problem enables us to solve the routing problem: For routing a message from s to t , we first define four connected quadratic subnetworks in the environment of s and t which are connected (see Figure 3); then we apply the modified Traverse and Search algorithm to the quadratic subnetworks to route a message. Again we use the idea of expanding ring search: We begin with a small, restricted region which is enlarged if the target cannot be reached because of a barrier (see Figure 4).

Theorem 2. *Given a mesh network of arbitrary size, a source node s and a target node t . There is a $\mathcal{O}(c^\ell)$ -time-competitive routing algorithm with traffic $\mathcal{O}(h^2/g_\ell + p \cdot G(\ell))$, where h is the shortest possible hop distance between s and t , and p the total perimeter size; g_ℓ and $G(\ell)$ are defined as in Section 4.4.*

A proof is given in [16].

Theorem 3. *For $g_\ell = \Theta(h)$ and $g_i = h^{\frac{(i+1)}{(\ell+1)}}$, $i = 0, \dots, \ell-1$ the algorithm of Theorem 2 needs time $h \cdot c \sqrt{\frac{\log h}{\log \log h}}$ and causes traffic $\mathcal{O}\left(h + p \sqrt{\frac{\log h}{\log \log h}} h \sqrt{\frac{4 \log \log h}{\log h}}\right)$.*

Proof. We set $g_i := h^{(i+1)/(\ell+1)}$ so that $g_i/g_{i-1} = h^{1/(\ell+1)}$:

$$G(\ell) = \sum_{i=1}^{\ell} h^{\frac{1}{i+1}} \prod_{j=i}^{\ell} \log h^{\frac{j+1}{i+1}} + h^{\frac{1}{\ell+1}} \prod_{i=0}^{\ell} \log h^{\frac{i+1}{\ell+1}} \leq (\ell + 1) h^{\frac{1}{\ell+1}} \log^{\ell+1} h.$$

In order to minimize this term we set $h^{\frac{1}{\ell+1}} = \log^{\ell+1} h$. By using $\sqrt{\frac{\log h}{\log \log h}} = \ell + 1$ we obtain the traffic of the routing algorithm. \square

Corollary 1. *The routing algorithm of Theorem 3 has competitive time ratio of $c \sqrt{\frac{\log h}{\log \log h}}$ and comparative traffic ratio of $\mathcal{O}\left(\sqrt{\frac{\log h}{\log \log h}} h \sqrt{\frac{4 \log \log h}{\log h}}\right)$, yielding a sub-linear combined comparative ratio of $h^{\mathcal{O}\left(\sqrt{\frac{\log \log h}{\log h}}\right)}$, which is in $o(h^\epsilon)$ for all $\epsilon > 0$.*

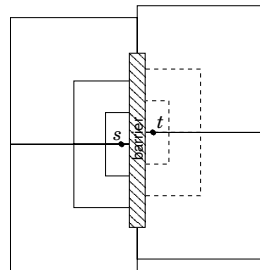
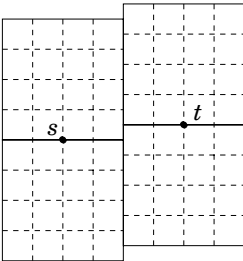


Fig. 3. Search area with grid subdivision

Fig. 4. Enlargement of the search area

References

1. Dana Angluin, Jeffery Westbrook, and Wenhong Zhu. Robot navigation with range queries. In *Proc. of the 28th Annual ACM Symposium on Theory of Computing*, pages 469–478, 1996.
2. Piotr Berman. On-line searching and navigation. In Amos Fiat and Gerhard J. Woeginger, editors, *Online Algorithms: The State of the Art*, volume 1442 of *Lecture Notes in Computer Science*, pages 232–241. Springer, 1998.
3. Avrim Blum, Prabhakar Raghavan, and Baruch Schieber. Navigating in unfamiliar geometric terrain. *SIAM Journal on Computing*, 26:110–137, 1997.
4. Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.
5. P. Bose, P. Morin, I. Stojmenovic, and J. Urrutia. Routing with guaranteed delivery in ad hoc wireless networks. *Wireless Networks*, 7(6):609–616, 2001.
6. R. J. Cole, B. M. Maggs, and R. K. Sitaraman. Reconfiguring Arrays with Faults Part I: Worst-case Faults. *SIAM Journal on Computing*, 26(16):1581–1611, 1997.
7. Elias Koutsoupias and Christos H. Papadimitriou. Beyond competitive analysis. *SIAM Journal on Computing*, 30(1):300–317, 2000.
8. E. Kranakis, H. Singh, and J. Urrutia. Compass routing on geometric networks. In *Proc. 11th Canadian Conference on Computational Geometry*, pages 51–54, 1999.
9. F. Kuhn, R. Wattenhofer, and A. Zollinger. Asymptotically optimal geometric mobile ad-hoc routing. In *Proc. of the 6th int. Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, pages 24–33, 2002.
10. Vladimir J. Lumelsky. Algorithmic and complexity issues of robot motion in an uncertain environment. *Journal of Complexity*, 3(2):146–182, 1987.
11. Vladimir J. Lumelsky and Alexander A. Stepanov. Path-planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape. *Algorithmica*, 2:403–430, 1987.
12. M. Mauve, J. Widmer, and H. Hartenstein. A survey on position-based routing in mobile ad hoc networks. *IEEE Network Magazine*, 15(6):30–39, November 2001.
13. Christos H. Papadimitriou and Mihalis Yannakakis. Shortest paths without a map. In *Proc. of the 16th Int. Colloq. on Automata, Languages, and Programming (ICALP’89)*, pages 610–620. Elsevier Science Publishers Ltd., 1989.
14. N. Rao, S. Karetí, W. Shi, and S. Iyengar. Robot navigation in unknown terrains: Introductory survey of non-heuristic algorithms. Technical report, Oak Ridge National Laboratory, 1993. ORNL/TM-12410.
15. Stefan Rührup and Christian Schindelhauer. Competitive time and traffic analysis of position-based routing using a cell structure. In *Proc. of the 5th IEEE International Workshop on Algorithms for Wireless, Mobile, Ad Hoc and Sensor Networks (IPDPS/WMAN’05)*, page 248, 2005.
16. Stefan Rührup and Christian Schindelhauer. Online routing in faulty meshes with sub-linear comparative time and traffic ratio. Technical report, University of Paderborn, 2005. tr-rsfb-05-076.
17. Jie Wu. Fault-tolerant adaptive and minimal routing in mesh-connected multicomputers using extended safety levels. *IEEE Transactions on Parallel and Distributed Systems*, 11:149–159, February 2000.
18. Jie Wu and Zhen Jiang. Extended minimal routing in 2-d meshes with faulty blocks. In *Proc. of the 1st Intl. Workshop on Assurance in Distributed Systems and Applications (in conjunction with ICDCS 2002)*, pages 49–55, 2002.
19. Lev Zakrevski and Mark Karpovsky. Fault-tolerant message routing for multiprocessors. In *Parallel and Distributed Processing*, pages 714–730. Springer, 1998.

Heuristic Improvements for Computing Maximum Multicommodity Flow and Minimum Multicut

Garima Batra, Naveen Garg*, and Garima Gupta

Indian Institute of Technology Delhi, New Delhi, India

Abstract. We propose heuristics to reduce the number of shortest path computations required to compute a $1+\epsilon$ approximation to the maximum multicommodity flow in a graph. Through a series of improvements we are able to reduce the number of shortest path computations significantly. One key idea is to use the value of the best multicut encountered in the course of the algorithm. For almost all instances this multicut is significantly better than that computed by rounding the linear program.

1 Introduction

We consider the problem of computing the maximum multicommodity flow in a given undirected capacitated network with k source-sink pairs. The algorithm with the best asymptotic running time for this problem is due to Fleischer [1] and computes a $1 + \omega$ approximation to the maximum multicommodity flow in time $O((m/\omega)^2 \log m)$. A closely related quantity is the minimum multicut in the graph which is NP-hard to compute and for which there is an $O(\log k)$ -approximation algorithm [3].

In this paper we will be concerned with ways of implementing these algorithms so as to reduce the running time in practice. The algorithm of Fleischer is a combinatorial algorithm and similar such algorithms have been proposed for maximum concurrent flow and min-cost maximum concurrent flow and have been the subject of much experimentation [9],[8] and [4]. However, to the best of our knowledge there has been no attempt at a similar study for computing the maximum multicommodity flow. In theory, the maximum multicommodity flow problem can be solved by casting it as a maximum concurrent flow problem. It is not clear why this would also be a good approach in practice. In fact we believe that such is not the case; one reason for this is that even the choice of length functions, a key aspect of these potential reduction algorithms, are not the same for maximum concurrent flow and maximum multicommodity flow.

The novelty of our approach to reduce the running time for computing maximum multicommodity flow lies in our use of the minimum multicut to speed up the computation. This has the added advantage that we get a multicut which

* Work done as part of the “Approximation Algorithms” partner group of MPI-Informatik, Germany.

is much better than the one which would have been obtained by following the algorithm of [3]. We believe that this idea could also be incorporated for the computation of maximum concurrent flow and would yield, both a faster algorithm for computing the flow and a better approximation of the sparsest cut.

The paper is organised as follows. In Section 2 we provide a brief outline of the combinatorial algorithm for computing the maximum multicommodity flow and the approximation algorithm for the minimum multicut. In Section 3, we propose modifications to these algorithms. In Section 4 we present the results of our experiments and we conclude in Section 5 with a discussion of the results.

2 Overview of Existing Literature

In this section we provide a quick overview of the algorithm proposed by [2] and [1] for computing the maximum multicommodity flow and the algorithm of [3] for computing a small multicut.

We are given an undirected graph $G = (V, E)$, a capacity function $c : E \rightarrow \mathbf{R}^+$ and k source-sink pairs $(s_i, t_i), 1 \leq i \leq k$ and we wish to find k flow functions $f_i : E \rightarrow \mathbf{R}^+$ such that the flow of each commodity is conserved at every node other than its source and sink and the total flow through an edge e , $\sum_{i=1}^k f_i(e)$ is at most the capacity of e . The objective is to maximize the total flow routed. The dual of this problem is the minimum multicut problem in which we wish to find a collection of edges of minimum total capacity whose removal disconnects each $s_i - t_i$ pair. It is easy to see that the maximum multicommodity flow is at most the minimum multicut. It is also known that the minimum multicut is no more than $4 \ln(k + 1)$ times the maximum flow and that there are instances where it is as large as $c \log k$ times the maximum flow.

To compute the maximum multicommodity flow we associate a length function $l : E \rightarrow \mathbf{R}^+$ where $l(e) = e^{\epsilon f(e)/c(e)}$ where $f(e)$ is the flow through e , $c(e)$ is the capacity of e and ϵ is a constant which determines how close the flow computed is to the optimum. Define an $s - t$ path as a path between a source and the corresponding sink. The algorithm proceeds in a sequence of iterations. In each iteration, we find the shortest $s - t$ path, say P , and route flow equal to the minimum capacity edge on P , along this path. The edge lengths are updated and the process repeated until the length of the shortest $s - t$ path exceeds $n^{1/\epsilon}$. Fleischer's modification to the above algorithm was to consider the commodities in a round robin manner and route commodity j until the shortest $s_j - t_j$ path has length more than $\alpha(1 + \epsilon)$, where α is the length of the shortest $s - t$ path. With this modification the theoretical bound on the running time reduces by a factor k . This is because, while earlier we were doing k shortest path computations to route flow along one path, with Fleischer's modification we need roughly one shortest path computation for every routing of flow. Since the flow obtained in this manner is not feasible, we scale the flow through each edge by the maximum congestion. To prove that this flow is close to the maximum, we compare it to a feasible solution to the dual linear program. The assignment of lengths to the edges gives a solution to the dual linear program and its value

equals D/α where $D = \sum_{e \in E} l(e)c(e)$ and α is the length of the shortest $s - t$ path.

The computation of the multicut starts with an optimum (or a near optimum) solution to the dual linear program and does a rounding of the edge lengths using a region-growing procedure first proposed by Leighton and Rao [7]. The region growing procedure is essentially a shortest path computation from a source vertex. At each step of this computation we compare the total capacity of the edges with endpoint in the region to the total volume ($l(e)c(e)$) of edges inside the region and stop growing the region when the capacity of the cut is no more than $\ln(k + 1)$ times the volume of the region. We continue by picking a source vertex which is still connected to the corresponding sink and growing a region around it. The multicut so obtained has capacity at most $4 \ln(k + 1)$ times the value of the dual solution.

3 Heuristics for Improving Running Time

In this section we present some heuristics which help reduce the total number of shortest path computations. Since most of the time required for computing a maximum multicommodity flow is spent in finding shortest paths, this reduces the overall running time.

We first observe that in the Garg-Könemann procedure the initial edge lengths are uniform. It is also possible to assign each edge an initial length which is inversely proportional to its capacity. In fact this is the length assignment for the maximum concurrent flow problem. It is surprising that the initial length assignment has a significant effect on the running time in practice. Our experiments in Section 4 demonstrate that it is better to choose uniform initial lengths.

3.1 Modification to Fleischer's Work: Algorithm A

After finding the shortest path for an $s_i - t_i$ pair we route enough flow along the path so that its length exceeds $\alpha(1 + \epsilon)$. This is similar to the idea of Fleischer, except that we now adopt it for each path. Note that the amount of flow routed along a path can be much more than the minimum capacity edge on the path. After routing flow along this path we find the next shortest $s_i - t_i$ path and only if its length exceeds $\alpha(1 + \epsilon)$ do we proceed to the next commodity. The algorithm terminates when the feasible flow found is at least $1/(1 + \omega)$ times the value of the dual solution which is the smallest value of D/α seen in any iteration. This is clearly better than running the algorithm for a predetermined number of iterations which might be much larger than the number of iterations needed for getting the primal and dual solutions within $(1 + \omega)$ of each other. The algorithm obtained with these modifications will be referred to as **Algorithm A**.

3.2 Modification to Garg-Könemann's Work: Algorithm B

In the analysis of the approximation guarantee of the previous algorithm we only use the fact that the length of the path along which flow is routed is at most

$ne^{\epsilon F/\beta}$ where F is the total flow routed and β is the best dual solution obtained so far. This fact can be used to route flow along any path whose length is less than $ne^{\epsilon F/\beta}$. In particular, we find the shortest path and continue routing flow along it as long as its length is less than $ne^{\epsilon F/\beta}$. Note that as we route flow along the path both F and β change; this fact is incorporated in the computation of the amount of flow that can be routed along this path. After exhausting the path we recompute the shortest path between the same source-sink pair and continue. We refer to this modification to Algorithm A as **Algorithm B**.

3.3 Better β Updation: Algorithm C

In the case of a single commodity the maximum flow equals the minimum $s - t$ cut which is the same as the optimum solution to the dual linear program. In the course of shortest path computations, using Dijkstra's algorithm, we encounter certain $s - t$ cuts. By keeping track of the minimum capacity of such cuts we obtain an alternate bound on β . Our next modification to Algorithm B is to choose the smaller of the two upper bounds as the value of β in the expression $ne^{\epsilon F/\beta}$. We refer to this modified algorithm as **Algorithm C**. This algorithm also reports the minimum $s - t$ cut it encountered. Our results in Section 4 demonstrate that this cut is quite close to the flow computed and that this method of updating β outperforms, by huge margins, our earlier method of updating β .

3.4 Aggressive Routing: Algorithm D

Instead of routing flow along a path till its length exceeds $ne^{\epsilon F/\beta}$, as we do in Algorithms B and C, we can route flow along a path till its length exceeds D/β . This modification is also valid for computing the maximum multicommodity flow and the theoretical bound continues to hold as is borne out by the analysis of the approximation guarantee. Algorithm C uses the value of the minimum $s - t$ cut encountered during each Dijkstra computation, to obtain a tighter upper bound on the value of β for the single commodity flow problem. For multiple commodities, the value of the capacity of a multicut is an upper bound on β . A crude approximation to the multicut is the union of all $s_i - t_i$ cuts. Thus by keeping track of the best $s_i - t_i$ cut encountered and taking the sum of the capacity of these cuts we get a method of updating β . **Algorithm D** for multiple commodities incorporates this method for updating β as well as the best D/α value encountered in an iteration, enabling us to route flow more aggressively to achieve a reduction in the number of Dijkstra computations.

3.5 Conservative Approach: Algorithm E

The aggressive approach of Algorithm D might lead to a very large amount of flow being routed on a path due to which one of the edges of the path becomes the edge with maximum congestion and this leads to a reduction in the feasible flow routed. This contradicts our intention of sending large amounts of flow at each step so as to decrease the number of iterations required to reach maximum flow. To balance these two aspects we send an amount of flow equal to

$\max(f_1, \min(f_2, f_3))$ where f_1, f_2, f_3 are the amounts of flow which when routed along this path would cause the length of the path to exceed $\alpha(1 + \epsilon)$, the congestion to exceed the maximum congestion and the length of the path to exceed D/β respectively. The justification of this choice is as follows. We do not want the maximum congestion to increase or the length of the path to exceed D/β . However, it is acceptable to let the length of the path increase to $\alpha(1 + \epsilon)$ since we are only routing flow along a path whose length is at most $1 + \epsilon$ times the shortest $s - t$ path. We refer to the above algorithm as **Algorithm E**.

3.6 β Updation Using Set Cover: Algorithm F

For multiple commodities it makes sense to obtain a tighter upper bound on β by computing a good approximation to the minimum multicut. We do this by keeping track of the capacity and the $s_i - t_i$ pairs separated by all cuts encountered in the course of shortest path computations. Note that this number is polynomially bounded since the number of shortest path computations required by the algorithm is polynomial and in each shortest path computation we encounter only n different cuts. We then run the greedy algorithm for set cover on this collection of cuts. Here the sets are the cuts, their cost is the capacity of the cut and the elements they cover are the $s_i - t_i$ pairs separated by the cut. This set-cover computation is done at the end of each round. **Algorithm F** is a modification of Algorithm E for multiple commodities and incorporates this method of updating β in addition to the other two methods that were part of Algorithm E. Algorithm F also reports the best multicut that it finds using this method.

3.7 Using Optimum β : Algorithm G

Our final algorithm, which we refer to as **Algorithm G** is the same as Algorithm F but with the initial value of β set to the optimum value. As a consequence there is no update of β in any iteration. While the optimum value of β is not known in advance and so this algorithm cannot be implemented, our intention of experimenting with this algorithm is to see how the running time would have changed if we had known the optimum dual solution.

4 Experiments and Results

We computed the number of shortest path computations required by Algorithms A, B, C, D, E, F and G to achieve a 1.01 approximation to the maximum (multicommodity) flow for 4 classes of graphs.

1. **GRID**(n, k): These are $n \times n$ grids with edge capacities being random numbers in the range $[0.2, 1.2]$. The k source sink pairs are chosen randomly.
2. **GENRMF**(a, b): The generator for this family of graphs was developed by Goldfarb and Grigoriadis [5]. The graph has b square grids each of size $a \times a$. The edges of the grids have unit capacities. The nodes of one grid are connected to those of the next grid by a random matching and the capacities of these

edges are randomly chosen from the range $[1, 100]$. The source and sink are the lower left corner vertex of the first grid and the upper right corner vertex of the last grid respectively.

3. **NETGEN**(n, m, k): These are the graphs generated by the NETGEN generator [6]. Here n is the number of nodes, m the number of edges and k the number of commodities. The edges were assigned capacities in the range $[1, 100]$.
4. **CLIQUE**(a, b, k): These graphs were designed to get a small value of the minimum multicut. They are obtained by taking b cliques, each containing a nodes. There is a randomly selected edge between every pair of cliques, thereby forming a clique of cliques. The k source-sink pairs are also chosen randomly. Each edge is assigned a capacity which is a random number in the range $[0.2, 1.2]$.

Table 1 shows how the number of shortest path computations change with the choice of initial edge lengths when Algorithm A is run on graphs from the NETGEN family. These results prompted us to do the rest of the experiments with uniform initial edge lengths. Tables 2, 3 and 4 show the number of shortest path

Table 1. The effect of initial length on running time of Algorithm A for graphs from the NETGEN family

(n,m)	Length = 1	Length = 1/c(e)
200,1300	20863	247043
200,1500	24225	145499
200,2000	20317	84203
300,3000	29031	124334
300,4000	8694	32305
500,3000	15636	75085
500,20000	44722	531365
1000,10000	13527	98804
1000,20000	20911	148013

Table 2. Maximum Flow computation on graphs from the GRID family

n	Alg. A	Alg. B	Alg. C	Alg. D	Alg. E	Best s-t cut Flow
10	9283	5391	64	18	104	1.005
15	9591	10159	71	22	519	1.007
20	15620	13465	111	31	809	1.008
25	15904	7736	56	17	313	1.010
30	8874	8929	83	25	346	1.010
35	9630	8808	71	21	295	1.009
40	18022	11273	106	26	585	1.010
45	21077	11120	84	21	608	1.007
50	10522	6900	56	18	383	1.006

Table 3. Maximum Flow computation on graphs from the GENRMF family

n	Alg. A	Alg. B	Alg. C	Alg. D	Alg. E	<u>Best s-t cut</u> <u>Flow</u>	
5,5	58908	73513	1402	267	1233	1.009	
6,5	951711	154111	2432	616	1686	1.010	
7,5	113649	262430	2696	660	1037	1.010	
8,5	168566	333785	2718	1371	2718	1.010	
9,5	205094	487650	4562	921	1974	1.010	
10,5	250071	628899	7502	2254	3880	1.010	
11,5	323942	905905	8030	2493	4529	1.009	
12,5	383308	1110290	11375	2890	5871	1.010	
13,5	531897	1302604	15080	4906	6143	1.010	
14,5	559427	1782664	17276	4378	7064	1.010	
15,5	679675	1868791	16422	12632	7395	1.010	

Table 4. Maximum Flow computation on graphs from the NETGEN family

n	Alg. A	Alg. B	Alg. C	Alg. D	Alg. E	<u>Best s-t cut</u> <u>Flow</u>	
200,1300	20863	19426	246	62	512	1.009	
200,1500	24225	36624	425	91	438	1.007	
200,2000	20317	20506	292	51	370	1.006	
300,3000	17848	37516	479	101	336	1.008	
300,4000	8694	10792	47	9	197	1.008	
500,3000	15636	13105	122	31	341	1.009	
500,20000	44722	125430	1258	162	870	1.010	
1000,10000	13527	15902	193	31	218	1.008	
1000,20000	20911	31148	340	45	688	1.009	

Table 5. Maximum multicommodity flow computation on graphs from the GRID family

n	Alg. A	Alg. D	Alg. E	Alg. F	Alg. G	η	χ
10,5	316910	116248	108122	108122	27157	1.108	1.749
11,5	104731	25823	939	939	1068	1.006	1.087
12,5	372285	3971	111567	4465	3592	1.010	2.415
13,5	451465	283054	207653	219404	78521	1.057	2.782
14,5	581499	407020	581499	349233	122850	1.187	1.464
15,10	651400	477166	424879	429275	210060	1.275	1.881
16,10	1265081	1002172	978359	978359	401593	1.262	2.200
17,10	1204609	865802	765781	768940	221279	1.180	2.806
18,10	1623639	1278925	1226415	1385144	848565	1.043	1.705
19,10	2337653	299574	1705738	285099	285338	1.010	2.999
20,10	880087	579862	282074	282057	187999	1.022	4.495

computations required to compute (single commodity) maximum flow in GRID, GENRMF and NETGEN graphs by the algorithms A, B, C, D and E. Each row of the tables also shows the factor by which the smallest $s - t$ cut encountered

Table 6. Maximum multicommodity flow computation on graphs from the NETGEN family

n	Alg. A	Alg. D	Alg. E	Alg. F	Alg. G	η	χ
100,1000,10	965771	308996	39975	41173	40617	1.010	5.284
200,1300,15	1784704	814611	658623	646371	531710	1.073	5.952
200,1500,15	1230462	443499	340084	337862	3398146	1.043	5.262
200,2000,20	884969	87481	382410	382410	382410	1.054	6.867
300,4000,15	2901211	1461092	983566	971744	789900	1.034	6.270
500,3000,30	3521900	7044	14581	1821097	13867	1.010	1.631
700,30000,50	814082	414352	277310	277310	173382	1.020	3.878

Table 7. Maximum multicommodity flow computation on graphs from the CLIQUE family

n	Alg. A	Alg. D	Alg. E	Alg. F	Alg. G	η	χ
10,2,2	9187	6	4288	6	6	1.000	5.400
20,3,4	28990	280	12799	2199	1768	3.902	2.874
30,3,5	27160	67	13126	206	206	1.009	4.751
30,15,5	81418	53998	50796	39601	5011	1.792	3.084
50,5,5	40736	15676	20880	17193	4736	1.777	1.205
50,10,5	112955	273659	71222	47203	27507	1.487	3.324
100,10,10	877342	713805	520835	459512	362520	1.907	2.031
100,20,10	799180	783664	623439	637390	497888	2.037	3.268
150,15,5	88452	615498	47227	37574	14878	1.877	0.647
200,20,10	855318	2094486	626616	626616	524377	2.121	3.293
250,20,10	426379	1833822	341886	319136	178170	2.232	3.631

during the course of the shortest path computations exceeds the value of the flow computed. This entry was obtained from a run of Algorithm E. Tables 5, 6 and 7 show the number of shortest path computations required to compute maximum multicommodity flow in GRID, NETGEN and CLIQUE graphs by algorithms A, D, E, F and G. Once again, each row of the tables also shows how close the smallest multicut obtained by the greedy algorithm is to the value of the flow computed. In addition, we compare the value of the multicut obtained by the rounding technique in [3] with that of the smallest multicut obtained by the greedy algorithm. In tables 5,6, 7 the second last column, labeled η , is the ratio of the best multicut found by our algorithm to the value of the flow computed. The last column, labeled χ , is the ratio of the multicut found by our algorithm to the multicut computed using the approach in [3].

5 Discussion and Conclusions

For the single commodity max flow computation, the approach of updating β using the minimum $s - t$ cut encountered in the shortest path computations

yields significant improvements in running time as shown by the performance of Algorithms C, D and E. A reduction in the number of Dijkstra computations is observed as we move from Algorithm C to D, routing flow more exhaustively through individual paths. The deterioration in the performance of Algorithm E as compared to Algorithms C and D can be attributed to the fact that we are being a bit more conservative in E since we do not route so much flow on a path as to increase the maximum congestion. One would be tempted to go with the more aggressive approach of Algorithm D but our experiments with multiple commodities illustrate that such an aggressive approach has its pitfalls. When the amount of flow routed on a path is not constrained by the maximum congestion of an edge in the graph we find instances where the number of shortest path computations required is much larger than in Algorithm A. Of course, there are also instances where such an aggressive approach is much much better than Algorithm A. We found it very hard to predict whether this aggressive approach would work well on an instance or not and hence settled for the more conservative routing scheme.

Further, for almost all instances the best $s - t$ cut encountered is close to the value of the maximum flow and in all instances the best dual solution is given by the best $s - t$ cut encountered and not by the best D/α solution.

For multiple commodities, for most instances we see a progressive reduction in the number of shortest path computations as we move from algorithms A to E to F and finally G. In some instances the differences are very pronounced as in `CLIQUE(10,2,2)` or `CLIQUE(20,3,4)`.

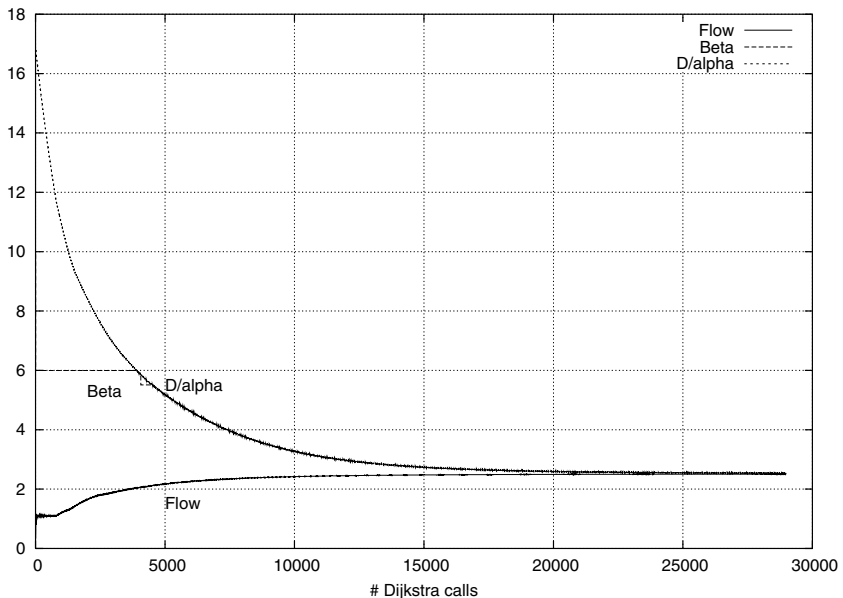


Fig. 1. A plot of D/α and the feasible flow in each iteration for Algorithm A

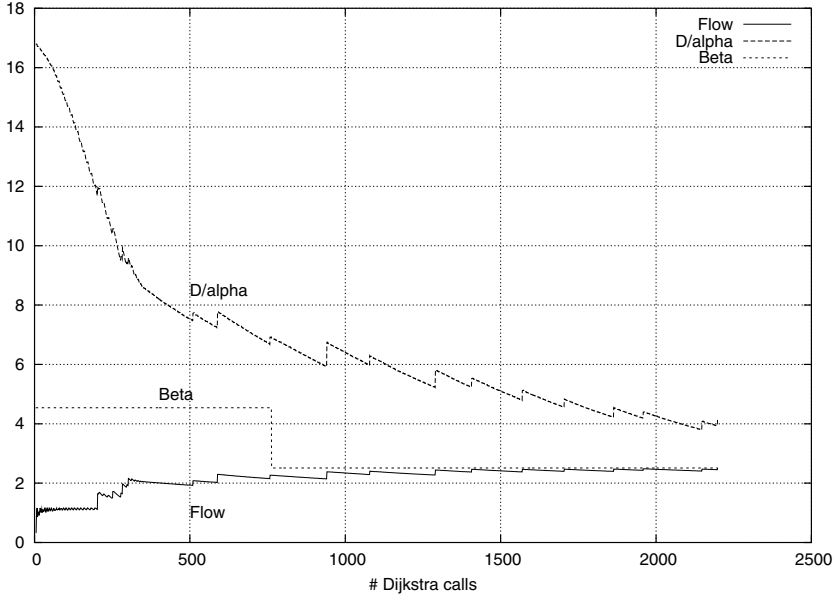


Fig. 2. A plot of D/α and the feasible flow in each iteration for Algorithm F

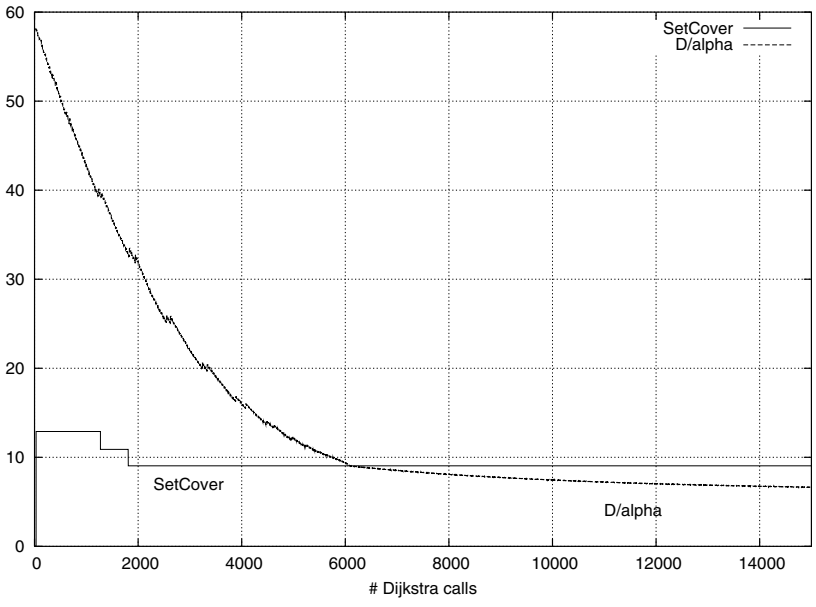


Fig. 3. A plot of D/α and the multicut obtained by greedy set-cover in each iteration of Algorithm F

To understand the behavior of Algorithm E we plotted the values of D/α and the feasible flow in each iteration (Figure 2) for the instance `CLIQUE(20,3,4)`. While in the case of Algorithm A these plots (Figure 1) were smooth reflecting a continuous change in the values of D/α and the feasible flow, for Algorithm E we find the value of D/α increasing in small spikes which, interestingly enough, correspond to step increases in the value of the feasible flow. These iterations correspond to the scenarios where the amount of flow routed is constrained by the D/β bound on the length of the path. As expected a fair bit of flow gets routed in these iterations and our constraint that the maximum congestion of an edge not increase leads to much of this flow reflecting as a feasible flow. We found this behavior in varying degrees in all instances.

Figure 3 shows a plot of D/α and the multicut obtained by running the greedy algorithm for set-cover for the instance `CLIQUE(50,10,5)`. As can be seen, in the initial iterations, D/α is much larger than the multicut found and so the value of β is determined by the multicut. This leads to frequent iterations where the flow is routed along a path till it gets a length of D/β ; this can be seen by the numerous spikes in the plot. In the later iterations the value of D/α becomes less than the best multicut and starts determining β . Now flow is routed only till the path length exceeds $\alpha(1 + \epsilon)$ and this corresponds to the smooth nature of the plot.

In all instances, the multicut computed by running the greedy set-cover algorithm on the cuts encountered is significantly better than the multicut computed by the region growing procedure of [3]. We believe that even theoretically it should be possible to argue that the multicut obtained in this manner is no more than $O(\log k)$ times the maximum multicommodity flow.

References

1. L. Fleischer. Approximating fractional multicommodity flow independent of the number of commodities. *SIAM J. Discrete Math.*, 13:505520, 2000.
2. N. Garg and J. Konemann. Faster and simpler algorithms for multicommodity flow and other fractional packing problems. In *Proceedings, IEEE Symposium on Foundations of Computer Science*, pages 300309, 1998.
3. N. Garg, V.V. Vazirani, and M. Yannakakis. Approximate max-flow min-(multi)cut theorems and their applications. *SIAM J. Comput.*, 25(2):235251, 1996.
4. A.V. Goldberg, J.D. Oldham, S. Plotkin, and C. Stein. An implementation of a combinatorial approximation algorithm for for minimum-cost multicommodity flows. In *Proceedings, MPS Conference on Integer Programming and Combinatorial Optimization*, 1998.
5. D. Goldfarb and M.D. Grigoriadis. A computational comparison of the dinic and network simplex methods for maximum flow. *Annals of Operations Research*, 13:83 123, 1988.
6. D. Klingman, A. Napier, and J. Stutz. *Netgen: A program for generating large scale capacitated assignment, transportation and minimum cost flow network problems*. *Management Science*, 20:814821, 1974.

7. F. Leighton and S. Rao. An approximate max-flow min-cut theorem for uniform multicommodity flow problems with application to approximation algorithms. In Proceedings, IEEE Symposium on Foundations of Computer Science, pages 422431, 1988.
8. T. Leong, P. Shor, and C. Stein. Implementation of a combinatorial multicommodity flow algorithm. In Network flows and matchings, Volume 12 of DIMACS series in Discrete Mathematics and Theoretical Computer Science, pages 387405. American Mathematical Society, 1993.
9. T. Radzik. Experimental study of a solution method for the multicommodity flow problem. In Workshop on Algorithm Engineering and Experiments, pages 79102, 2000.

Relax-and-Cut for Capacitated Network Design

Georg Kliewer and Larissa Timajev*

Department of Computer Science, University of Paderborn, Germany
{Georg.Kliewer, timajev}@upb.de

Abstract. We present an evaluation of a Lagrangean-based branch-and-bound algorithm with additional valid inequalities for the capacitated network design problem. The focus is on two types of valid inequalities, the cover inequalities and local cuts. We show how these inequalities can be considered in a Lagrangean relaxation without destroying the computationally simple structure of the subproblems. We present an extensive computational study on a large set of benchmark data. The results show that the presented algorithm outperforms many other exact and heuristical solvers in terms of running time and solution quality.

1 Introduction

The task of network design occurs in several applications in transportation and telecommunication. In long-term planning the structure of the network is defined. We define the nodes and edges of the network as well as capacities for them. The network is constructed by installing roads, bus lines or communication cables. The purpose for the network usage can be multifarious. The operator may want to transport goods on a road network, passengers in a flight network, or he has to route phone calls through the network. Several properties of the network determine the suitability for the application case, like robustness in the case of failures or extra capacities for fluctuating demand. In mid-term planning steps, other resources are dealt with - e.g. trucks, airplanes, drivers, and crews. The objective whilst generating routes and schedules is cost efficiency. In short-term planning physical entities and persons with concrete attributes (e.g. maintenance, vacation) must be considered. During operation the network is subject to several irregularities or failures. The task is to react to them by preserving cost efficiency.

The network design problem considered in this paper has several input data and degrees of planning. First, the set of nodes and potential edges is given. Second, the transportation demand, which must be routed through the network, is defined. Fixed costs for the installation of edges and variable costs for the transportation of commodities on edges, define the objective function. The network is a capacitated one - the edge capacity may not be exceeded by the routing.

Solution methods which can be used for network design depend heavily on the size of the problem, it's structure and side constraints. Exact methods and

* This work was partly supported by the German Science Foundation (DFG) project "Optimization in networks" under grant MO 285/15-2.

metaheuristics build the two main method classes. We are using branch-and-bound based methods for solving the network design problem. Depending on the setting, the algorithm can be stopped prior to termination, or some variables are fixed, such that we can not guarantee that the solutions found are optimal. In such cases the algorithm turns into a heuristical one.

1.1 Related Work

An overview of the field of network design is given in [BMM97, MW84, Cra00]. The capacitated network design problem as solved in this paper was first defined in [GC94, CFG01]. Several heuristic approaches for this problem formulation are presented in [CGF00], [GCG03], [GCG04], [CGH04]. In [HY00], a Lagrangian-relaxation based branch-and-bound algorithm is presented. In [SKK02a] we have presented Lagrangian cardinality cuts and coupled variable fixing algorithms in a similar algorithmic approach. Recently, several valid inequalities for the network design problem were discussed in [CCG03]. Relax-and-cut algorithms are presented in [Gui98, RG05].

1.2 Contribution

The contribution of this paper is threefold. First, we describe our implementation of a complete branch-and-bound system where the Lagrangian-relaxation is solved by a subgradient search or bundle-method. Second, we develop two valid inequality types for the network design problem. And third, we incorporate the inequalities in the branch-and-bound algorithm, obtaining the relax-and-cut algorithm.

The results obtained in numerical experiments show that our system outperforms other solvers like CPLEX in the exact setting. The heuristic configuration outperforms other heuristic algorithms in terms of solution quality and running time. The presented cuts are valuable and improve the systems performance significantly.

The paper is organized as follows. In the next section we formalize the mathematical model for the network design problem. Then we describe how additional cuts can be considered in Lagrangian relaxation without destroying the structure of the subproblem. Hereafter we investigate cover inequalities and local cuts. In the results section we draw comparisons between other solvers for the network design problem and between several configurations of our system. In the appendix comprehensive result data can be found for reference purposes.

2 The Capacitated Network Design Problem

The capacitated network design problem is defined as a mixed-integer linear problem. An optimal subset of arcs of a network $G = (\mathcal{N}, \mathcal{A})$ must be found such that a given transport demand can be accommodated within in the network. The demand is defined as an origin-destination pair (one commodity). One unit of a commodity $k \in \mathcal{C}$ can be transported via arc (i, j) for a cost c_{ij}^k . The total

demand for commodity k is r^k . The installation costs are defined for each arc (i, j) as f_{ij} which must be paid if any commodity is using this arc. Additionally, there is a capacity u_{ij} on each arc that limits the total amount of flow that can be routed via (i, j) .

For all arcs $(i, j) \in \mathcal{A}$ and commodities $k \in \mathcal{C}$, let $d_{ij}^k = \min\{r^k, u_{ij}\}$ be the maximal amount which can be routed. Define b_i^k as

$$b_i^k = \begin{cases} r^k, & \text{if } i = \text{origin of } k \\ -r^k, & \text{if } i = \text{destination of } k \\ 0 & \text{else} \end{cases}$$

Using variables x_{ij}^k for the flows and y_{ij} for the design decisions, the mixed-integer linear optimization problem for the capacitated network design is defined as follows:

$$\begin{aligned} z_{CNDP} = \min \quad & \sum_{ij} \sum_k c_{ij}^k x_{ij}^k + \sum_{ij} f_{ij} y_{ij} \\ \text{s.t.} \quad & \sum_{j:(i,j) \in \mathcal{A}} x_{ij}^k - \sum_{j:(j,i) \in \mathcal{A}} x_{ji}^k = b_i^k \quad \forall i \in \mathcal{N}, \forall k \in \mathcal{C} \quad (1) \\ & \sum_{k \in \mathcal{C}} x_{ij}^k \leq u_{ij} y_{ij} \quad \forall (i, j) \in \mathcal{A} \quad (2) \\ & x_{ij}^k \leq d_{ij}^k y_{ij} \quad \forall (i, j) \in \mathcal{A}, \forall k \in \mathcal{C} \quad (3) \\ & x_{ij}^k \geq 0 \quad \forall (i, j) \in \mathcal{A}, \forall k \in \mathcal{C} \quad (4) \\ & y_{ij} \in \{0, 1\} \quad \forall (i, j) \in \mathcal{A} \quad (5) \end{aligned}$$

The objective function is to minimize the sum of variable transport costs for commodities and fixed costs for arc installations.

Inequalities in (1) build $|\mathcal{N}||\mathcal{C}|$ flow constraints, which define each node as an origin, destination or transport node.

In (2) we obtain $|\mathcal{A}|$ capacity constraints. They couple the flow variables x with design variables y . The $|\mathcal{A}||\mathcal{C}|$ additional capacity constraints in (3) are redundant for all valid and optimal solutions of the model. They are called the *strong* constraints because they improve the LP-bound significantly. The $|\mathcal{A}||\mathcal{C}|$ non-zero constraints for the flow variables in (4) and $|\mathcal{A}|$ integer constraints in (5) complete the model.

2.1 Lagrangian Relaxation

If we relax the flow constraints (1) we obtain the following Lagrangian relaxation.

For each of the $|\mathcal{C}||\mathcal{N}|$ flow constraints we propose a Lagrange-multiplier ω_i^k . For a fixed $\omega \in \mathcal{R}^{|\mathcal{C}||\mathcal{N}|}$ the knapsack relaxation is

$$\begin{aligned} z_R(\omega) = \min \quad & \sum_{k \in \mathcal{C}} \sum_{(i,j) \in \mathcal{A}} c_{ij}^k x_{ij}^k + \sum_{(i,j) \in \mathcal{A}} f_{ij} y_{ij} \\ & + \sum_{k \in \mathcal{C}} \sum_{i \in \mathcal{N}} \omega_i^k \left(\sum_{j:(i,j) \in \mathcal{A}} x_{ij}^k - \sum_{j:(j,i) \in \mathcal{A}} x_{ji}^k - b_i^k \right) \end{aligned}$$

subject to constraints (2)–(5).

After reformulation we obtain:

$$\begin{aligned}
 z_R(\omega) = \min & \sum_{(i,j) \in \mathcal{A}} [\sum_{k \in \mathcal{C}} (c_{ij}^k + \omega_i^k - \omega_j^k) x_{ij}^k + f_{ij} y_{ij}] \\
 & - \sum_{k \in \mathcal{C}} \sum_{i \in \mathcal{N}} \omega_i^k b_i^k \\
 \text{s.t.} & \\
 & \left. \begin{aligned}
 \sum_{k \in \mathcal{C}} x_{ij}^k &\leq u_{ij} y_{ij} \\
 x_{ij}^k &\leq d_{ij}^k y_{ij} \quad \forall k \in \mathcal{C} \\
 x_{ij}^k &\geq 0 \quad \forall k \in \mathcal{C} \\
 y_{ij} &\in \{0, 1\}
 \end{aligned} \right\} \forall (i, j) \in \mathcal{A}. \tag{1}
 \end{aligned}$$

The occurring subproblems decompose independently for each arc and are relatively simple fractional knapsack problems. They can be solved very fast by a greedy algorithm.

There are two different optimization algorithms for the Lagrangian multiplier problem: the subgradient search method and the bundle method. Besides our own implementation of the subgradient search method we have integrated the bundle method solver of Antonio Frangioni [Fra97, CFG01] into our system.

2.2 System Architecture

The implemented system for the capacitated network design problem uses the branch-and-bound and the relax-and-cut algorithm. Both are using the knapsack Lagrangian relaxation (and the additional cuts respectively). As already mentioned, subgradient search or bundle method solvers can be used for the calculation of the lower bounds in the branch-and-bound nodes. Two variable branching strategies and several exact and heuristic variable fixing algorithms are available. The description of these system components lies beyond the scope of this paper. Details can be found in [SKK02a, SKK02b]. The focus here is on the relax-and-cut algorithm and the implemented valid inequalities.

3 Relax-and-Cut for CNDP

In this section we describe how the Lagrangian-relaxation based branch-and-bound algorithm can be extended to use valid inequalities for the capacitated network design problem. The first type of the inequalities is based on the following idea. The capacity of a network cut which separates origins and destinations of some commodities must be large enough to route the transport demand of these commodities. The second type uses a set of the integer design variables and cuts off some nodes of the branch-and-bound tree with high costs. To preserve the computational structure of the subproblems and to avoid destroying the simplicity, the cuts are not inserted into the subproblems. New Lagrangian multipliers are defined and we add them to the objective function.

3.1 Cover Inequalities

In every network cut (S, \bar{S}) in G there are arc subsets $C \subseteq (S, \bar{S})$ with the property that the overall arc capacity in $(S, \bar{S}) \setminus C$ is not sufficient for routing

from S to \bar{S} . In this case at least one arc from the set C must be opened to obtain a valid solution.

For a given cutset inequality $\sum_{(i,j) \in (S,\bar{S})} u_{ij} y_{ij} \geq r_{(S,\bar{S})}$ let $Y_{(S,\bar{S})}$ be the set of y values s.t.: $Y_{(S,\bar{S})} = \left\{ y \in \{0,1\}^{|(S,\bar{S})|} : \sum_{(i,j) \in (S,\bar{S})} u_{ij} y_{ij} \geq r_{(S,\bar{S})} \right\}$.

We denote an arc set $C \subseteq (S,\bar{S})$ as a *cover* for (S,\bar{S}) if the capacity of $(S,\bar{S}) \setminus C$ is not sufficient: $\sum_{(i,j) \in (S,\bar{S}) \setminus C} u_{ij} < r_{(S,\bar{S})}$.

The cover is a *minimal cover* if it is sufficient to open any arc of the arc set to cover the demand: $\sum_{(i,j) \in (S,\bar{S}) \setminus C} u_{ij} + u_{pq} \geq r_{(S,\bar{S})} \quad \forall (p,q) \in C$

Proposition 1. *Let $C \subseteq (S,\bar{S})$ be a cover of (S,\bar{S}) . For each $y \in Y_{(S,\bar{S})}$ and therefore for every valid solution the **cover inequality** is valid: $\sum_{(i,j) \in C} y_{ij} \geq 1$.*

In case C is minimal, the inequality defines a facet of $Y'_{(S,\bar{S})} = Y_{(S,\bar{S})} \cap \{y : y_{ij} = 1 \ \forall (i,j) \in (S,\bar{S}) \setminus C\}$ and is a $(|C| - 1)$ -dimensional facet of $\text{conv}(Y_{(S,\bar{S})})$.

For separation of the cover inequalities we use a similar heuristic for a binary knapsack problem as in [CCG03]. After separation we start a lifting procedure to strengthen the inequalities. Due to space limitations we omit the description of the algorithms here.

3.2 Local Cuts

The second type of cuts is a generalization of a classical reduced cost variable fixing procedure. If a tight upper bound for the CNDP is known, it is an easy task to check whether a variable y_{ij} can still be set to 0 or 1 without worsening the lower bound too much.

Let $(\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_*)$ be a branch-and-bound node. Hereby \mathcal{A}_0 is the set of arcs which are fixed to 0 (resp. \mathcal{A}_1 to 1) and \mathcal{A}_* is the set of unfixed arcs. Denote the reduced costs for the design variable y_{ij} as \hat{g}_{ij} . The Lagrangian objective can be written as:
$$z_R(\omega) = \min_{y \in \{0,1\}^{|\mathcal{A}_*|}} \sum_{(i,j) \in \mathcal{A}_*} \hat{g}_{ij} y_{ij} + \sum_{(i,j) \in \mathcal{A}_1} \hat{g}_{ij} - \omega^T b$$

If the best known valid solution has the costs \bar{z}_{CNDP} , we can show:

Proposition 2. *a) Let $T_+ \subseteq \{(i,j) \in \mathcal{A}_* : \hat{g}_{ij} > 0\}$ be the subset with positive reduced costs, s.t. $z_R(\omega) + \hat{g}_{pq} + \hat{g}_{rs} \geq \bar{z}_{CNDP} \quad \forall (p,q), (r,s) \in T_+$ with $(p,q) \neq (r,s)$.*

All valid solutions which are in the subtree of the current node and have more than one variable from T_+ equal to one have at least the cost value \bar{z}_{CNDP} . For this reason we can add the following cuts which are locally valid for the given subtree: $\sum_{(i,j) \in T_+} y_{ij} \leq 1$

b) Similar cuts can be proposed for the set $T_- \subseteq \{(i,j) \in \mathcal{A}_ : \hat{g}_{ij} < 0\}$: $\sum_{(i,j) \in T_-} y_{ij} \geq |T_-| - 1$*

3.3 Relax-and-Cut Algorithm

Cutting planes algorithms are often used to strength the LP-relaxation in a root node of a branch-and-bound tree. In cases where valid inequalities are defined in

every search node, one obtains the branch-and-cut algorithm. In this section we discuss a similar technique (*relax-and-cut*) in a Lagrangian-relaxation setting. For an overview on relax-and-cut algorithms see for example [Gui98, RG05].

In the following illustrative example, the mixed-integer linear problem

$$(P) \quad z_P = \min \quad c^T x$$

$$\text{s.t. } Ax \geq b \tag{2}$$

$$x \in X = \{x \in Z^n : Dx \geq q\}. \tag{3}$$

consists of two types of constraints: the simple constraints (3) and the complicated constraints (2).

The Lagrangian relaxation of (2) gives us the following model: $z_{LR}(\omega) = \min\{c^T x + \omega^T(b - Ax) : x \in X\}$

This problem can be solved easily for a given $\omega \geq 0$. Now we propose a family of valid inequalities $\alpha^{kT} x \geq \beta^k$, $k \in \mathcal{K}$ which can reduce the gap between the optimal value z_P and the optimal value z_{LM} of the Lagrangian multiplier problem: $z_{LM} = \max\{z_{LR}(\omega) : \omega \geq 0\}$

During the subgradient search method or the bundle method we generate violated inequalities $\alpha^{kT} x \geq \beta^k$, $k \in \mathcal{K}$. For these cuts we define new Lagrangian multipliers ν_k . In later iteration, the penalty value $\nu_k(\beta^k - \alpha^{kT} x)$ is inserted into the Lagrangian objective function. Thus the Lagrangian multiplier problem (3.3) receives new variables $\nu_k \geq 0$ and a potentially larger optimal value z_{LM} . The question is in which cases the introduced cuts lead to a higher value z_{LM} .

The Lagrangian multiplier problem has the same optimal objective value as the following problem:

$$(L) \quad z_L = \min\{c^T x : Ax \geq b, x \in \text{conv}(X)\}.$$

The updated problem with a cut is:

$$(L') \quad z'_L = \min\{c^T x : Ax \geq b, \alpha^T x \geq \beta, x \in \text{conv}(X)\}.$$

The feasible regions of (L) und (L') are shown in Figure 1.

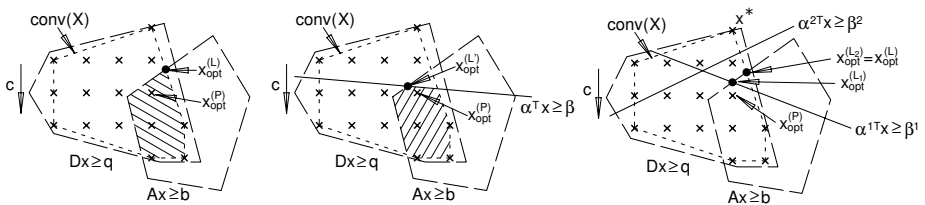


Fig. 1. The feasible sets of (L) and (L') with the cut $\alpha^T x \geq \beta$. Efficient and inefficient cuts for z_{LM} .

The cut $\alpha^T x \geq \beta$ is an efficient cut if all the optimal solutions of (L) are separated by the cut.

Let $x^* \in X$ be the optimal solution of the Lagrangian subproblem (3.3) and Q the feasible region of (L) . Since every point in $Q \cap X$ is feasible for (P) , points of X , which are separated by a valid inequality for (P) , are outside of Q . This means that by separating x^* by a cut $\alpha^T x \geq \beta$ it holds $x^* \notin Q$. For this reason it is not clear whether we cut off some points from Q or not (see also [Gui98]).

In Figure 1 we see two cuts $\alpha^{kT} x \geq \beta^k$, $k = 1, 2$ which separate x^* from (P) . The first one cuts off the optimal solution of (L) . The second cut is inefficient - it does not improve the bound z_{LM} .

In the relax-and-cut algorithms we are looking for violated cuts which cut off the current Lagrangian solution x^* . It is impossible to check whether the cut is efficient because we do not know the optimal Lagrangian solution during one iteration of the subgradient search or bundle method optimization. Nevertheless, there are a lot of successful applications of the relax-and-cut algorithm in the literature. See for example applications for steiner tree problems in [Luc92], the weighted clique problem in [HFK01] and the vehicle routing problem in [MLM04]. In this work we show that the relax-and-cut algorithm is also successful for the capacitated network design problem.

4 Numerical Results

The evaluation of the presented algorithms is carried out on a series of benchmarks which are widely used for the capacitated network design problem. In table 1 we can see the problem characteristics. The description of the problem data generator can be found in [CFG01].

Table 1. Benchmark data instances

	Canad-R1	Canad-R2	PAD	Canad-C
Number of nodes	10	20	12-24	20-30
Number of arcs	35-83	120-318	50-440	230-700
Number of commodities	10-50	40-200	50-160	40-400
Number of instances	72	81	41	31

The instances from Canad-R1 and PAD can be solved to optimality. The sets Canad-R2 and Canad-C are solved partially. For many instances optimal solutions are still unknown. In such cases we give the best known lower and upper bounds. The tables in the appendix contain all the details concerning the benchmark instances and the objective values.

4.1 Experiments

The experiments in this paper were carried out on a Pentium III processor with 3 GHz and 1 Gbyte memory. For the experiments in Figure 4 we used Pentium III processor with 850 MHz and 512 Mbyte memory.

In order to make the results less sensitive to some outliers in our experiments, we are using the concept of *performance profiles*. Let S be the set of the approaches to compare to. Let P be the set of the benchmark instances and $t_{p,s}$ the time of the approach s for solving the problem p . The performance profile of the approach $s' \in S$ on the set P is the function $Q_{s'}(r)$. It is defined as the number of instances in P which were solved in time $r \cdot \min\{t_{p,s} : s \in S\}$ or faster (see [DM02]):

$$Q_{s'}(r) = \left| \left\{ p \in P : \frac{t_{p,s'}}{\min\{t_{p,s} : s \in S\}} \leq r \right\} \right|.$$

Experiment 1: Comparison of the Exact Methods. In Figure 2 we use the benchmark set **PAD**. The performance profiles of CPLEX 9.0, the subgradient based solver **NDBB**, and the bundle-based solver **NDBC** are shown. The task was to compute the optimal solution of the given CNBP problem instance.

We can see that the **NDBC** solver is the better solver in this comparison. The running time on the whole benchmark and also the mean performance ratio is better. The cumulative distribution of the performance profiles confirm this result.

Experiment 2: Comparison Against Other Heuristical Solvers. In Figure 3 we present the results on the benchmark set **Canad-C**. The results of the following solvers were obtained from the authors of the papers: TABU-ARC and TABU-CYCLE [GCG03], TABU-PATH [CGF00], PATH-RELINKING [GCG04], SS/PL/ID [CGH04].

The heuristical variable fixing in **NDBC** were first presented in [HY00] and were adapted in our project to be used with a bundle method solver. It was stopped in these experiments after 600 seconds for each instance. The exact variant of the **NDBC** solver was stopped after 1800 seconds. We compared the solution quality obtained by the approaches. The cumulative distributions show the plots for the two variants of the CPLEX 9.0 solver. The first one was configured to obtain good solutions as fast as possible and was stopped after 600 seconds for each instance. The standard CPLEX 9.0 branch-and-cut algorithm ran for 14400 seconds.

As a result we can conclude that the heuristical version of the **NDBC** solver delivers the best solution quality compared to all other solvers. It also outperforms the other solvers in terms of running time.

Experiment 3: Impact of the Cuts in the Relax-and-Cut Algorithm. This is the comparison of four different configurations of the **NDBC** solver.

Configuration	Generation of cuts
NDBC 00	both cuts types are not generated,
NDBC 01	only the cover cuts are generated,
NDBC 10	only local cuts are generated,
NDBC 11	both cuts types are generated.

In Figure 4 we used a subset of the benchmark **CANAD-R2**. We can observe that the cover cuts reduce the running time by 23% and the number of generated nodes by half. The best configuration is the one with both types of cuts.

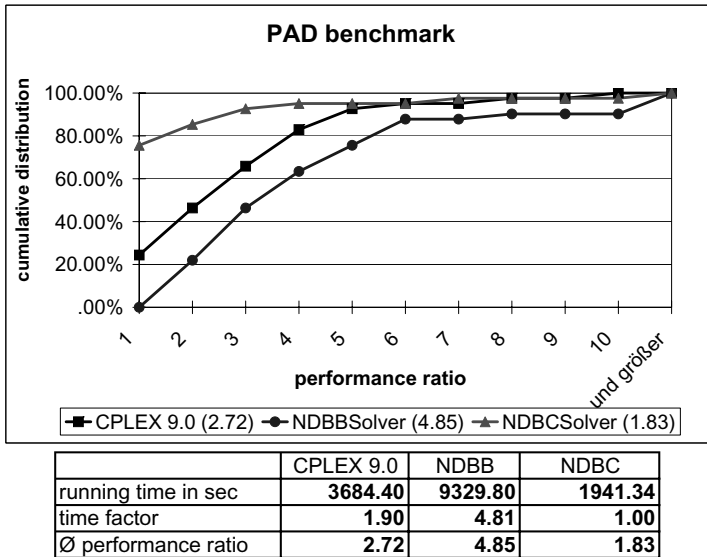


Fig. 2. Experiment 1: Comparison of the exact methods

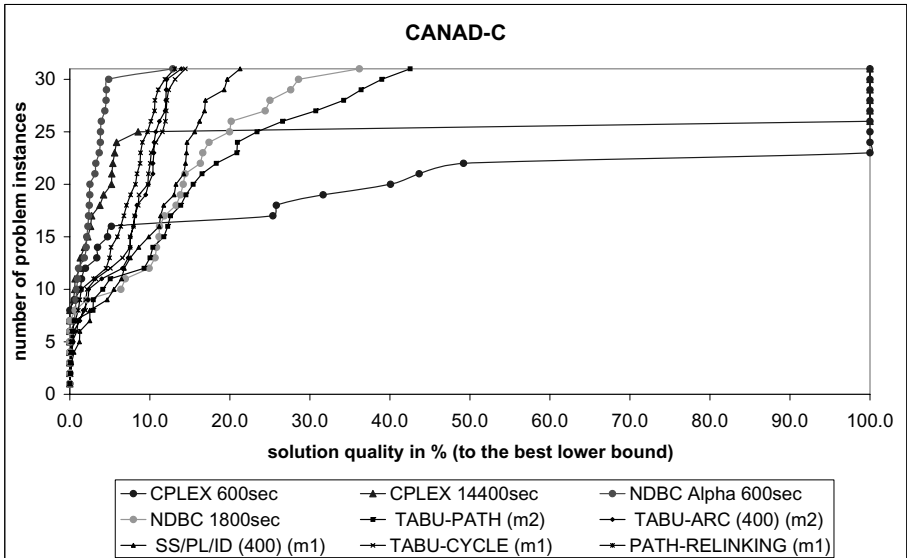


Fig. 3. Experiment 2: Comparison against other heuristical solvers

Experiment 4: Impact of the Cuts in the Heuristical Setting. In Figure 5 we can observe that the cover cuts also significantly improve the solution quality. The local cuts are not effective in this setting. The results are nearly identical for a configuration with or without them.

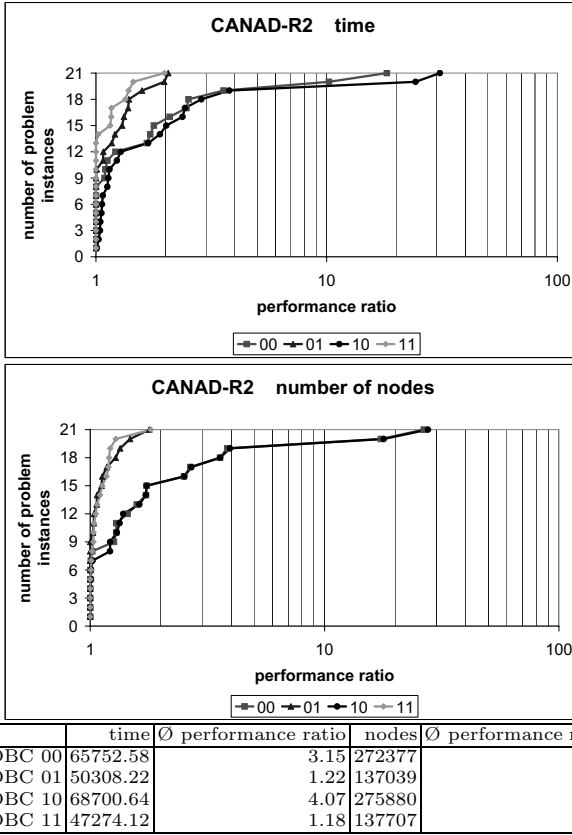


Fig. 4. Experiment 3: Impact of the cuts in the relax-and-cut algorithm

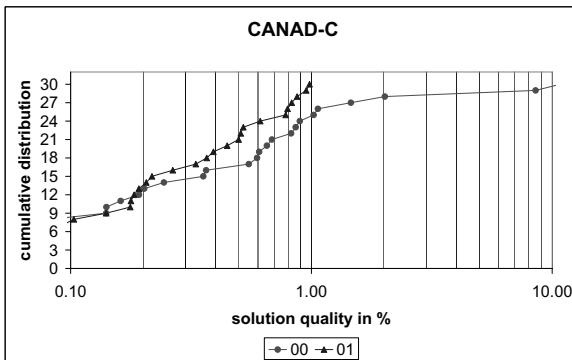


Fig. 5. Experiment 4: Impact of the cuts in the heuristical setting

5 Conclusion

Our experimental evaluation shows that the newly presented relax-and-cut algorithm in the **NDBC** solver, outperforms other exact solvers on the **PAD** benchmark and all the heuristical solvers on the benchmark **CANAD-C**. To our knowledge, this is the most efficient solver for the capacitated network design problem.

The incorporation of cover inequalities and local cuts in the relax-and-cut algorithm improves the overall performance significantly.

While many of the system components, namely the subgradient solver, the bundle method solver, the relax-and-cut algorithm itself, and the presented cuts for CNDP are conceptually not new, this work makes a valuable contribution. The overall system is a demonstration of what is currently possible in the area of capacitated network design.

The results also show that research on some additional cuts for CNDP could improve the performance of the relax-and-cut algorithm even further.

Acknowledgements

We would like to express our gratitude to Antonio Frangioni for providing the bundle method solver for our system and to Bernard Gendron for providing the benchmark data and the numerical results of other approaches for comparisons.

References

- [BMM97] A. Balakrishnan, T. Magnanti, and P. Mirchandani. Network design. *Annotated Bibliographies in Combinatorial Optimization*, pages 311–334, 1997.
- [CCG03] M. Chouman, T.G. Crainic, and B. Gendron. A cutting-plane algorithm based on cutset inequalities for multicommodity capacitated fixed charge network design. Technical report, Centre for Research on Transportation, Montreal, Canada, 2003.
- [CFG01] T.G. Crainic, A. Frangioni, and B. Gendron. Bundle-based relaxation methods for multicommodity capacitated fixed charge network design problems. *Discrete Applied Mathematics*, 112:73–99, 2001.
- [CGF00] T.G. Crainic, M. Gendreau, and J.M. Farvolden. A simplex-based tabu search method for capacitated network design. *INFORMS Journal on Computing*, 12(3):223–236, 2000.
- [CGH04] T.G. Crainic, B. Gendron, and G. Hernu. A slope scaling/Lagrangian perturbation heuristic with long-term memory for multicommodity fixed-charge network design. *Journal of Heuristics*, 10(5):525–545, 2004.
- [Cra00] T.G. Crainic. Service network design in freight transportation. *European Journal of Operational Research*, 122:272–288, 2000.
- [DM02] E.D. Dolan and J.J. More. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201 – 213, 2002.
- [Fra97] A. Frangioni. *Dual Ascent Methods and Multicommodity Flow Problems*. PhD thesis, Dipartimento di Informatica, Universit di Pisa, 1997.

- [GC94] B. Gendron and T.G. Crainic. Relaxations for multicommodity capacitated network design problems. Technical report, Center for Research on Transportation, Montreal, Canada, 1994.
- [GCG03] I. Ghamlouche, T.G. Crainic, and M. Gendreau. Cycle-based neighbourhoods for fixed-charge capacitated multicommodity network design. *Operations research*, 51(4):655–667, 2003.
- [GCG04] I. Ghamlouche, T.G. Crainic, and M. Gendreau. Path relinking, cycle-based neighbourhoods and capacitated multicommodity network design. *Annals of Operations research*, 131(1–4):109–133, 2004.
- [Gui98] M. Guignard. Efficient cuts in lagrangean 'relax-and-cut' schemes. *European Journal of Operational Research*, 105:216–223, 1998.
- [HFK01] M. Hunting, U. Faigle, and W. Kern. A lagrangian relaxation approach to the edge-weighted clique problem. *European Journal of Operational Research*, 131(1):119–131, 2001.
- [HY00] K. Holmberg and D. Yuan. A lagrangean heuristic based branch-and-bound approach for the capacitated network design problem. *Operations Research*, 48:461–481, 2000.
- [Luc92] Abilio Lucena. Steiner problem in graphs: Lagrangian relaxation and cutting planes. In *COAL Bulletin*, volume 21, pages 2–8. Mathematical Programming Society, 1992.
- [MLM04] C. Martinhon, A. Lucena, and N. Maculan. Stronger k-tree relaxations for the vehicle routing problem. *European Journal of Operational Research*, 158(1):56–71, 2004.
- [MW84] T.L. Magnanti and R.T. Wong. Network design and transportation planning: Models and algorithms. *Transportation Science*, 18(1):1–55, 1984.
- [RG05] T. Ralphs and M. Galati. Decomposition and dynamic cut generation in integer programming. Technical report, Lehigh University, PA, 2005. To appear in *Mathematical Programming*.
- [SKK02a] M. Sellmann, G. Kliewer, and A. Koberstein. Lagrangian cardinality cuts and variable fixing for capacitated network design. In R. Moehring and R. Raman, editors, *Proceedings of the 10th Annual European Symposium on Algorithms (ESA)*, Springer, LNCS 2461, pages 845–858, 2002.
- [SKK02b] M. Sellmann, G. Kliewer, and A. Koberstein. Lagrangian cardinality cuts and variable fixing for capacitated network design. Technical report tr-ri-02-234, University of Paderborn, 2002.

On the Price of Anarchy and Stability of Correlated Equilibria of Linear Congestion Games^{*,**,***}

George Christodoulou and Elias Koutsoupias

National and Kapodistrian University of Athens,
Department of Informatics and Telecommunications
{gchristo, elias}@di.uoa.gr

Abstract. We consider the price of stability for Nash and correlated equilibria of linear congestion games. The price of stability is the optimistic price of anarchy, the ratio of the cost of the best Nash or correlated equilibrium over the social optimum. We show that for the sum social cost, which corresponds to the average cost of the players, every linear congestion game has Nash and correlated price of stability at most 1.6. We also give an almost matching lower bound of $1 + \sqrt{3}/3 = 1.577$.

We also consider the price of anarchy of correlated equilibria. We extend existing results about Nash equilibria to correlated equilibria and show that for the sum social cost, the price of anarchy is exactly 2.5, the same for pure and mixed Nash and for correlated equilibria. The same bound holds for symmetric games as well. We also extend the results about Nash equilibria to correlated equilibria for weighted congestion games and we show that when the social cost is the total latency, the price of anarchy is $(3 + \sqrt{5})/2 = 2.618$.

1 Introduction

Recently, a new vigorous subfield of computer science emerged which studies how the viewpoint and behavior of users affects the performance of computer networks or systems, by modeling the situation as a game.

One of the most important questions in game theory is what is the correct solution concept of a game. There are many proposals but three types of equilibria stand out in the literature of non-cooperative game theory. The first and stronger equilibrium occurs when there are *dominant* strategies, in which each player has an optimal strategy independently of what the other players do. Unfortunately not every game has such a solution (and sometimes even if it has

* Research supported in part by the IST (FLAGS, IST-2001-33116) programme.

** Research supported in part by the programme ΕΠΕΑΕΚ ΙΙ under the task “ΠΥΘΑΓΟΡΑΣ-ΙΙ: ΕΝΙΣΧΥΣΗ ΕΡΕΥΝΗΤΙΚΩΝ ΟΜΑΔΩΝ ΣΤΑ ΠΑΝΕΠΙΣΤΗΜΙΑ (project title: *Algorithms and Complexity in Network Theory*)” which is funded by the European Social Fund (75%) and the Greek Ministry of Education (25%).

*** Research supported in part by the programme ΙΕΝΕΔ 2003 of General Secretariat for Research and Technology (project title: *Optimization Problems in Networks*).

one, as in the case of the game of prisoner's dilemma, it may lead to unexpected solutions). The second and the most well-known equilibrium is the *Nash equilibrium*, whose existence is assured for every finite game by the famous theorem by Nash. The third type of equilibria is the notion of *correlated equilibrium*, introduced by Aumann [1], and which is almost as widely studied as the Nash equilibria; recently, this kind of equilibria is the focus of computational studies (see for example [22,21]).

A correlated strategy for a game is a probability distribution over all the possible pure strategy profiles. One can interpret the situation as follows: there is a trusted mediator who performs the random experiment and announces the resulted strategies to each player in private. The players although they know the distribution, they are not informed about the outcome of the experiment but just about their own strategy. They may choose to follow or not the mediator's advice according to their utility function. A correlated strategy is a correlated equilibrium if no player has any reason to unilaterally disobey to the mediator's advice. Roughly speaking this paradigm is also an interpretation for mixed Nash equilibria with the exception that the distribution is formed by independent random experiments (one for each player) over the possible pure strategies for each player. That, is Nash equilibria is the rank 1 restriction of correlated equilibria.

The three types of equilibria (dominant, Nash, and correlated) are related by inclusion: the set of Nash equilibria contains the dominant strategies and the set of correlated equilibria contains all Nash equilibria. Thus, by Nash's theorem, the existence of correlated equilibria is also guaranteed for every finite game.

One of the main tools in the study of selfish behavior is the *price of anarchy* [14,20], a measure that compares the worst case performance Nash equilibrium to that of the optimal allocation. Naturally, the concept of the price of anarchy extends to correlated equilibria. If we know only that the players play at some equilibrium, the price of anarchy bounds the deterioration of system performance due to selfish behavior. On the other hand, there is the optimistic point of view in which the players are guided to play at the best Nash equilibrium. Especially with correlated equilibria, the latter makes much more sense: The mediator who selects the probability distribution, the correlated equilibrium, and presents it to the players, can select the correlated equilibrium with minimum system cost. In other words, one can view correlated equilibria as a mechanism for enforcing good behavior on selfish users. The optimistic price of anarchy of the best equilibrium is also called *price of stability* [3].

In this paper we consider the price of anarchy and stability of Nash and correlated equilibria of congestion games, a class of games which suitably models traffic networks and abstracts well many game-theoretic situations in networks and systems. Congestion games, introduced by Rosenthal [23], provide a common framework for almost all previous studies of price of anarchy which originated in [14] and [26]. Congestion games have the fundamental property that a pure Nash equilibrium always exists. In [18], it is proved that congestion games (also called potential games) are characterized by a potential: the local optima of the potential correspond exactly to pure Nash equilibria (see [8] for computational issues related to this issue).

Whether one studies the price of anarchy or the price of stability, a critical matter is the definition of the social optimum: the system allocation mostly desired by the system designer. From the system designers point of view there are two natural notions of social cost: the *maximum* or the *average* (or sum) cost among the players. For the original model of parallel links or machines in [14], the social cost was the maximum cost among the players, which is equal to the makespan. For the Wardrop model studied by Roughgarden and Tardos [26], the social cost was the average player cost. A third social cost is the *total latency* which is the sum of squares of the loads on each facility. For unweighted games this is identical to the average (or sum) cost among the players.

Here we deal mainly with the average social cost but we also consider the maximum social cost and the total latency social cost (for weighted congestion games). We also consider the price of anarchy of the natural subclass of symmetric congestion games, where the available actions are the same for all the players.

1.1 Our Results

We study linear general—not only network—congestion games with cost (latency) functions of the form $f_e(k) = a_e k + b_e$ with nonnegative coefficients. We focus mainly on the sum social cost which is the sum of the cost of all players and we consider all types of equilibria: dominant strategies, pure and mixed Nash equilibria, and correlated equilibria.

These equilibria are related by inclusion and this hierarchy allows us to look for the strongest possible results. In particular, when we obtain a lower bound on the price of stability or the price of anarchy for dominant strategies, this lower bound holds for all types of equilibria. (It is important to emphasize that this holds for the price of stability because a strong dominant strategy implies unique Nash and correlated equilibrium). And on the other end, when we obtain an upper bound for correlated equilibria, this holds for all types of equilibria. Interestingly—but not entirely unexpectedly—such general results are easier to prove in some cases (when we are not distracted by the additional structure of the specific subproblems).

This is an additional reason for studying the price of stability and anarchy of correlated equilibria. Not only correlated equilibria is a natural well-motivated and well-studied class of equilibria, but it is also an appropriate (and computationally feasible) generalization of Nash equilibria which allows some of the results to be simplified and strengthened.

Price of Stability: For linear congestion games we give an upper bound of 1.6 for Nash and correlated equilibria (Theorem 1). Although bounding directly the price of stability seems hard—after all, we need to bound the best not the worst equilibrium—we resort to a clever trick by making use of the potential of congestion games. More specifically, instead of bounding the cost of the best equilibrium, we bound the cost of the pure Nash equilibrium which has minimum potential. Since every local optimum of the potential corresponds to a pure Nash equilibrium (by the handy theorem of Rosenthal [23,18]), such a Nash equilibrium is guaranteed to exist. In fact, the proof of Theorem 1 does not even

need to consider the Nash equilibrium with minimum potential. All we need to consider is that the potential of the Nash equilibrium is less than the potential of the optimal strategies.

We give a non-trivial lower bound of $1 + \frac{\sqrt{3}}{3} \approx 1.577$ for dominant strategies (Theorem 2). This is a surprisingly strong result: It states in the strongest possible way that for some games, selfishness deteriorates the efficiency of some systems by approximately 58%. It is also perhaps the most technical part of this work. Naturally, both the upper and lower bounds hold for all types of equilibria (dominant strategies, pure and mixed Nash, and correlated equilibria). An open problem is to close the small gap between 1.577 and 1.6. We believe that our lower bound is tight, but our upper bound approach cannot reach this bound without substantial restructuring.

We also observe that for the max social cost (i.e., the maximum cost among the players) the price of stability is $\Theta(\sqrt{N})$ (Theorem 3). This follows by a minor modification to the lower bound (about pure Nash equilibria) given in [4].

Price of anarchy: For linear congestion games, we extend some of the results of our STOC'05 paper [4] on the price of anarchy. There we showed bounds on Nash equilibria which we extend here to correlated equilibria. At the same time we strengthen the bounds. More specifically, we show that the correlated price of anarchy of the sum social cost is 2.5 for the asymmetric case (Theorem 4) and $\frac{5N-2}{2N+1}$ for the symmetric case (Theorem 5), where N is the number of players. Since in [4], we had matching lower bounds for pure Nash equilibria, these are also tight bounds for pure and mixed Nash and correlated equilibria.

We also extend the results of [2] about the price of anarchy of Nash equilibria for weighted linear congestion games when the social cost is the total latency: The price of anarchy of correlated equilibria is $\frac{3+\sqrt{5}}{2} \approx 2.618$ (Theorem 6). Although we prove a more general result, our proof is substantially simpler.

With these results we resolve many open problems in this area, although there are plenty of them left. Our results together with the existing literature suggest an intriguing dichotomy between the sum (or average) social cost and the maximum social cost. For the sum social cost, the price of anarchy of pure Nash equilibria (or even of dominant strategies, if they exist) is almost equal to the price of anarchy of the much larger class of mixed Nash equilibria and even correlated equilibria. On the contrary, for the max social cost, pure Nash equilibria have much smaller price of anarchy than mixed Nash equilibria. Relevant to this meta-theorem is the fully-mixed-conjecture, which first appeared in [16], and states that for every game of the original model of [14], the fully mixed Nash equilibrium has the worst price of anarchy among all Nash equilibria.

1.2 Related Work

The closer works in spirit, results, and techniques are the STOC'05 papers [4,2]. Both these papers study the price of anarchy of pure and mixed Nash equilibria for linear congestion games and congestion games with polynomial delay functions. In particular, both papers show that price of anarchy of *pure* Nash equilibria of linear congestion games is 2.5 for the average social cost (and the total latency cost which is identical in this case). Paper [4] also shows that the

same bound holds also for symmetric games for both the average and the maximum social cost but it gets up to $\Theta(\sqrt{N})$ for the maximum social cost (in the general case). It also gives a 2.618 bound for mixed equilibria and the average social cost. On the other hand, [2] considers weighted congestion games for the total latency social cost and shows that the price of anarchy for both pure and mixed equilibria is 2.618.

The study of the price of anarchy was initiated in [14], for (weighted) congestion games of m parallel links. The price of anarchy for the maximum social cost is proved to be $\Omega(\frac{\log m}{\log \log m})$, while in [16,13,7] they proved $\Theta(\frac{\log m}{\log \log m})$. In [7], they extended the result to m parallel links with different speeds and showed that the price of anarchy is $\Theta(\frac{\log m}{\log \log \log m})$. In [6], more general latency functions are studied, especially in relation to queuing theory. For the same model of parallel links, [10] and [15] consider the price of anarchy for other social costs.

In [28], they give bounds for the case of the average social cost for the parallel links model. For the same model and the maximum social cost, [11] showed that the price of anarchy is $\Theta(\log N / \log \log N)$ (a similar, perhaps unpublished, result was obtained by the group of [28]). The case of singleton strategies is also considered in [12] and [15]. In [9], they consider the mixed price of anarchy of symmetric network weighted congestion games, when the network is layered.

The non-atomic case of congestion games was considered in [26,27] where they showed that for linear latencies the average price of anarchy is $4/3$. They also extended this result to polynomial latencies. Furthermore, [5,25] considered the social cost of maximum latency.

2 The Model

A congestion game is a tuple $(N, E, (\mathcal{S}_i)_{i \in N}, (f_e)_{e \in E})$ where $N = \{1, \dots, n\}$ is the set of players, E is a set of facilities, $\mathcal{S}_i \subseteq 2^E$ is a collection of pure strategies for player i : a pure strategy $A_i \in \mathcal{S}_i$ is a set of facilities, and finally f_e is a cost (or latency) function associated with facility e . We are concerned with linear cost functions: $f_e(k) = a_e \cdot k + b_e$ for nonnegative constants a_e and b_e . A *pure strategy profile* $A = (A_1, \dots, A_n)$ is a vector of strategies, one for each player. The cost of player i for the pure strategy profile A is given by $c_i(A) = \sum_{e \in A_i} f_e(n_e(A))$, where $n_e(A)$ is the number of the players using e in A . A pure strategy profile A is a Nash equilibrium if no player has any reason to unilaterally deviate to another pure strategy: $\forall i \in N, \forall s \in \mathcal{S}_i \quad c_i(A) \leq c_i(A_{-i}, s)$, where (A_{-i}, s) is the strategy profile produced if just player i deviates from A_i to s .

The *social cost* of A is either the maximum cost of a player $\text{MAX}(A) = \max_{i \in N} c_i(A)$ or the average of the players' costs. For simplicity, we consider the sum of all costs (which is N times the average cost) $\text{SUM}(A) = \sum_{i \in N} c_i(A)$. These definitions extend naturally to the cases of mixed and correlated strategies (with expected costs, of course).

A *mixed* strategy p_i for a player i , is a probability distribution over his pure strategy set \mathcal{S}_i . A *correlated* strategy q for a set of players, is any probability distribution over the set \mathcal{S} of possible combinations of pure strategies that these players can choose, where $\mathcal{S} = \times_{i \in N} \mathcal{S}_i$. Given a correlated strategy q , the ex-

pected cost of a player $i \in N$ is $c_i(q) = \sum_{s \in \mathcal{S}} q(s)c_i(s)$. A correlated strategy q is a *correlated equilibrium* if q satisfies the following

$$c_i(q) \leq \sum_{s \in \mathcal{S}} q(s)c_i(s_{-i}, \delta_i(s_i)), \quad \forall i \in N, \quad \forall \delta_i(s_i) : \mathcal{S}_i \rightarrow \mathcal{S}_i$$

A congestion game is *symmetric* (or single-commodity) if all the players have the same strategy set: $\mathcal{S}_i = \mathcal{C}$. We use the term “asymmetric” (or multi-commodity) to refer to all games (including the symmetric ones).

The correlated price of anarchy of a game is the worst-case ratio, among all correlated equilibria, of the social cost over the optimum social cost, $opt = \min_{P \in \mathcal{S}} sc(P)$.

$$PA = \sup_{q \text{ is a corr. eq.}} \frac{sc(q)}{opt}$$

The correlated price of stability of a game is the best-case ratio, among all correlated equilibria, of the social cost over the optimum.

$$PS = \inf_{q \text{ is a corr. eq.}} \frac{sc(q)}{opt}$$

When we refer to the price of stability (resp. anarchy) of a class of games, we mean the maximum (or supremum) price of stability (resp. anarchy) among all games in the class.

In weighted congestion games, each player controls an amount of traffic w_i , and the cost of a facility e depends on the total load on the facility. For this case, some of our results involve the total latency social cost. For a pure strategy profile $A \in \mathcal{S}$, the total latency is defined as $C(A) = \sum_{e \in E} \theta_e(A) \cdot f_e(\theta_e(A))$. Notice that the sum and the total latency social costs coincide for unweighted congestion games.

3 The Correlated Price of Stability of Congestion Games

In this section we study the price of stability of congestion games with linear cost functions of the form $f_e(x) = a_e \cdot x + b_e$, with non-negative coefficients a_e and b_e . We will use the following simple lemma:

Lemma 1. *For every nonnegative integers α, β it holds: $\alpha\beta + 2\beta - \alpha \leq \frac{1}{8}a^2 + 2\beta^2$.*

Given a strategy profile A of a congestion game, we denote by $\Phi(A)$ the potential of A , using the potential function introduced in [23]: $\Phi(A) = \sum_{e \in E} \sum_{i=1}^{n_e(A)} f_e(i)$. The potential has the nice property that when $\Phi(A)$ is a local optimum, then A is a pure Nash equilibrium. To establish an upper bound on the price of stability, we simply bound the price of anarchy of the subclass of Nash equilibria whose potential does not exceed the potential of the optimal allocation. Clearly, by the property of the potential, this subclass of Nash equilibria is not empty.

Theorem 1. *Let A be a pure Nash equilibrium and P be any pure strategy profile such that $\Phi(A) \leq \Phi(P)$, then $SUM(A) \leq \frac{8}{5}SUM(P)$. This shows that the correlated price of stability is at most 1.6.*

Proof. For every pure strategy profile $X = (X_1, \dots, X_N)$ we have

$$\begin{aligned} \text{SUM}(X) &= \sum_{i \in N} c_i(X) = \sum_{i \in N} \sum_{e \in X_i} f_e(n_e(X)) = \sum_{e \in E} n_e(X) f_e(n_e(X)) \\ &= \sum_{e \in E} (a_e n_e^2(X) + b_e n_e(X)) \end{aligned}$$

where $n_e(X)$ is the number of the players in the allocation X that use e . We compute the potential for a strategy profile X :

$$\Phi(X) = \sum_{e \in E} \sum_{i=1}^{n_e(X)} f_e(i) = \sum_{e \in E} \sum_{i=1}^{n_e(X)} (a_e \cdot i + b_e) = \frac{1}{2} \text{SUM}(X) + \frac{1}{2} \sum_{e \in E} (a_e + b_e) n_e(X).$$

From the potential inequality $\Phi(A) \leq \Phi(B)$, we have

$$\text{SUM}(A) + \sum_{e \in E} (a_e + b_e) n_e(A) \leq \text{SUM}(P) + \sum_{e \in E} (a_e + b_e) n_e(P),$$

from which we obtain

$$\text{SUM}(A) \leq \sum_{e \in E} (a_e (n_e(P)^2 + n_e(P) - n_e(A)) + b_e (2n_e(P) - n_e(A))). \quad (1)$$

From the Nash inequality we have that for every player i and for every strategy P_i , it holds $c_i(A) \leq c_i(A_{-i}, P_i) \leq \sum_{e \in P_i} f_e(n_e(A) + 1)$. If we sum for all players we get

$$\text{SUM}(A) \leq \sum_{e \in E} n_e(P) f_e(n_e(A) + 1) = \sum_{e \in E} a_e n_e(A) n_e(P) + \sum_{e \in E} (a_e + b_e) n_e(P). \quad (2)$$

So if we sum (1) and (2), and use Lemma 1, we get

$$\begin{aligned} 2\text{SUM}(A) &\leq \sum_{e \in E} a_e (n_e(A) n_e(P) + n_e^2(P) + 2n_e(P) - n_e(A)) + b_e (3n_e(P) - n_e(A)) \\ &\leq \sum_{e \in E} a_e \left(\frac{1}{8} n_e^2(A) + 3n_e^2(P) \right) + \sum_{e \in E} b_e (3n_e(P) - n_e(A)) \\ &\leq \frac{1}{8} \text{SUM}(A) + 3\text{SUM}(P) \end{aligned}$$

and the theorem follows. \square

3.1 Lower Bound

We now provide an almost matching lower bound.

Theorem 2. *There are linear congestion games whose dominant equilibrium — and therefore the Nash and correlated equilibria — have price of stability of the SUM social cost approaching $1 + \sqrt{3}/3 \approx 1.577$ as the number of players N tends to infinity.*

Proof. We describe a game of N players with parameters α , β , and m which we will fix later to obtain the desired properties. Each player i has two strategies A_i and P_i , where the strategy profile (A_1, \dots, A_N) will be the equilibrium and (P_1, \dots, P_N) will have optimal social cost.

There are 3 types of facilities:

- N facilities α_i , $i = 1, \dots, N$, each with cost function $f(k) = \alpha k$. Facility α_i belongs only to strategy P_i .
- $N(N-1)$ facilities β_{ij} , $i, j = 1, \dots, N$ and $i \neq j$, each with cost $f(k) = \beta k$. Facility β_{ij} belongs only to strategies A_i and P_j .
- $\binom{N}{m}$ facilities γ_S , one for each subset S of $\{1, \dots, N\}$ of cardinality m and with cost function $f(k) = k$. Facility γ_S belongs to strategy A_i iff $i \notin S$ and to strategy P_j iff $j \in S$.

We will first compute the cost of every player and every strategy profile. By symmetry, we need only to consider the cost $cost_A(k)$ of player 1 and the cost $cost_P(k)$ of player N of the strategy profile $(A_1, \dots, A_k, P_{k+1}, \dots, P_N)$. We could count the cost that every facility contributes to $cost_A(k)$ and $cost_P(k)$, but this results in complicated sums. A simpler way is to resort to probabilities and consider the contribution of a random facility of each of the 3 types. For example, consider a random facility γ_S which is used by player 1, i.e. $1 \notin S$. The probability that player $j = k+1, \dots, N$ uses this facility is equal to the probability that $j \in S$ which is equal to $m/(N-1)$. Also the probability that player $i = 2, \dots, k$ uses the facility is equal to the probability that $i \notin S$ which is equal to $(N-1-m)/(N-1)$. Therefore the expected number of players that use the facility γ_S is

$$1 + (N-k) \frac{m}{N-1} + (k-1) \frac{N-1-m}{N-1} = \frac{kN + mN - 2km + m - k}{N-1}.$$

Taking into account that there are $\binom{N-1}{m}$ such facilities, the contribution of type 3 facilities to the cost $cost_A(k)$ of player 1 is $\binom{N-1}{m} \frac{kN + mN - 2km + m - k}{N-1}$. With similar but simpler considerations we compute the contribution to $cost_A(k)$ of facilities of the second type to be $(2N-k-1)\beta$. Therefore,

$$cost_A(k) = (2N-k-1)\beta + \binom{N-1}{m} \frac{kN + mN - 2km + m - k}{N-1}.$$

Similarly, we compute

$$cost_P(k) = \alpha + (N+k-1)\beta + \binom{N-1}{N-m} \frac{kN + mN - 2km - m + k}{N-1}.$$

(In fact by symmetry, and with the exception of the term α , the cost $cost_P(k)$ results from $cost_A(k)$ when we replace k and m by $N-k$ and $N-m$, respectively.)

We now want to select the parameters α and β so that the strategy profile (A_1, \dots, A_N) is dominant. Equivalently, at every strategy profile $(A_1, \dots, A_k,$

P_{k+1}, \dots, P_N), player i , $i = 1, \dots, k$, has no reason to switch to strategy P_i . This is expressed by the constraint

$$\text{cost}_A(k) \leq \text{cost}_P(k - 1), \quad \text{for every } k = 1, \dots, N. \tag{3}$$

Magically, all these constraints are satisfied by equality when

$$\alpha = \binom{N}{m} \frac{N^2 - 2m - 2Nm + N}{2N}, \quad \text{and } \beta = \binom{N}{m} \frac{N^2 + 4m^2 - 4Nm - N}{2N(N - 1)},$$

as one can verify with straightforward, albeit tedious, substitution. (The mystery disappears when we observe that both $\text{cost}_A(k)$ and $\text{cost}_P(k)$ are linear in k .)

In summary, for the above values of the parameters α and β , we obtain the desired property that the strategy profile (A_1, \dots, A_N) is a dominant strategy. If we increase α by any small positive ϵ , inequality (3) becomes strict and the dominant strategy is unique (therefore unique Nash and correlated equilibrium).

We now want to select the value of the parameter m so that the price of anarchy of this equilibrium is as high as possible. The price of anarchy is $\text{cost}_A(N)/\text{cost}_P(0)$ which for the above values of α and β can be simplified to

$$pa = \frac{3N^2 + 6m^2 - 8Nm - N}{2N^2 + 6m^2 - 6Nm - 2m}.$$

For $m/N \approx 1/2 - \sqrt{3}/6$, the price of anarchy tends to $pa = 1 + \sqrt{3}/3 \approx 1.577$, as N tends to infinity. \square

The above lower bound is based on asymmetric games. It is open whether a similar result can be obtained for symmetric games.

Theorem 3. *The price of stability for dominant strategies for asymmetric congestion games and for maximum social cost is $\Theta(\sqrt{N})$.*

Proof. A minor modification of the example in the lower bound in [4] works. We simply add a small term ϵ to the bad Nash equilibrium case, in order to turn the strategies into dominant ones. On the other hand, the price of stability cannot be more than the pure price of anarchy of which shown in [4] to be $O(\sqrt{N})$. \square

4 The Correlated Price of Anarchy of Congestion Games

We now turn our attention to the correlated price of anarchy of congestion games, again for linear cost functions of the form $f_e(x) = a_e \cdot x + b_e$, with non-negative coefficients a_e and b_e . We consider the sum (or average) social cost. The following is a simple fact which will be useful in the proof of the next theorem.

Lemma 2. *For every pair of nonnegative integers α, β , it holds $\beta(\alpha + 1) \leq \frac{1}{3}\alpha^2 + \frac{5}{3}\beta^2$.*

Theorem 4. *The correlated price of anarchy of the average social cost is $\frac{5}{2}$.*

Proof. The lower bound is established in [2,4] (for pure equilibria). To establish the upper bound, let q be a correlated equilibrium and P be an optimal (or any other) allocation. The cost of player i at the correlated equilibrium is $c_i(q) = \sum_{s \in \mathcal{S}} q(s) c_i(s) = \sum_{s \in \mathcal{S}} q(s) \sum_{e \in s_i} f_e(n_e(s))$. We want to bound the expected social cost, the sum of the expected costs of the players: $\text{SUM}(q) = \sum_i c_i(q) = \sum_{s \in \mathcal{S}} q(s) \sum_{e \in E} n_e(s) f_e(n_e(s))$, with respect to the optimal cost $\text{SUM}(P) = \sum_i c_i(P) = \sum_{e \in E} n_e(P) f_e(n_e(P))$. At the correlated equilibrium

$$\begin{aligned} c_i(q) &= \sum_{s \in \mathcal{S}} q(s) \sum_{e \in s_i} f_e(n_e(s)) \leq \sum_{s \in \mathcal{S}} q(s) \sum_{e \in P_i} f_e(n_e(s_{-i}, P_i)) \\ &\leq \sum_{s \in \mathcal{S}} q(s) \sum_{e \in P_i} f_e(n_e(s) + 1) \end{aligned}$$

where (s_{-i}, P_i) is the usual notation in Game Theory to denote the allocation that results when we replace s_i by P_i . If we sum over all players i , we can bound the expected social cost as

$$\begin{aligned} \text{SUM}(q) &\leq \sum_{i \in N} \sum_{s \in \mathcal{S}} q(s) \sum_{e \in P_i} f_e(n_e(s) + 1) = \sum_{s \in \mathcal{S}} q(s) \sum_{e \in E} n_e(P) f_e(n_e(s) + 1) \\ &= \sum_{s \in \mathcal{S}} q(s) \sum_{e \in E} (a_e(n_e(P)(n_e(s) + 1) + b_e n_e(P)) \\ &\leq \sum_{s \in \mathcal{S}} q(s) \sum_{e \in E} (a_e(\frac{1}{3}n_e^2(s) + \frac{5}{3}n_e^2(P)) + b_e n_e(P)) \\ &= \frac{1}{3}\text{SUM}(q) + \frac{5}{3}\text{SUM}(P) \end{aligned}$$

where the second inequality follows from Lemma 2. □

For the symmetric games the correlated price of anarchy is also $5/2$. In fact, as the next theorem establishes, it is slightly less: $\frac{5N-2}{2N+1}$. This is tight, as a matching lower bound for pure Nash equilibria in [4] shows.

Theorem 5. *The average correlated price of anarchy of symmetric congestion games with linear cost functions is $\frac{5N-2}{2N+1}$.*

4.1 Asymmetric Weighted Games

In this subsection we assume that the social cost is the total latency and we even allow players to have weights. The main theorem of this subsection was first proved in [2] for mixed Nash equilibria. Here we generalize it to correlated equilibria. Our proof is shorter and in our opinion simpler, but it borrows a lot of ideas from [2]. We will need the following lemma:

Lemma 3. *For every non negative real α, β , it holds $\alpha\beta + \beta^2 \leq \frac{\sqrt{5}-1}{4}\alpha^2 + \frac{\sqrt{5}+5}{4}\beta^2$.*

Theorem 6. *For linear weighted congestion games, the correlated price of anarchy of the total latency is at most $\frac{3+\sqrt{5}}{2} \approx 2.618$.*

Proof. Let q be a correlated equilibrium and P be an optimal (or any other) allocation. The cost of player i at the correlated equilibrium is $c_i(q) = \sum_{s \in \mathcal{S}} q(s) c_i(s) = \sum_{s \in \mathcal{S}} q(s) \sum_{e \in s_i} f_e(\theta_e(s))$, where $\theta_e(s)$ is the total load on the facility e for the allocation s . We want to bound the expected total latency: $C(q) = E[\sum_{e \in E} l_e f_e(l_e)] = \sum_{s \in \mathcal{S}} q(s) \sum_{e \in E} \theta_e(s) f_e(\theta_e(s))$, where l_e is a random variable indicating the actual load on the facility e , with respect to the optimal cost $C(P) = \sum_{e \in E} \theta_e(P) f_e(\theta_e(P))$. At the correlated equilibrium

$$c_i(q) = \sum_{s \in \mathcal{S}} q(s) \sum_{e \in s_i} f_e(\theta_e(s)) \leq \sum_{s \in \mathcal{S}} q(s) \sum_{e \in P_i} f_e(\theta_e(s_{-i}, P_i)) \leq \sum_{s \in \mathcal{S}} q(s) \sum_{e \in P_i} f_e(\theta_e(s) + w_i)$$

where (s_{-i}, P_i) is the usual notation in Game Theory to denote the allocation that results when we replace s_i by P_i . If we multiply this inequality with w_i we get

$$\sum_{s \in \mathcal{S}} q(s) \sum_{e \in s_i} f_e(\theta_e(s)) w_i \leq \sum_{s \in \mathcal{S}} q(s) \sum_{e \in P_i} f_e(\theta_e(s) + w_i) w_i$$

If we sum over all players i , the left part of the inequality is the expected total latency $C(q)$, that we can bound, with the help of Lemma 3, as

$$\begin{aligned} C(q) &= \sum_{i \in N} \sum_{s \in \mathcal{S}} q(s) \sum_{e \in s_i} f_e(\theta_e(s)) w_i = \sum_{s \in \mathcal{S}} q(s) \sum_{e \in E} \theta_e(s) f_e(\theta_e(s)) \\ &\leq \sum_{i \in N} \sum_{s \in \mathcal{S}} q(s) \sum_{e \in P_i} f_e(\theta_e(s) + w_i) w_i \\ &= \sum_{i \in N} \sum_{s \in \mathcal{S}} q(s) \sum_{e \in P_i} (a_e(\theta_e(s) + w_i) + b_e) w_i \\ &\leq \sum_{s \in \mathcal{S}} q(s) \left(\sum_{e \in E} a_e(\theta_e(s) \theta_e(P) + \theta_e^2(P)) + \sum_{e \in E} b_e \theta_e(P) \right) \\ &\leq \sum_{s \in \mathcal{S}} q(s) \left(\sum_{e \in E} a_e \left(\frac{\sqrt{5}-1}{4} \theta_e^2(s) + \frac{\sqrt{5}+5}{4} \theta_e^2(P) \right) + \sum_{e \in E} b_e \theta_e(P) \right) \\ &\leq \frac{\sqrt{5}-1}{4} C(q) + \frac{\sqrt{5}+5}{4} C(P) \end{aligned}$$

and the theorem follows. \square

References

1. R. Aumann. Subjectivity and correlation in randomized games. *Journal of Mathematical Economics*, 1, pages 67-96, 1974.
2. B. Awerbuch, Y. Azar and A. Epstein. The Price of Routing Unsplittable Flow. *37th Annual ACM STOC*, pages 57-66, 2005.
3. E. Anshelevich, A. Dasgupta, J. Kleinberg, E. Tardos, T. Wexler and T. Roughgarden. The Price of Stability for Network Design with Fair Cost Allocation. In *45th Annual IEEE FOCS*, pages 59-73, 2004.
4. G. Christodoulou and E. Koutsoupias. The price of anarchy of finite congestion games. *37th Annual ACM STOC*, pages 67-73, 2005.

5. J. R. Correa, A. S. Schulz and N. S. Moses. Computational Complexity, Fairness, and the Price of Anarchy of the Maximum Latency Problem. In *Proceedings of the 10th Int. Conf. IPCO*, pages 59-73, 2004.
6. A. Czumaj, P. Krysta, B. Vöcking. Selfish traffic allocation for server farms. In *Proceedings on 34th Annual ACM STOC*, pages 287-296, 2002.
7. A. Czumaj and B. Vöcking. Tight Bounds for Worst-case Equilibria. In *Proceedings of the 13th Annual ACM-SIAM SODA*, pp. 413-420, January 2002.
8. A. Fabrikant, C. Papadimitriou, and K. Tulwar. On the complexity of pure equilibria. In *Proceedings of the 36th Annual ACM STOC*, pages 604-612, June 2004.
9. D. Fotakis, S. C. Kontogiannis and P. G. Spirakis. Selfish Unsplittable Flows. In *Proceedings of the 31st ICALP*, pages 593-605, 2004.
10. M. Gairing, T. Lücking, M. Mavronicolas and B. Monien. The Price of Anarchy for Polynomial Social Cost. In *Proceedings of the 29th MFCS*, pages 574-585, 2004.
11. M. Gairing, T. Lücking, M. Mavronicolas and B. Monien. Computing Nash equilibria for scheduling on restricted parallel links. In *Proceedings of the 36th Annual ACM STOC*, pages 613-622, 2004.
12. M. Gairing, T. Lücking, M. Mavronicolas, B. Monien and M. Rode. Nash Equilibria in Discrete Routing Games with Convex Latency Functions. In *Proceedings of the 31st ICALP*, pages 645-657, 2004.
13. E. Koutsoupias, M. Mavronicolas, and P. Spirakis. Approximate Equilibria and Ball Fusion. In *Proceedings of the 9th SIROCCO*, 2002
14. E. Koutsoupias and C. H. Papadimitriou. Worst-case equilibria. In *Proceedings of the 16th Annual STACS*, pages 404-413, 1999.
15. T. Lücking, M. Mavronicolas, B. Monien and M. Rode. A New Model for Selfish Routing. In *Proceedings of the 21st Annual STACS*, pages 547-558, 2004.
16. M. Mavronicolas and P. G. Spirakis. The price of selfish routing. In *Proceedings on 33rd Annual ACM STOC*, pages 510-519, 2001.
17. I. Milchtaich. Congestion Games with Player-Specific Payoff Functions. *Games and Economic Behavior* 13, pages 111-124, 1996.
18. D. Monderer and L. S. Shapley. Potential Games. *Games and Economic Behavior* 14, pages 124-143, 1996.
19. M. J. Osborne and A. Rubinstein. *A Course in Game Theory*. MIT Press, 1994.
20. C. H. Papadimitriou. Algorithms, games, and the Internet. In *Proceedings of the 33rd Annual ACM STOC*, pages 749-753, 2001.
21. C. H. Papadimitriou. Computing Correlated Equilibria in Multiplayer Games. *37th Annual ACM STOC*, pages 49-56, 2005.
22. C. H. Papadimitriou and T. Roughgarden. Computing Equilibria in Multi-Player Games. In *Proceedings of the 16th Annual ACM-SIAM SODA*, pages 82-91, 2005.
23. R. W. Rosenthal. A class of games possessing pure-strategy Nash equilibria. *International Journal of Game Theory*, 2:65-67, 1973.
24. T. Roughgarden. The price of anarchy is independent of the network topology. *Journal of Computer and System Sciences*, 67(2), pages 341-364, Sep. 2003.
25. T. Roughgarden. The maximum latency of selfish routing. In *Proceedings of the 15th Annual ACM-SIAM SODA*, pages 980-981, 2004.
26. T. Roughgarden and E. Tardos. How bad is selfish routing? *Journal of the ACM*, 49(2):236-259, 2002.
27. T. Roughgarden and E. Tardos. Bounding the inefficiency of equilibria in nonatomic congestion games. *Games and Economic Behavior*, 47(2):389-403, 2004.
28. S. Suri, C. D. Tóth and Y. Zhou. Selfish load balancing and atomic congestion games. In *Proceedings of the 16th annual ACM SPAA*, pages 188-195, 2004.

The Complexity of Games on Highly Regular Graphs

Konstantinos Daskalakis and Christos H. Papadimitriou*

UC Berkeley, Computer Science Division, Soda Hall, Berkeley, CA 94720
{costis, christos}@cs.berkeley.edu

Abstract. We present algorithms and complexity results for the problem of finding equilibria (mixed Nash equilibria, pure Nash equilibria and correlated equilibria) in games with extremely succinct description that are defined on highly regular graphs such as the d -dimensional grid; we argue that such games are of interest in the modelling of large systems of interacting agents. We show that mixed Nash equilibria can be found in time exponential in the succinct representation by quantifier elimination, while correlated equilibria can be found in polynomial time by taking advantage of the game's symmetries. Finally, the complexity of determining whether such a game on the d -dimensional grid has a pure Nash equilibrium depends on d and the dichotomy is remarkably sharp: it is solvable in polynomial time (in fact **NL**-complete) when $d = 1$, but it is **NEXP**-complete for $d \geq 2$.

1 Introduction

In recent years there has been some convergence of ideas and research goals between game theory and theoretical computer science, as both fields have tried to grapple with the realities of the Internet, a large system connecting optimizing agents. An important open problem identified in this area is that of computing a mixed Nash equilibrium; the complexity of even the 2-player case is, astonishingly, open (see, e.g., [9,15]). Since a mixed Nash equilibrium is always guaranteed to exist, ordinary completeness techniques do not come into play. The problem does fall into the realm of “exponential existence proofs” [11], albeit of a kind sufficiently specialized that, here too, no completeness results seem to be forthcoming. On the other hand, progress towards algorithms has been very slow (see, e.g., [10,7]).

We must mention here that this focus on complexity issues is not understood and welcome by all on the other side. Some economists are mystified by the obsession of our field with the complexity of a problem (Nash equilibrium) that arises in a context (rational behavior of agents) that is not computational at all. *We believe that complexity issues are of central importance in game theory*, and not just the result of professional bias by a few computer scientists. The reason is simple: Equilibria in games are important concepts of rational behavior

* Supported by NSF ITR Grant CCR-0121555 and by a Microsoft Research grant.

and social stability, reassuring existence theorems that enhance the explanatory power of game theory and justify its applicability. An intractability proof would render these existence theorems largely moot, and would cast serious doubt on the modelling power of games. How can one have faith in a model predicting that a group of agents will solve an intractable problem? In the words of Kamal Jain: “If your PC cannot find it, then neither can the market.”

However, since our ambition is to model by games the Internet and the electronic market, we must extend our complexity investigations well beyond 2-person games. This is happening: [2,4,5,12,10] investigate the complexity of multi-player games of different kinds. But there is an immediate difficulty: Since a game with n players and s strategies each needs ns^n numbers to be specified (see Section 2 for game-theoretic definitions) the input needed to define such a game is exponentially long. This presents with two issues: First, a host of tricky problems become easy just because the input is so large. More importantly, exponential input makes a mockery of claims of relevance: No important problem can need an astronomically large input to be specified (and we *are* interested in large n , and of course $s \geq 2$). Hence, all work in this area has focused on certain natural classes of *succinctly representable games*.

One important class of succinct games is that of the *graphical games* proposed and studied by Michael Kearns et al. [4,5]. In a graphical game, we are given a graph with the players as nodes. It is postulated that an agent’s utility depends on the strategy chosen by the player *and by the player’s neighbors in the graph*. Thus, such games played on graphs of bounded degree can be represented by polynomially many (in n and s) numbers. Graphical games are quite plausible and attractive as models of the interaction of agents across a large network or market. There has been a host of positive complexity results for this kind of games. It has been shown, for example, that correlated equilibria (a sophisticated equilibrium concept defined in Section 2) can be computed in polynomial time for graphical games that are trees [4], later extended to all graphical games [10].

But if we are to study truly large systems of thousands or millions of interacting agents, it is unrealistic to assume that we know the arbitrarily complex details of the underlying interaction graph and of the behavior of every single player — the size of such description would be forbidding anyway. One possibility, explored brilliantly in the work of Roughgarden and Tardos [14], is to assume a continuum of behaviorally identical players. In the present paper we explore an alternative model of large populations of users, within the realm of graphical games. Imagine that the interaction graph is perhaps the $n \times n$ grid, and that all players are locally identical (our results apply to many highly regular topologies of graphs and the case of several player classes). The representation of such a game would then be extremely succinct: Just the game played at each locus, and n , the size of the grid. *Such games, called highly regular graph games, are the focus of this paper*. For concreteness and economy of description, we mainly consider the homogeneous versions (without boundary phenomena) of the highly regular graphs (cycle in 1 dimension, torus in 2, and so on); however,

both our positive and negative results apply to the grid, as well as all reasonable generalizations and versions (see the discussion after Theorem 4).

We examine the complexity of three central equilibrium concepts: *pure Nash equilibrium*, *mixed Nash equilibrium* and the more general concept of *correlated equilibrium*. Pure Nash equilibrium may or may not exist in a game, but, when it does, it is typically much easier to compute than its randomized generalization (it is, after all, a simpler object easily identified by inspection). Remarkably, in highly regular graph games this is reversed: By a symmetry argument combined with quantifier elimination [1,13], we can compute a (succinct description of a) mixed Nash equilibrium in a d -dimensional highly regular graph game in exponential time (see theorem 2; recall that the best known algorithms for even 2-player Nash equilibria are exponential in the worst case). In contrast, regarding pure Nash equilibria, we establish an interesting dichotomy: The problem is polynomially solvable (and **NL**-complete) for $d = 1$ (the cycle) but becomes **NEXP**-complete for $d \geq 2$ (the torus and beyond). The algorithm for the cycle is based on a rather sophisticated analysis of the cycle structure of the Nash dynamics of the basic game. **NEXP**-completeness is established by a generic reduction which, while superficially quite reminiscent of the tiling problem [6], relies on several novel tricks for ensuring faithfulness of the simulation. Finally, our main algorithmic result states that a succinct description of a correlated equilibrium in a highly regular game of any dimension can be computed in polynomial time.

2 Definitions

In a *game* we have n players $1, \dots, n$. Each player p , $1 \leq p \leq n$, has a finite set of *strategies* or *choices*, S_p with $|S_p| \geq 2$. The set $S = \prod_{i=1}^n S_i$ is called *the set of strategy profiles* and we denote the set $\prod_{i \neq p} S_i$ by S_{-p} . The *utility* or *payoff* function of player p is a function $u_p : S \rightarrow \mathbb{N}$. The *best response function* of player p is a function $BR_{u_p} : S_{-p} \rightarrow 2^{S_p}$ defined by

$$BR_{u_p}(s_{-p}) \triangleq \{s_p \in S_p \mid \forall s'_p \in S_p : u_p(s_{-p}; s_p) \geq u_p(s_{-p}; s'_p)\}$$

that is, for every $s_{-p} \in S_{-p}$, $BR_{u_p}(s_{-p})$ is the set of all strategies s_p of player p that yield the maximum possible utility given that the other players play s_{-p} .

To specify a game with n players and s strategies each we need ns^n numbers, an amount of information exponential in the number of players. However, players often interact with a limited number of other players, and this allows for much more succinct representations:

Definition 1. *A graphical game is defined by:*

- A graph $G = (V, E)$ where $V = \{1, \dots, n\}$ is the set of players.
- For every player $p \in V$:
 - A non-empty finite set of strategies S_p
 - A payoff function $u_p : \prod_{i \in \mathcal{N}(p)} S_i \rightarrow \mathbb{N} \left(\mathcal{N}(p) \triangleq \{p\} \cup \{v \in V \mid (p, v) \in E\} \right)$

Graphical games can achieve considerable succinctness of representation. But if we are interested in modelling huge populations of players, we may need, and may be able to achieve, even greater economy of description. For example, it could be that the graph of the game is highly regular and that the games played at each neighborhood are identical. This can lead us to an extremely succinct representation of the game - logarithmic in the number of players. The following definition exemplifies these possibilities.

Definition 2. *A d -dimensional torus game is a graphical game with the following properties:*

- *The graph $G = (V, E)$ of the game is the d -dimensional torus:

 - $V = \{1, \dots, m\}^d$
 - $((i_1, \dots, i_d), (j_1, \dots, j_d)) \in E$ if there is a $k \leq d$ such that:

$$j_k = i_k \pm 1(\text{mod } m) \text{ and } j_r = i_r, \text{ for } r \neq k$$*
- *All the m^d players are identical in the sense that:

 - they have the same strategy set $\Sigma = \{1, \dots, s\}$
 - they have the same utility function $u : \Sigma^{2d+1} \rightarrow \mathbb{N}$*

Notice that a torus game with utilities bounded by u_{max} requires $s^{2d+1} \log |u_{max}| + \log m$ bits to be represented.

A torus game is *fully symmetric* if it has the additional property that the utility function u is symmetric with respect to the $2d$ neighbors of each node. Our negative results will hold even for this special case, while our positive results will apply to all torus games. We could also define torus games with *unequal sides* and *grid games*: torus games where the graph does not wrap around at the boundaries, and so $d + 1$ games must be specified, one for the nodes in the middle and one for each type of boundary node. Furthermore, there are the fully symmetric special cases for each. It turns out that very similar results would hold for all such kinds.

Consider a game G with n players and strategy sets S_1, \dots, S_n . For every strategy profile s , we denote by s_p the strategy of player p in this strategy profile and by s_{-p} the $(n - 1)$ -tuple of strategies of all players but p . For every $s'_p \in S_p$ and $s_{-p} \in S_{-p}$ we denote by $(s_{-p}; s'_p)$ the strategy profile in which player p plays s'_p and all the other players play according to s_{-p} . Also, we denote by $\Delta(A)$ the set of probability distributions over a set A and we'll call the set $\prod_{i=1}^n \Delta(S_i)$ *set of mixed strategy profiles* of the game G . For a mixed strategy profile σ and a mixed strategy σ'_p of player p , the notations σ_p, σ_{-p} and $(\sigma_{-p}; \sigma'_p)$ are analogous to the corresponding notations for the strategy profiles. Finally, by $\sigma(s)$ we'll denote the probability distribution in product form $\sigma_1(s_1)\sigma_2(s_2)\dots\sigma_n(s_n)$ that corresponds to the mixed strategy profile σ .

Definition 3. *A strategy profile s is a pure Nash equilibrium if for every player p and strategy $t_p \in S_p$ we have $u_p(s) \geq u_p(s_{-p}; t_p)$.*

Definition 4. A mixed strategy profile σ of a game $G = \langle n, \{S_p\}_{1 \leq p \leq n}, \{u_p\}_{1 \leq p \leq n} \rangle$ is a mixed Nash equilibrium if for every player p and for all mixed strategies $\sigma'_p \in \Delta(S_p)$ the following is true: $\mathbb{E}_{\sigma(s)}[u_p(s)] \geq \mathbb{E}_{(\sigma_{-p}, \sigma'_p)(s)}[u_p(s)]$, where by $\mathbb{E}_{f(s)}[u_p(s)]$, $f \in \Delta(S)$, we denote the expected value of the payoff function $u_p(s)$ under the distribution f .

Definition 5. A probability distribution $f \in \Delta(S)$ over the set of strategy profiles of a game G is a correlated equilibrium iff for every player p , $1 \leq p \leq n$, and for all $i, j \in S_p$, the following is true:

$$\sum_{s_{-p} \in S_{-p}} [u_p(s_{-p}; i) - u_p(s_{-p}; j)] f(s_{-p}; i) \geq 0$$

Every game has a mixed Nash Equilibrium [8] and, therefore, a correlated equilibrium, since, as can easily be checked, a mixed Nash equilibrium is a correlated equilibrium in product form. However, it may or may not have a pure Nash equilibrium.

The full description of an equilibrium of any kind in a torus game would require an exponential (doubly exponential in the correlated case) number of bits. Accordingly, our algorithms shall always output some kind of succinct representation of the equilibrium, from which one can generate the equilibrium in output polynomial time. In other words, a *succinct representation* of an equilibrium (or any other object) x is a string y such that $|y|$ is polynomial in the input size and $x = f(y)$ for some function f computable in time polynomial in $|x| + |y|$.

3 An Algorithm for Mixed Nash Equilibria

We start with a theorem due to Nash [8].

Definition 6. An automorphism of a game $G = \langle n, \{S_p\}, \{u_p\} \rangle$ is a permutation ϕ of the set $\bigcup_{p=1}^n S_p$ along with two induced permutations of the players ψ and of the strategy profiles χ , with the following properties:

- $\forall p, \forall x, y \in S_p$ there exists $p' = \psi(p)$ such that $\phi(x) \in S_{p'}$ and $\phi(y) \in S_{p'}$
- $\forall s \in S, \forall p : u_p(s) = u_{\psi(p)}(\chi(s))$

Definition 7. A mixed Nash equilibrium of a game is symmetric if it is invariant under all automorphisms of the game.

Theorem 1. [8] Every game has a symmetric mixed Nash equilibrium.

Now we can prove the following:

Theorem 2. For any $d \geq 1$, we can compute a succinct representation of a mixed Nash equilibrium of a d -dimensional torus game in time polynomial in $(2d)^s$, the size of the game description and the number of bits of precision required.

Proof. Suppose we are given a d -dimensional torus game $G = \langle m, \Sigma, u \rangle$ with $n = m^d$ players. By theorem 1, game G has a symmetric mixed Nash equilibrium σ . We claim that in σ all players play the same mixed strategy. Indeed for every pair of players p_1, p_2 in the torus, there is an automorphism (ϕ, ψ, χ) of the game such that $\psi(p_1) = p_2$ and ϕ maps the strategies of player p_1 to the same strategies of player p_2 . (In this automorphism, the permutation ψ is an appropriate d -dimensional cyclic shift of the players and permutation ϕ always maps strategies of one player to the same strategies of the player's image.) Thus in σ every player plays the same mixed strategy.

It follows that we can describe σ succinctly by giving the mixed strategy σ_x that every player plays. Let's suppose that $\Sigma = \{1, 2, \dots, s\}$. For all possible supports $T \subseteq 2^\Sigma$, we can check if there is a symmetric mixed Nash equilibrium σ with support T^n as follows. Without loss of generality let's suppose that $T = \{1, 2, \dots, j\}$ for some $j, j \leq s$. We shall construct a system of polynomial equations and inequalities with variables p_1, p_2, \dots, p_j , the probabilities of the strategies in the support.

Let us call E_l the expected payoff of an arbitrary player p if s/he chooses the pure strategy l and every other player plays σ_x . E_l is a polynomial of degree $2d$ in the variables p_1, p_2, \dots, p_j . Now σ_x is a mixed Nash equilibrium of the game if and only if the following conditions hold (because of the symmetry, if they hold for one player they hold for every player of the torus):

$$\begin{aligned} E_l &= E_{l+1}, \forall l \in \{1, \dots, j-1\} \\ E_j &\geq E_l, \forall l \in \{j+1, \dots, s\} \end{aligned}$$

We need to solve s simultaneous polynomial equations and inequalities of degree $2d$ in $O(s)$ variables. It is known [13] that this problem can be solved in time polynomial in $(2d)^s$, the number of bits of the numbers in the input and the number of bits of precision required. Since the number of bits required to define the system of equations and inequalities is polynomial in the size of the description of the utility function, we get an algorithm polynomial in $(2d)^s$, the size of the game description and the number of bits of precision required. \square

4 A Polynomial Algorithm for Correlated Equilibria

Theorem 3. *Given a torus game G , we can compute a succinct representation of a correlated equilibrium in time polynomial in the description of the game.*

Proof. Suppose d is the dimension of the torus on which the game is defined and m is its size. We will compute a function in the neighborhood of an arbitrary player p and then show how this function can be extended to a correlated equilibrium of the game in output polynomial time. *The construction is easier when m is a multiple of $2d+1$, and thus we shall assume first that this is the case.* We rewrite the defining inequalities of a correlated equilibrium as follows:

$$\forall i, j : \sum_{s_{neigh} \in \Sigma^{2d}} [u(s_{neigh}; i) - u(s_{neigh}; j)] \sum_{s_{oth} \in \Sigma^{m^d - 2d - 1}} f(s_{oth}; s_{neigh}; i) \geq 0 \quad (1)$$

$$\Leftrightarrow \forall i, j : \sum_{s_{neigh} \in \Sigma^{2d}} [u(s_{neigh}; i) - u(s_{neigh}; j)] f_p(s_{neigh}; i) \geq 0 \quad (2)$$

where f_p is the marginal distribution corresponding to player p and its $2d$ neighbors. Easily, we can construct a linear program that finds a non-negative function f_p satisfying inequalities (2). To ensure that the linear program will return a non-identically zero solution, if such a solution exists, we require all variables to be in $[0, 1]$ and the objective function tries to maximize their sum. Also, let us for now add $O(s^{2d+1} \cdot (2d+1)!)$ symmetry constraints to the linear program so that the function defined by its solution is symmetric with respect to its arguments. It will be clear later that $O(s^{2d+1} \cdot 2d)$ constraints are enough.

We proved in section 3 that G has a mixed Nash equilibrium in which all players play the same mixed strategy and, thus, a correlated equilibrium in product form and symmetric with respect to all the players. Therefore, our linear program has at least one non-zero feasible solution which (after normalization) defines a probability distribution $g(s_0, s_1, \dots, s_{2d})$ over the set Σ^{2d+1} which is symmetric with respect to its arguments. We argue that every such distribution can be extended by an output polynomial algorithm to a correlated equilibrium for the game G , **provided m is a multiple of $2d+1$** . We, actually, only need to show that we can construct a probability distribution $f \in \Delta(\Sigma^{m^d})$ with the property that the marginal distribution of the neighborhood of every player is equal to the probability distribution g . Then, by the definition of g , inequalities (1) will hold and, thus, f will be a correlated equilibrium of the game.

We first give the intuition behind the construction: g can be seen as the joint probability of $2d+1$ random variables X_0, X_1, \dots, X_{2d} ; if we can “tile” the d -dimensional torus with the random variables X_0, X_1, \dots, X_{2d} in such a way that all the $2d+1$ random variables appear in the neighborhood of every player, then we can define probability distribution f as the probability distribution that first draws a sample $(l_0, l_1, \dots, l_{2d})$ according to g and then assigns strategy l_i to all players that are tiled with X_i for all i . This way the marginal distribution of f in every neighborhood will be the same as g (since g is symmetric).

We now show how this can be done in a systematic way. Let us fix an arbitrary player of the torus as the origin and assign orientation to each dimension, so that we can label each player x with a name $(x_1, x_2, \dots, x_d) \in \{1, \dots, m\}^d$. The configurations of strategies in the support of f will be in a one-to-one correspondence with the configurations in the support of g . Specifically, for every configuration $s = (l_0^s, l_1^s, \dots, l_{2d}^s)$ in the support of g , we include in the support of f the configuration in which every player (x_1, x_2, \dots, x_d) plays strategy $l_{(x_1+2x_2+3x_3+\dots+dx_d \bmod 2d+1)}^s$. So we define the support of f to be:

$$\mathcal{S}_f = \{s \in \Sigma^{m^d} \mid \exists (l_0^s, l_1^s, \dots, l_{2d}^s) \in \mathcal{S}_g \text{ s.t. } s_x = l_{(x_1+2x_2+3x_3+\dots+dx_d \bmod 2d+1)}^s\}$$

and the distribution itself to be $f(s) = g(l_0^s, l_1^s, \dots, l_{2d}^s)$, if $s \in \mathcal{S}_f$, and 0 otherwise.¹ From the definition of f it follows easily that the marginal distribution of every player's neighborhood is equal to g . This completes the proof. Additionally, we note that for our construction to work we don't need g to be fully symmetric. We only need it to be equal to $O(2d)$ out of the total number of $O((2d+1)!)^2$ functions that result by permuting its arguments. So the number of symmetry constraints we added to the linear program can be reduced to $O(s^{2d+1} \cdot 2d)$.

To generalize this result to the case in which m is not a multiple of $2d+1$, let us first reflect on the reason why the above technique fails in the case where m is not a multiple of $2d+1$. If m is not a multiple of $2d+1$, then it is not hard to see that, given an arbitrary probability distribution g defined in the neighborhood of one player, we cannot always find a probability distribution on the torus so that its marginal in the neighborhood of every player is the same as g , even if g is symmetric. Thus, instead of starting of with the computation of a probability distribution in the neighborhood of one player, we compute a probability distribution h with an augmented number of arguments. Let's call $v = m \bmod 2d+1$. Probability distribution h will have the following properties:

- it will have $2 \times (2d+1)$ arguments
- it will be symmetric with respect to its arguments (we'll relax this requirement later in the proof to get fewer symmetry constraints for our linear program)
- the marginal distribution f_p of the first $2d+1$ arguments will satisfy inequalities (2); then, because of the symmetry of the function, the marginal of every ordered subset of $2d+1$ arguments will satisfy inequalities (2)

Again, the existence of a probability distribution h with the above properties follows from the existence of a symmetric mixed Nash equilibrium in which all players play the same mixed strategy. Moreover, such a distribution can be found in polynomial time by solving the linear program that corresponds to the above constraints. To conclude the proof we show how we can use h to produce a correlated equilibrium in output polynomial time. Before doing so, we make the following observations (again let us consider an arbitrary player as the origin and assign orientation to each dimension so that every player x has a name $(x_1, x_2, \dots, x_d) \in \{1, \dots, m\}^d$):

- Probability distribution h can be seen as the joint probability distribution of $2 \times (2d+1)$ random variables $X_0, X_1, \dots, X_{2d}, Z_0, Z_1, \dots, Z_{2d}$. If we can “tile” the d -dimensional torus with these random variables in such a way that the neighborhood of every player contains $2d+1$ distinct random variables, then this tiling implies a probability distribution on the torus with the property that the marginal of every player's neighborhood is equal to f_p and thus is a correlated equilibrium.

¹ In terms of tiling the d -dimensional torus with the random variables X_0, X_1, \dots, X_{2d} , our construction assigns variable X_0 to the origin player and then assigns variables periodically with step 1 in the first dimension, step 2 in the second dimension etc.

- Given $2d + 1$ random variables, we can use the tiling scheme described above to tile **any d -dimensional grid** in such a way that every neighborhood of size i has i distinct random variables. However, if we “fold” the grid to form the torus this property might not hold if m is not a multiple of $2d + 1$; there might be player with at least one coordinate equal to 1 or m (who before was at the boundary of the grid) whose neighborhood does not have $2d + 1$ distinct random variables.

Following this line of thought, if we can partition the players of the d -dimensional torus in disjoint d -dimensional grids and we tile every grid with a set of random variables so that every two neighboring grids are tiled with disjoint sets of random variables, then we automatically define a tiling with the required properties. To do so, we partition the d -dimensional torus in 2^d grids Γ_t , $t \in \{0, 1\}^d$, where grid Γ_t is the subgraph of the d -dimensional torus induced by players with names (x_1, x_2, \dots, x_d) , where $x_i \in \{1, 2, \dots, m - (2d + 1 + v)\}$ if the i -th bit of t is 0 and $x_i \in \{m - (2d + 1 + v) + 1, \dots, m\}$ if the i -th bit of t is 1. For every grid Γ_t , let's call $n(\Gamma_t)$ the number of 1's in t . The following observation is key to finishing the proof:

Lemma 1. *If two grids Γ_t , $\Gamma_{t'}$ are neighboring then $n(\Gamma_t) = n(\Gamma_{t'}) + 1$ or $n(\Gamma_t) = n(\Gamma_{t'}) - 1$.*

Using lemma 1, we can tile the d -dimensional torus as follows: if a grid Γ_t has even $n(\Gamma_t)$ then use random variables X_0, X_1, \dots, X_{2d} and the tiling scheme described above to tile it; otherwise use random variables Z_0, Z_1, \dots, Z_{2d} to tile it. This completes the proof. Again, note that we only need h to be equal to $O(d)$ out of the total number of $O((2 \times (2d + 1))!)$ functions that result by permuting its arguments; so we need a polynomial in the game complexity number of symmetry constraints. \square

5 The Complexity of Pure Nash Equilibria

We show our dichotomy result: Telling whether a d -dimensional torus game has a pure Nash equilibrium is **NL**-complete if $d = 1$ and **NEXP**-complete if $d > 1$.

5.1 The Ring

Theorem 4. *Given a 1-dimensional torus game we can check whether the game has a pure Nash equilibrium in polynomial time; in fact the problem is nondeterministic logarithmic space complete.*

Proof. Given such a game we construct a directed graph $T = (V_T, E_T)$ as follows.

$$V_T = \{(x, y, z) \mid x, y, z \in \Sigma : y \in \text{BR}_u(x, z)\}$$

$$E_T = \{(v_1, v_2) \mid v_1, v_2 \in V_T : v_{1y} = v_{2x} \wedge v_{1z} = v_{2y}\}$$

The construction of graph T can be done in time polynomial in s and $|V_T| = O(s^3)$. Thus, the adjacency matrix A_T of the graph has $O(s^3)$ rows and columns. Also, due to the construction it is easy to see that the following is true.

Lemma 2. *The input game has a pure Nash equilibrium iff there is a closed walk of length m in T .*

Checking whether there exists a closed walk of length m in graph T can easily be done in polynomial time, for example by finding A_T^m using repeated squaring.

We briefly sketch why it can be done in nondeterministic logarithmic space. There are two cases: If m is at most polynomial in s (and, thus, in the size of T), then we can guess the closed walk in logarithmic space, counting up to m . The difficulty is when m is superpolynomial in s , and we must guess the closed walk of length m in space $\log \log m$, that is, without counting up to m . We first establish that every such walk can be decomposed into a short closed walk of length $q \leq s^6$, plus a set of cycles, all connected to this walk, and of lengths c_1, \dots, c_r (each cycle repeated a possibly exponential number of times) such that the greatest common divisor of c_1, \dots, c_r divides $m - q$. We therefore guess the short walk of T of length q , guess the r cycles, compute the greatest common divisor of their lengths $g = \gcd(c_1, \dots, c_r)$ in logarithmic space, and then check whether g divides $m - q$; this latter can be done by space-efficient long division, in space $\log g$ and without writing down $m - q$. Finally, **NL**-completeness is shown by a rather complicated reduction from the problem of telling whether a directed graph has an odd cycle, which can be easily shown to be **NL**-complete. \square

The same result holds when the underlying graph is the 1-dimensional grid (the path) with some modifications to the proof. We, also, note that the problem has the same complexity for several generalizations of the path topology such as the *ladder graph* and many others.

5.2 The Torus

The problem becomes **NEXP**-complete when $d > 1$, even in the fully symmetric case. The proof is by a generic reduction.

Theorem 5. *For any $d \geq 2$, deciding whether there exists a pure Nash equilibrium in a fully symmetric d -dimensional torus game is **NEXP**-complete.*

Proof. A non-deterministic exponential time algorithm can choose in $O(m^d)$ nondeterministic steps a pure strategy for each player, and then check the equilibrium conditions for each player. Thus, the problem belongs to the class **NEXP**.

Our **NEXP**-hardness reduction is from the problem of deciding, given a one-tape nondeterministic Turing machine M and an integer t , whether there is a computation of M that halts within $5t - 2$ steps. We present the $d = 2$ case, the generalization to $d > 2$ being trivial. Given such M and t , we construct the torus game $G_{M, t}$ with size $m = 5t + 1$. Intuitively, the strategies of the players will correspond to states and tape symbols of M , so that a Nash equilibrium will spell a halting computation of M in the “tableau” format (rows are steps and columns are tape squares). In this sense, the reduction is similar to that showing completeness of the *tiling* problem: Can one tile the $m \times m$ square by square tiles of unit side and belonging to certain types, when each side of each type comes with a label restricting the types that can be used next to it? Indeed,

each strategy will simulate a tile having on the horizontal sides labels of the form (*symbol, state*) whereas on the vertical sides (*state, action*). Furthermore, each strategy will also be identified by a pair (i, j) of integers in $\{0, 1, \dots, 4\}$, standing for the coordinates modulo 5 of the node that plays this strategy; the necessity of this, as well as the choice of 5, will become more clear later in the proof. Superficially, the reduction now seems straightforward: Have a strategy for each tile type, and make sure that the best response function of the players reflects the compatibility of the tile types. There are, however, several important difficulties in doing this, and we summarize the principal ones below.

Difficulty 1. In tiling, the compatibility relation can be partial, in that no tile fits at a place when the neighbors are tiled inappropriately. In contrast, in our problem the best response function must be total.

Difficulty 2. Moreover, since we are reducing to fully symmetric games, the utility function must be symmetric with respect to the strategies of the neighbors. However, even if we suppose that for a given 4-tuple of tiles (strategies) for the neighbors of a player there is a matching tile (strategy) for this player, that tile does not necessarily match every possible permutation-assignment of the given 4-tuple of tiles to the neighbors. To put it otherwise, symmetry causes a *lack of orientation*, making the players unable to distinguish among their ‘up’, ‘down’, ‘left’ and ‘right’ neighbors.

Difficulty 3. The third obstacle is the *lack of boundaries* in the torus which makes it difficult to define the strategy set and the utility function in such a way as to ensure that some “bottom” row will get tiles that describe the initial configuration of the Turing machine and the computation tableau will get built on top of that row.

It is these difficulties that require our reduction to resort to certain novel stratagems and make the proof rather complicated. Briefly, we state here the essence of the tricks and we refer the reader to the full report [16] for further details:

Solution 1. To overcome the first difficulty, we introduce three special strategies and we define our utility function in such a way that (a) these strategies are the best responses when we have no tile to match the strategies of the neighbors and (b) no equilibria of the game can contain any of these strategies.

Solution 2. To overcome the second difficulty we attach coordinates modulo 5 to all of the tiles that correspond to the interior of the computation tableau and we define the utility function in such a way that in every pure Nash equilibrium a player who plays a strategy with coordinates modulo 5 equal to (i, j) has a neighbor who plays a strategy with each of the coordinates $(i \pm 1, j \pm 1 \pmod{5})$. This implies (through a nontrivial graph-theoretic argument) that the torus is “tiled” by strategies respecting the counting modulo 5 in both dimensions.

Solution 3. To overcome difficulty 3, we define the side of the torus to be $5t + 1$ and we introduce strategies that correspond to the boundaries of the computation tableau and are best responses only in the case their neighbors’

coordinates are not compatible with the counting modulo 5. The choice of side length makes it impossible to tile the torus without using these strategies and, thus, ensures that one row and one column (at least) will get strategies that correspond to the boundaries of the computation tableau. \square

6 Discussion

We have classified satisfactorily the complexity of computing equilibria of the principal kinds in highly regular graph games. We believe that the investigation of computational problems on succinct games, of which the present paper as well as [3] and [12,10] are examples, will be an active area of research in the future. One particularly interesting research direction is to formulate and investigate games on highly regular graphs in which the players' payoffs depend in a natural and implicit way on the players' location on the graph, possibly in a manner reflecting routing congestion, facility location, etc., in a spirit similar to that of non-atomic games [14].

References

1. G. E. Collins "Quantifier elimination for real closed fields by cylindrical algebraic decomposition," *Springer Lecture Notes in Computer Science*, 33, 1975.
2. A. Fabrikant, C. H. Papadimitriou, K. Talwar "The Complexity of Pure Nash Equilibria," *STOC*, 2004.
3. L. Fortnow, R. Impagliazzo, V. Kabanets, C. Umans "On the complexity of succinct zero-sum games," *IEEE Conference on Computational Complexity*, 2005.
4. S. Kakade, M. Kearns, J. Langford, and L. Ortiz "Correlated Equilibria in Graphical Games," *ACM Conference on Electronic Commerce*, 2003.
5. M. Kearns, M. Littman, S. Singh "Graphical Models for Game Theory," *UAI*, 2001.
6. H. Lewis, C. H. Papadimitriou "Elements of the Theory of Computation," *Prentice-Hall*, 1981.
7. R. J. Lipton, E. Markakis "Nash Equilibria via Polynomial Equations," *LATIN*, 2004.
8. J. Nash "Noncooperative games," *Annals of Mathematics*, 54, 289–295, 1951.
9. C. H. Papadimitriou "Algorithms, Games, and the Internet," *STOC*, 2001.
10. C. H. Papadimitriou "Computing correlated equilibria in multiplayer games," *STOC*, 2005.
11. C. H. Papadimitriou "On the Complexity of the Parity Argument and Other Inefficient Proofs of Existence," *J. Comput. Syst. Sci.*, 48(3), 1994.
12. C. H. Papadimitriou, T. Roughgarden "Computing equilibria in multiplayer games," *SODA*, 2005.
13. J. Renegar "On the Computational Complexity and Geometry of the First-Order Theory of the Reals, I, II, III," *J. Symb. Comput.*, 13(3), 1992.
14. T. Roughgarden, E. Tardos "How bad is selfish routing?," *J. ACM*, 49(2), 2002.
15. R. Savani, B. von Stengel "Exponentially many steps for finding a Nash equilibrium in a bimatrix game," *FOCS*, 2004.
16. <http://www.eecs.berkeley.edu/~costis/RegularGames.pdf>

Computing Equilibrium Prices: Does Theory Meet Practice?

Bruno Codenotti¹, Benton McCune², Rajiv Raman², and Kasturi Varadarajan²

¹ Toyota Technological Institute at Chicago, Chicago IL 60637
bcodenotti@tti-c.org

² Department of Computer Science, The University of Iowa,
Iowa City IA 52242
{bmccune, rraman, kvaradar}@cs.uiowa.edu

Abstract. The best known algorithms for the computation of market equilibria, in a general setting, are not guaranteed to run in polynomial time. On the other hand, simple poly-time algorithms are available for various restricted - yet important - markets.

In this paper, we experimentally explore the *gray zone* between the general problem and the poly-time solvable special cases. More precisely, we analyze the performance of some simple algorithms, for inputs which are relevant in practice, and where the theory does not provide poly-time guarantees.

1 Introduction

The market equilibrium problem (in its exchange version) consists of finding prices and allocations (of goods to traders) such that each trader maximizes her utility function and the market clears (see Section 2 for precise definitions). A fundamental result in economic theory states that, under mild assumptions, market clearing prices exist [1].

Soon after this existential result was shown, researchers started analyzing economic processes leading to the equilibrium. The most popular of them takes the name of tâtonnement, and consists of updating the current price of each good based on its excess demand. Unfortunately, the tâtonnement process, in its differential version, only converges under restrictive assumptions, such as *gross substitutability* [2].

The failure of tâtonnement to provide global convergence stimulated a substantial amount of work on the computation of the equilibrium. Scarf and some coauthors developed pivotal algorithms which search for an equilibrium within the simplex of prices [25,12]. Unlike tâtonnement, these algorithms always reach a solution, although they lack a clear economic interpretation and they can be easily seen to require exponential time, even on certain simple instances.

Motivated by the lack of global convergence of tâtonnement, and by the lack of a clear economic interpretation for Scarf's methods, Smale developed a global

* The first author is on leave from IIT-CNR, Pisa, Italy. Work by the second and fourth authors was supported by NSF CAREER award CCR-0237431.

Newton's method for the computation of equilibrium prices [28]. His approach provides a price adjustment mechanism which takes into account all the components of the Jacobian of the excess demand functions. However, Smale's technique does not come with polynomial time guarantees, and its behavior, when the current price is far from equilibrium, does not seem to be analyzable. For this reason, most solvers based on Newton's method, including PATH, the solver used in Section 4.2 and which is available under the popular GAMS framework, do a line search within each Newton's iteration, in order to guarantee that some progress is made even far from equilibrium (see [14]).

This state of affairs motivated theoretical computer scientists to investigate whether the problem could be solved in polynomial time. The computational complexity of the general market equilibrium problem turns out to be tightly connected to that of finding fixed points in quite arbitrary settings [23]. So polynomial time algorithms for the general case would have a wide impact on several related computational issues, including that of finding a Nash equilibrium for a two person nonzero sum game, and it is unclear whether they exist. Therefore, the current theoretical work has been devoted to isolating restrictive yet important families of markets for which the problem can be solved in polynomial time [10,18,5,8,9,30]. The most important restrictions for which polynomial time algorithms have been discovered arise either when the traders have utility functions that satisfy a property known as *gross substitutability* or when the aggregate demand satisfies a property known as the *weak axiom of revealed preferences* (see [6] for a review).

In this paper we continue the work started in [7], and experimentally analyze and compare several approaches, evaluating their performance on market settings that are significantly more versatile and flexible than those used in [7].

Our main findings are the following -

1. For certain choices of market types, the PATH solver exhibits a large variance in performance and often fails to converge when applied to exchange economies with nested CES functions; this phenomenon is in sharp contrast with what is observed in [7] for CES functions. Still, the solver does a reasonably good job for most market types. When production is added to the mix, however, it becomes relatively easier to find market types where PATH exhibits poor performance.
2. The tâtonnement algorithm also often fails to converge for certain market types with nested CES functions, although it generally converges for most market types. This is another setting where the transition from CES to nested CES functions is a significant one: indeed it was shown in [7] that tâtonnement is generally convergent on exchange economies with CES functions.
3. The welfare adjustment process is almost always convergent on CES exchange economies, where the proportionality of initial endowments is slightly distorted by taxation. Even when the initial endowments are farthest from being proportional, the process converges, although with a significantly larger number of iterations, for all but a few market types.

2 Market Models, Definitions, and Background

Let us consider m economic agents which represent traders of n goods. Let \mathbf{R}_+^n denote the subset of \mathbf{R}^n with all nonnegative coordinates. The j -th coordinate in \mathbf{R}^n will stand for good j . Each trader i has a concave utility function $u_i : \mathbf{R}_+^n \rightarrow \mathbf{R}_+$, which represents her preferences for different bundles of goods, and an initial endowment of goods $w_i = (w_{i1}, \dots, w_{in}) \in \mathbf{R}_+^n$. At given prices $\pi \in \mathbf{R}_+^n$, trader i will sell her endowment, and get the bundle of goods $x_i = (x_{i1}, \dots, x_{in}) \in \mathbf{R}_+^n$ which maximizes $u_i(x)$ subject to the budget constraint $\pi \cdot x \leq \pi \cdot w_i$, where $x \cdot y$ denotes the inner product between x and y . Let $W_j = \sum_i w_{ij}$ denote the total amount of good j in the market.

An equilibrium is a vector of prices $\pi = (\pi_1, \dots, \pi_n) \in \mathbf{R}_+^n$ at which there is a bundle $\bar{x}_i = (\bar{x}_{i1}, \dots, \bar{x}_{in}) \in \mathbf{R}_+^n$ of goods for each trader i such that the following two conditions hold: (i) For each good j , $\sum_i \bar{x}_{ij} \leq W_j$ and (ii) For each trader i , the vector \bar{x}_i maximizes $u_i(x)$ subject to the constraints $\pi \cdot x \leq \pi \cdot w_i$ and $x \in \mathbf{R}_+^n$.

The celebrated result of Arrow and Debreu [1] states that, under mild assumptions, such an equilibrium exists. A special case occurs when the initial endowments are *collinear*, i.e., when $w_i = \delta_i w$, $\delta_i > 0$, so that the relative incomes of the traders are independent of the prices. This special case is equivalent to the *Fisher model*, which is a market of n goods desired by m utility maximizing buyers with fixed incomes.

A utility function $u(\cdot)$ is *homogeneous* (of degree one) if it satisfies $u(\alpha x) = \alpha u(x)$, for all $\alpha > 0$. A *homothetic* utility function is a monotonic transformation of a homogeneous utility function. A homothetic utility function represents consumers' preferences of the following kind: a bundle x is preferred to a bundle y if and only if the bundle αx is preferred to the bundle αy , for all $\alpha > 0$.

A *CES (constant elasticity of substitution)* utility function is a homogeneous function of the form $u(x_i) = (\sum_j (a_{ij} x_{ij})^\rho)^{1/\rho}$, where $-\infty < \rho < 1$ but $\rho \neq 0$. In all of these definitions, $a_{ij} \geq 0$. The parameter $\sigma = 1/(1 - \rho)$ is called the elasticity of substitution of the utility function u .

As ρ tends to 1 (resp. 0, $-\infty$), the CES utility function tends to a linear (resp. Cobb-Douglas, Leontief) utility function ([3], page 231).

A nested CES function from $\mathbf{R}_+^n \rightarrow R$ is defined recursively: (1) Any CES function is a nested CES function; and (2) if $g : \mathbf{R}_+^t \rightarrow R$ is a t -variate CES function, and h_1, \dots, h_t are n -variate nested CES functions, then $f = g(h_1(), \dots, h_t())$ is a nested CES function. A nested CES function may be visualized using a tree-like structure, where at each node of the tree we have a CES function.

Nested CES functions are used extensively to model both production and consumption in applied general equilibrium: We refer the reader to the book by Shoven and Whalley [27] for a sense of their pervasiveness.

3 Theory vs Practice

Theoretical work on the market equilibrium problem has shown that the computational tools needed to find a market equilibrium must have the ability to

compute fixed points. However there are simple techniques (as well as polynomial time algorithms) which can provably solve some special instances. In practice, the range of applicability of these techniques might be significantly larger than what theory guarantees. In the words of Herbert Scarf, *it might be a prudent strategy to try a simple technique like the price adjustment mechanism even though this is not guaranteed to converge with certainty* [26].

We now show how to put Scarf’s words to work, and describe some scenarios to which our experiments are tailored.

How Common Are Hard Instances? The celebrated SMD Theorem (see [21], pp. 598-606) states the essentially arbitrary nature of the market excess demand. In the light of this result, neither the identification of instability phenomena [24], nor that of simple economies with multiple equilibria [16] come as surprises. On the other hand, it is relevant to understand how common these phenomena are, and when they represent intrinsic computational hurdles as opposed to barriers to our current understanding of the problem.

In this paper, we conduct our experiments using nested CES functions. The transition from CES (used in [7]) to nested CES functions allows us to model more elaborate consumer behavior. We will also see that these functions make it much easier to generate *hard instances*, for which, e.g., the PATH solver exhibits significant variations in computational performance.

Welfare Adjustment Schemes. A family of computational techniques follows from Negishi’s characterization of the market equilibrium as the solution to a welfare maximization problem, where the *welfare function* is a positive linear combination of the individual utilities [22].

Let $\alpha = (\alpha_1, \dots, \alpha_m) \in \mathfrak{R}^m$, where $\alpha_i > 0$, be any vector. Consider the allocations that maximize, over $x_i \in \mathbf{R}_+^n$, the function $\sum_{i=1}^m \alpha_i u_i(x_i)$ subject to the constraint that $\sum_i x_{ij} \leq \sum_i w_{ij}$.

The optimal allocations \bar{x}_i are called the *Negishi welfare optimum* at the *welfare weights* α_i . Let $\pi = (\pi_1, \dots, \pi_n) \in \mathbf{R}_+^n$, where the “dual price” π_j is the Lagrangian multiplier associated with the constraint in the program corresponding to the j -th good. Define $B_i(\alpha) = \pi \cdot w_i - \pi \cdot \bar{x}_i$, the budget surplus of the i -th trader at prices π and with allocation \bar{x}_i . Define $f_i(\alpha) = B_i(\alpha)/\alpha_i$, and $f(\alpha) = (f_1(\alpha), \dots, f_m(\alpha))$. Negishi [22] shows that there is a vector α^* such that $f(\alpha^*) = 0$ and the corresponding dual prices constitute an equilibrium for the economy.

This characterization suggests an approach for finding an equilibrium by a search in the space of *Negishi weights*. This approach, which is complementary to the traditional price space search, is elaborated by Mantel in [20], where the author shows that if the utility functions are strictly concave and log-homogeneous, and generate an excess demand that satisfies gross substitutability, then we have $\frac{\partial f_i(\alpha)}{\partial \alpha_i} < 0$ and $\frac{\partial f_j(\alpha)}{\partial \alpha_i} > 0$ for $j \neq i$. This is the analog of gross substitutability in the “Negishi space.” He also shows that a differential welfare-weight adjustment process, which is the equivalent of tâtonnement, converges to the equilibrium in these situations.

The Distortion Induced by Taxation. The transition from equilibrium theory to applied equilibrium requires certain distortions to be taken into account. Such distortions include the presence of transaction costs, transportation costs, tariffs, and taxes; these elements affect the equilibrium conditions and sometimes change some mathematical and economic properties of the problem. For instance, in models with taxes and tariffs one typically loses the Pareto optimality of the equilibrium allocations.

We now consider the effect that *ad valorem taxes* can have on certain exchange economies, and in Section 4.4 we will analyze some related computational consequences.

The stage for an exchange economy with ad valorem taxes is the following (see [19], pp. 2127-2128). Let $\tau_j \geq 0$ be the tax associated with good j , and let $\theta_i \geq 0$ with $\sum_i \theta_i = 1$ be given.

The demand of the i -th consumer problem is to maximize $u_i(x_i)$ subject to the budget constraint $\sum_j \pi_j(1 + \tau_j)x_{ij} \leq \sum_j \pi_j w_{ij} + \theta_i R$ and $x \in \mathbf{R}_+^n$, where $\theta_i R$ is a lump sum returned to consumer i after the collection of taxes.

The equilibrium conditions involve, in addition to the maximization above, market clearance (for each good j , $\sum_i \bar{x}_{ij} = \sum_i w_{ij}$), and the requirement that R should equal the amount collected from the taxation, i.e.,

$$R = \sum_j \pi_j \tau_j \sum_i x_{ij}.$$

We now show that an equilibria for such an economy are in a one-to-one correspondence to equilibria in an exchange economy without taxes, where the traders have a different set of initial endowments.

From market clearance it follows that the tax revenue at equilibrium must be $R = \sum_j \pi_j \tau_j \sum_i x_{ij} = \sum_j \pi_j \tau_j \sum_i w_{ij}$. We can therefore eliminate the requirement $R = \sum_j \pi_j \tau_j \sum_i x_{ij}$ by plugging in $R = \sum_j \pi_j \tau_j \sum_i w_{ij}$ in the budget constraint of each trader. The right hand side of the budget constraint for the i 'th trader is then $\sum_j \pi_j (w_{ij} + \theta_i \tau_j \sum_i w_{ij})$.

After dividing and multiplying each term of this expression by $1 + \tau_j$, the budget constraint can be rewritten as

$$\sum_j \pi_j (1 + \tau_j) x_{ij} \leq \sum_j \pi_j (1 + \tau_j) w'_{ij}, \text{ where } w'_{ij} = \frac{w_{ij}}{1 + \tau_j} + \theta_i \frac{\tau_j}{1 + \tau_j} \sum_i w_{ij}.$$

Note that $\sum_i w'_{ij} = \sum_i w_{ij}$ for each j . We have therefore reduced the equilibrium in the original economy to an equilibrium in a pure exchange economy, where the i -th trader now has an initial endowment w'_i : (π_1, \dots, π_n) is an equilibrium for the original economy if and only if $((1 + \tau_1)\pi_1, \dots, (1 + \tau_n)\pi_n)$ is an equilibrium for the pure exchange economy.

Therefore an exchange economy with ad valorem taxes E is equivalent to an exchange economy without taxes E' where the initial endowments have been redistributed, and the total amounts of goods in E' is the same as in E .

One interesting case where this phenomenon carries negative computational consequences is that of exchange economies with collinear endowments and ho-

mothetic preferences. In this setting an equilibrium can be computed in polynomial time by convex programming [8], based on certain aggregation properties of the economy [13]. The redistribution of endowments associated with taxation clearly destroys the collinearity of endowments, and thus the aggregation properties. (Notice that it might even induce multiple equilibria [29].)

In Section 4.4 we experimentally analyze a welfare adjustment scheme applied to certain economies where the collinear endowments receive various degrees of distortion.

4 Experimental Results

4.1 The Computational Environment and the Market Types

The experiments have been performed on a machine with an AMD Athlon, 64 bit, 2202.865 Mhz processor, 2GB of RAM, and running Red Hat Linux Release 3, Kernel Version - 2.4.21-15.0.4.

We ran our experiments on two level nested CES functions, where there are three equal size nests for all the agents. For most cases, we assigned the same elasticity of substitution to all the bottom nests; unless stated otherwise, this is assumed in what follows. We ran our experiments on various types of markets using generators that output consumers with different types of preferences and endowments. For reasons of space, the definitions of the different market types are not included in this version of the paper, and can be found in the expanded version available at the first author's website.

4.2 The Performance of an Efficient General Purpose Solver

In this section, we present the outcomes of the experiments we have carried out with the PATH solver available under GAMS/MPSGE.

PATH is a sophisticated solver, based on Newton's method, which is the most used technique to solve systems of nonlinear equations [11,14]. Newton's method constructs successive approximations to the solution, and works very well in the proximity of the solution. However, there are no guarantees of progress far from the solution. For this reason, PATH combines Newton's iterations with a line search which makes sure that at each iteration the error bound decreases, by enforcing the decrease of a suitably chosen *merit function*. This line search corresponds to a linear complementarity problem, which PATH solves by using a pivotal method [14].

Our experiments have the goal of studying the performance of GAMS/PATH for exchange and production economies with nested CES functions, with an eye on the sensitivity of PATH to specific market types and parameter ranges of interest. This will allow us to set proper grounds in order to compare the performance of PATH with that of the other approaches we have implemented.

For exchange economies with CES functions, the performance of PATH over a wide range of endowment and preference types is quite consistent and does not vary significantly [7]. On the contrary, in the case of nested CES functions, there

are market types for which we have observed a significant variation in terms of performance. In particular, we could generate many instances where the solver either failed to converge to an equilibrium or took a very long time to do so.

This phenomenon is exemplified by the data reported in Figure 1(b), which depicts the results on a market with 50 traders and 50 goods. The top and bottom elasticities range from 0.1 to 2 (in steps of 0.1), and we ran 5 experiments for each combination. The running times are plotted as a function of the ratio between the top and the bottom elasticity of substitution. Notice that the special case of CES functions corresponds to $x = 1$ on the graphs.

In addition to being sensitive to the market type, the performance of PATH also depends on the ratio between the top and the bottom elasticity. When the top elasticities are smaller than the bottom elasticities, the running time is consistently short. On the contrary, for the market type above, when the top elasticities are larger than the bottom elasticities, PATH often does not converge to an equilibrium. Figure 1(a) shows the outcomes of a similar experiment executed on a different market type, for which the solver performed better.

In general, only a few market types are capable of generating the behavior shown in Figure 1(b).

We also investigated the performance of PATH when production, modeled by nested CES functions, is added. The addition of production causes significant variation in the performance of PATH, even in the parameter ranges for which PATH does consistently well for the exchange model. This is strikingly noticeable in Figure 2, which clearly shows how the addition of production significantly worsens the performance of PATH.

4.3 Tâtonnement for Exchange Economies

In its continuous version, the tâtonnement process is governed by the differential equation system: $\frac{d\pi_i}{dt} = G_i(Z_i(\pi))$ for each $i = 1, 2, \dots, n$, where $G_i()$ is some

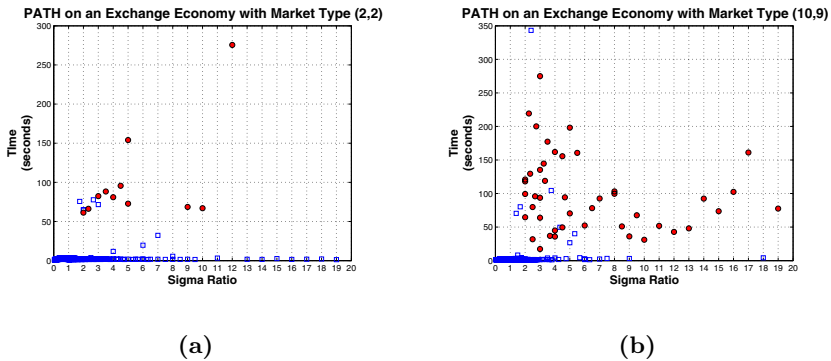


Fig. 1. Performance of PATH on exchange economies with 50 consumers and 50 goods – modeled by nested CES functions. The graphs show the running time vs the ratio between top and bottom elasticities. (a) A market of type (2,2) (b) An market of type (10,9).

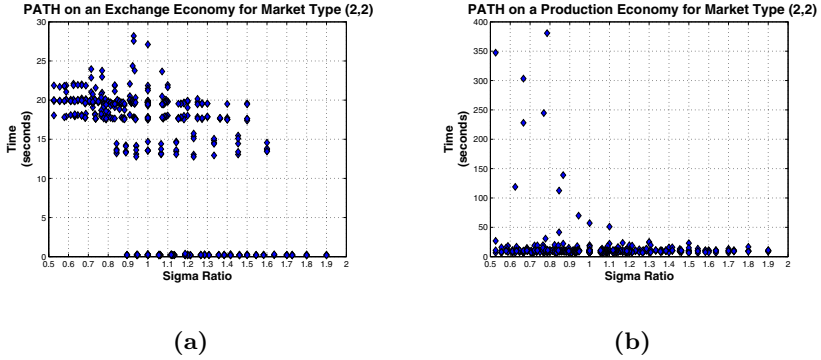


Fig. 2. A comparison of PATH on exchange and production models. The graphs show the running time vs the ratio between top and bottom elasticities. (a) An exchange economy of type (2,2) with 100 traders and goods. (b) A production economy with 50 consumers, producers, consumers goods, and factors. Initial endowments of consumer goods and preferences for them are both of type 2.

continuous, sign-preserving function and the derivative π_i is with respect to time. The continuous version of tâtonnement is more amenable to analysis of convergence and it is this process that was shown to be convergent by Arrow, Block and Hurwicz [2] markets satisfying GS. We refer the reader to [7] for a more detailed introduction to tâtonnement.

In our implementation of tâtonnement, the starting price vector is $(1, \dots, 1)$. Let π^k be the price vector after k iterations (price updates). In iteration $k + 1$, the algorithm computes the excess demand vector $Z(\pi^k)$ and then updates each price using the rule $\pi_i^{k+1} \leftarrow \pi_i^k + c_{i,k} \cdot Z_i(\pi^k)$.

One specific choice of $c_{i,k}$ that we have used in many of our experiments is $c_{i,k} = \frac{\pi_i^k}{i \cdot \max_j |Z_j(\pi^k)|}$. This choice ensures that $|c_{i,k} \cdot Z_i(\pi)| \leq \pi_i^k$ and therefore π continues to remain nonnegative. Also noteworthy is the role of i that ensures that the “step size” diminishes as the process approaches equilibrium.

The results of the experiments are summarized in Figures 3 and 4, which show the number of iterations and percentage of failures averaged over the runs for all endowment matrices. This choice was motivated by the fact that the changes in the initial endowments did not seem to have any significant impact on the performance.

While the tâtonnement scheme typically converges for markets with CES utility functions [7], it exhibits a significant variation when applied to exchange economies with nested CES utility functions. Such variation is a function of the ratio between the top and bottom values of the parameter σ . The tâtonnement process tend to have failures or to be very slow especially when the top elasticities are smaller than the bottom elasticities, and for certain types of markets, as can be seen from Table 4.

Notice that this is the opposite phenomenon of what we observed for PATH in Figure 1(b).

Preference Type	$\frac{\sigma_{top}}{\sigma_{hot}} < 1$	CES	$\frac{\sigma_{top}}{\sigma_{hot}} > 1$
0	992.8	782.6	470.3
1	1230	927	474.1
2	1214	1749	806.7
3	1426	1118	604

Fig. 3. Number of iterations of tâtonnement for different types of markets. Here the number of iterations has been averaged over the four different types of endowments.

Preference Type	$\frac{\sigma_{top}}{\sigma_{hot}} < 1$	CES	$\frac{\sigma_{top}}{\sigma_{hot}} > 1$
0	0	0	0
1	0	0	0
2	0	0	0
3	0	0	0
5	6.9	0	0
6	33.0	0	1.1
7	1.1	0	0
8	6.8	0	0

Fig. 4. Percentage of failures of the tâtonnement process for different types of markets. Here the runs have been averaged over the four different types of endowment matrices.

4.4 Welfare Adjustment

In the spirit of the welfare adjustment scheme, and the work of Jain et al. [18] and Ye [30], we implemented an algorithm for computing the equilibrium for an exchange market that uses an algorithm for the Fisher setting as a black box.

The algorithm starts off from an arbitrary initial price π^0 , and computes a sequence of prices as follows. Given π^k , the algorithm sets up a Fisher instance by setting the money of each trader to be $e_i = \pi^k \cdot w_i$, where w_i is the i -th trader's initial endowment. (The goods in the Fisher instance are obtained by aggregating the initial endowment w_i of each trader.) Let π^{k+1} be the price vector that is the solution of the Fisher instance. If π^{k+1} is within a specified tolerance of π^k , we stop and return π^{k+1} . (One can show that π^{k+1} must be an approximate equilibrium.) Otherwise, we compute π^{k+2} and proceed.

We tested our iterative algorithm for an exchange economy where traders have CES utility functions, with a particular attention to instances which can be viewed as distortions of collinear endowments produced by taxation. More precisely, we ran extensive experiments, on initial endowments chosen as $w_i(\beta) = \beta e_i + (1 - \beta)\gamma_i w$, where e_i denotes the i -th unit vector, w is the vector of all 1s, $\gamma_i \geq 0$, and $\sum_i \gamma_i = 1$. For $\beta = 0$, the endowments are collinear and the theory guarantees that π^2 , the price generated after the second step, gives the equilibrium price. For $\beta = 1$, the endowments are such that the income of the i -th trader equals the price of the i -th good, and we are as far as one can conceive

from the independence of incomes on prices. Any intermediate configuration, in particular when β is small, can be viewed as a distortion from collinearity induced by taxation.

In Figure 5 we report some typical data we have obtained. The graph shows the number of iterations as a function of β , for several economies with 25 goods and 25 traders. For values of β not too close to 1, the number of iterations is consistently below a very small constant. Very similar results have been obtained for other input settings.

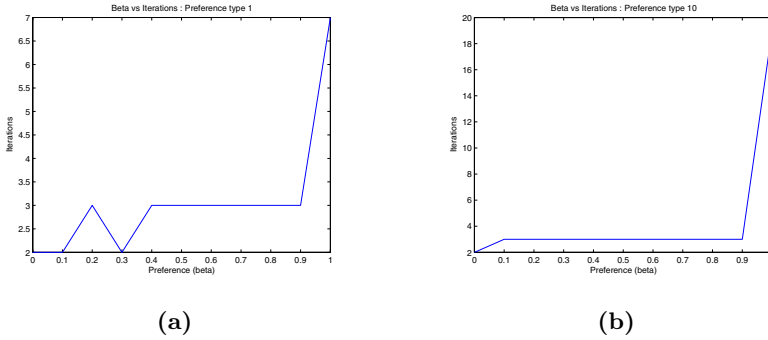


Fig. 5. Instances with 25 goods and 25 traders. The graphs show the number of iterations vs the value of β . (a) Random preferences. (b) Preferences are anti-diagonal with asymmetric noise.

Elasticity	Number of Failures	Average Number of iterations
0.1	5	58.6
0.2	6	96.75
0.3	1	84.33

Fig. 6. The number of failures (out of 10) for a run of the iterative Fisher algorithm on a CES exchange market with 25 traders and goods, preference type 2, and diagonal endowments. The third column shows the average number of Fisher iterations over the successful runs.

We also ran algorithms to test convergence as a function of the elasticities of substitution, with $\beta = 1$ for markets with preference type 2. The table in Figure 6 shows the number of failures and the average number of iterations (when the algorithm converged). The elasticities ranged from 0.1 to 1.0 with a step of 0.1. The algorithm always converged when the elasticities were larger than 0.3. The runs are averaged over 10 randomly generated markets, for each value of the elasticity. The endowment and elasticities are of type 9 and 2, respectively. As we see from the table, the algorithm always converges beyond an elasticity of 0.3.

References

1. K.J. Arrow and G. Debreu, Existence of an Equilibrium for a Competitive Economy, *Econometrica* 22 (3), pp. 265–290 (1954).
2. K. J. Arrow, H. D. Block, and L. Hurwicz. On the stability of the competitive equilibrium, II. *Econometrica* 27, 82–109 (1959).
3. K.J. Arrow, H.B. Chenery, B.S. Minhas, R.M. Solow, Capital-Labor Substitution and Economic Efficiency, *The Review of Economics and Statistics*, 43(3), 225–250 (1961).
4. A. Brooke, D. Kendrick, and A. Meeraus, GAMS: A user's guide, The Scientific Press, South San Francisco (1988).
5. B. Codenotti, S. Pemmaraju, and K. Varadarajan, On the Polynomial Time Computation of Equilibria for Certain Exchange Economies. SODA 2005.
6. B. Codenotti, S. Pemmaraju, and K. Varadarajan, The computation of market equilibria, *ACM SIGACT News*, Volume 35, Issue 4, 23-37 (December 2004).
7. B. Codenotti, B. McCune, S. Pemmaraju, R. Raman, K. Varadarajan, An experimental study of different approaches to solve the market equilibrium problem, ALENEX 2005.
8. B. Codenotti and K. Varadarajan, Efficient Computation of Equilibrium Prices for Markets with Leontief Utilities. ICALP 2004.
9. B. Codenotti, B. McCune, K. Varadarajan, Market Equilibrium via the Excess Demand Function. STOC 2005.
10. N. R. Devanur, C. H. Papadimitriou, A. Saberi, V. V. Vazirani, Market Equilibrium via a Primal-Dual-Type Algorithm. FOCS 2002, pp. 389-395. (Full version with revisions available on line, 2003.)
11. S.P. Dirkse and M.C. Ferris, A pathsearch damped Newton method for computing general equilibria, *Annals of Operations Research* p. 211-232 (1996).
12. B. C. Eaves and H. Scarf, The Solution of Systems of Piecewise Linear Equations, *Mathematics of Operations Research*, Vol. 1, No. 1, pp. 1-27 (1976).
13. E. Eisenberg, Aggregation of Utility Functions. *Management Sciences*, Vol. 7 (4), 337–350 (1961).
14. M. C. Ferris and T. S. Munson. Path 4.6. Available on line at <http://www.gams.com/solvers/path.pdf>.
15. V. A. Ginsburgh and J. L. Waelbroeck. *Activity Analysis and General Equilibrium Modelling*, North Holland, 1981.
16. S. Gjerstad, Multiple Equilibria in Exchange Economies with Homothetic, Nearly Identical Preference, University of Minnesota, Center for Economic Research, Discussion Paper 288, 1996.
17. M. Hirota, On the Probability of the Competitive Equilibrium Being Globally Stable: the CES Example, Social Sciences Working Paper 1129, Caltech (2002).
18. K. Jain, M. Mahdian, and A. Saberi, Approximating Market Equilibria, Proc. APPROX 2003.
19. T.J. Kehoe, Computation and Multiplicity of Equilibria, in *Handbook of Mathematical Economics Vol IV*, pp. 2049-2144, North Holland (1991).
20. R. R. Mantel, The welfare adjustment process: its stability properties. *International Economic Review* 12, 415-430 (1971).
21. A. Mas-Colell, M. D. Whinston, J. R. Green, *Microeconomic Theory*, Oxford University Press, 1995.
22. T. Negishi, Welfare Economics and Existence of an Equilibrium for a Competitive Economy, *Metroeconomica* 12, 92-97 (1960).

23. C.H. Papadimitriou, On the Complexity of the Parity Argument and other Inefficient Proofs of Existence, *Journal of Computer and System Sciences* 48, pp. 498-532 (1994).
24. H. Scarf, Some Examples of Global Instability of the Competitive Equilibrium, *International Economic Review* 1, 157-172 (1960).
25. H. Scarf, The Approximation of Fixed Points of a Continuous Mapping, *SIAM J. Applied Math.*, 15, pp. 1328-1343 (1967).
26. H. Scarf, Comment on "On the Stability of Competitive Equilibrium and the Patterns of Initial Holdings: An Example", *International Economic Review*, 22(2) 469-470 (1981).
27. J.B. Shoven, J. Whalley, *Applying General Equilibrium*, Cambridge University Press (1992).
28. S. Smale, A convergent process of price adjustment, *Journal of Mathematical Economics*, 3 (1976), pp. 107-120.
29. J. Whalley, S. Zhang, Tax induced multiple equilibria. Working paper, University of Western Ontario (2002).
30. Y. Ye, A Path to the Arrow-Debreu Competitive Market Equilibrium, Discussion Paper, Stanford University, February 2004. To appear in *Mathematical Programming*.

Efficient Exact Algorithms on Planar Graphs: Exploiting Sphere Cut Branch Decompositions*

Frederic Dorn¹, Eelko Penninkx², Hans L. Bodlaender², and Fedor V. Fomin¹

¹ Department of Informatics, University of Bergen, N-5020 Bergen, Norway
{frederic.dorn, fedor.fomin}@ii.uib.no

² Department of Information and Computing Sciences, Utrecht University,
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
{epennink, hansb}@cs.uu.nl

Abstract. Divide-and-conquer strategy based on variations of the Lipton-Tarjan planar separator theorem has been one of the most common approaches for solving planar graph problems for more than 20 years. We present a new framework for designing fast subexponential exact and parameterized algorithms on planar graphs. Our approach is based on geometric properties of planar branch decompositions obtained by Seymour & Thomas, combined with new techniques of dynamic programming on planar graphs based on properties of non-crossing partitions. Compared to divide-and-conquer algorithms, the main advantages of our method are a) it is a generic method which allows to attack broad classes of problems; b) the obtained algorithms provide a better worst case analysis. To exemplify our approach we show how to obtain an $O(2^{6.903\sqrt{n}}n^{3/2} + n^3)$ time algorithm solving weighted HAMILTONIAN CYCLE. We observe how our technique can be used to solve PLANAR GRAPH TSP in time $O(2^{10.8224\sqrt{n}}n^{3/2} + n^3)$. Our approach can be used to design parameterized algorithms as well. For example we introduce the first $2^{O(\sqrt{k})}k^{O(1)} \cdot n^{O(1)}$ time algorithm for parameterized PLANAR k -CYCLE by showing that for a given k we can decide if a planar graph on n vertices has a cycle of length $\geq k$ in time $O(2^{13.6\sqrt{k}}\sqrt{k}n + n^3)$.

1 Introduction

The celebrated Lipton-Tarjan planar separator theorem [18] is one of the basic approaches to obtain algorithms with subexponential running time for many problems on planar graphs [19]. The usual running time of such algorithms is $2^{O(\sqrt{n})}$ or $2^{O(\sqrt{n}\log n)}$, however the constants hidden in big-Oh of the exponent are a serious obstacle for practical implementation. During the last few years a lot of work has been done to improve the running time of divide-and-conquer type algorithms [3, 4].

* This work was partially supported by Norges forskningsråd project 160778/V30, and partially by the Netherlands Organisation for Scientific Research NWO (project *Treewidth and Combinatorial Optimisation*).

One of the possible alternatives to divide-and-conquer algorithms on planar graphs was suggested by Fomin & Thilikos [12]. The idea of this approach is very simple: compute treewidth (or branchwidth) of a planar graph and then use the well developed machinery of dynamic programming on graphs of bounded treewidth (or branchwidth)[5]. For example, given a branch decomposition of width ℓ of a graph G on n vertices, it can be shown that the maximum independent set of G can be found in time $O(2^{\frac{3\ell}{2}}n)$. The branchwidth of a planar graph G is at most $2.122\sqrt{n}$ and it can be found in time $O(n^3)$ [22] and [13]. Putting all together, we obtain an $O(2^{3.182\sqrt{n}}n + n^3)$ time algorithm solving INDEPENDENT SET on planar graphs. Note that planarity comes into play twice in this approach: First in the upper bound on the branchwidth of a graph and second in the polynomial time algorithm constructing an optimal branch decomposition. A similar approach combined with the results from Graph Minors [20] works for many parameterized problems on planar graphs [8]. Using such an approach to solve, for example, Hamiltonian cycle we end up with an $2^{O(\sqrt{n}\log n)}n^{O(1)}$ algorithm on planar graphs, as all known algorithms for this problem on graphs of treewidth ℓ require $2^{O(\ell\log \ell)}n^{O(1)}$ steps. In this paper we show how to get rid of the logarithmic factor in the exponent for a number of problems. The main idea to speed-up algorithms obtained by the branch decomposition approach is to exploit planarity for the third time: for the first time planarity is used in dynamic programming on graphs of bounded branchwidth.

Our results are based on deep results of Seymour & Thomas [22] on geometric properties of planar branch decompositions. Loosely speaking, their results imply that for a graph G embedded on a sphere Σ , some branch decompositions can be seen as decompositions of Σ into discs (or sphere cuts). We are the first describing such geometric properties of so-called sphere cut branch decompositions. Sphere cut branch decompositions seem to be an appropriate tool for solving a variety of planar graph problems. A refined combinatorial analysis of the algorithm shows that the running time can be calculated by the number of combinations of non-crossing partitions. To demonstrate the power of the new method we apply it to the following problems.

PLANAR HAMILTONIAN CYCLE. The TRAVELING SALESMAN PROBLEM (TSP) is one of the most attractive problems in Computer Science and Operations Research. For several decades, almost every new algorithmic paradigm was tried on TSP including approximation algorithms, linear programming, local search, polyhedral combinatorics, and probabilistic algorithms [17]. One of the first known exact exponential time algorithms is the algorithm of Held and Harp [14] solving TSP on n cites in time $2^n n^{O(1)}$ by making use of dynamic programming. For some special cases like EUCLIDEAN TSP (where the cites are points in the Euclidean plane and the distances between the cites are Euclidean distances), several researchers independently obtained subexponential algorithms of running time $2^{O(\sqrt{n}\cdot\log n)}n^{O(1)}$ by exploiting planar separator structures (see e.g. [15]). Smith & Wormald [23] succeed to generalize these results to d -space and the running time of their algorithm is $2^{d^{O(d)}} \cdot 2^{O(dn^{1-1/d}\log n)} + 2^{O(d)}$. Until very recent there was no known $2^{O(\sqrt{n})}n^{O(1)}$ -time algorithm even for a very special case

of TSP, namely PLANAR HAMILTONIAN CYCLE. Recently, Deĭneko et al. [7] obtained a divide-and-conquer type algorithm of running time roughly $2^{126\sqrt{n}}n^{O(1)}$ for this problem. Because their goal was to get rid of the logarithmic factor in the exponent, they put no efforts in optimizing their algorithm. But even with careful analysis, it is difficult to obtain small constants in the exponent of the divide-and-conquer algorithm due to its recursive nature.

In this paper we use sphere cut branch decompositions not only to obtain a $O(2^{6.903\sqrt{n}}n^{3/2} + n^3)$ time algorithm for PLANAR HAMILTONIAN CYCLE, but also the first $2^{O(\sqrt{n})}n^{O(1)}$ time algorithm for a generalization, PLANAR GRAPH TSP, which for a given weighted planar graph G is a TSP with distance metric the shortest path metric of G .

Parameterized PLANAR k -CYCLE. The last ten years were the evidence of a rapid development of a new branch of computational complexity: Parameterized Complexity (see the book of Downey & Fellows [9]). Roughly speaking, a parameterized problem with parameter k is *fixed parameter tractable* if it admits an algorithm with running time $f(k)|I|^\beta$. Here f is a function depending only on k , $|I|$ is the length of the non-parameterized part of the input and β is a constant. Typically, f is an exponential function, e.g. $f(k) = 2^{O(k)}$. During the last two years much attention was paid to the construction of algorithms with running time $2^{O(\sqrt{k})}n^{O(1)}$ for different problems on planar graphs. The first paper on the subject was the paper by Alber et al. [1] describing an algorithm with running time $O(2^{70\sqrt{k}}n)$ for the PLANAR DOMINATING SET problem. Different fixed parameter algorithms for solving problems on planar and related graphs are discussed in [3, 4, 8]. In the PLANAR k -CYCLE problem a parameter k is given and the question is if there exists a cycle of length at least k in a planar graph. There are several ways to obtain algorithms solving different generalizations of PLANAR k -CYCLE in time $2^{O(\sqrt{k}\log k)}n^{O(1)}$, one of the most general results is Eppstein's algorithm [10] solving the PLANAR SUBGRAPH ISOMORPHISM problem with pattern of size k in time $2^{O(\sqrt{k}\log k)}n$. By making use of sphere cut branch decompositions we succeed to find an $O(2^{13.6\sqrt{k}}kn + n^3)$ time algorithm solving PLANAR k -CYCLE.

2 Geometric Branch Decompositions of Σ -Plane Graphs

In this section we introduce our main technical tool, sphere cut branch decompositions, but first we give some definitions.

Let Σ be a sphere $(x, y, z: x^2 + y^2 + z^2 = 1)$. By a Σ -plane graph G we mean a planar graph G with the vertex set $V(G)$ and the edge set $E(G)$ drawn (without crossing) in Σ . Throughout the paper, we denote by n the number of vertices of G . To simplify notations, we usually do not distinguish between a vertex of the graph and the point of Σ used in the drawing to represent the vertex or between an edge and the open line segment representing it. An O -arc is a subset of Σ homeomorphic to a circle. An O -arc in Σ is called *noose* of a Σ -plane graph G if it meets G only in vertices. The length of a noose O is

$|O \cap V(G)|$, the number of vertices it meets. Every noose O bounds two open discs Δ_1, Δ_2 in Σ , i.e. $\Delta_1 \cap \Delta_2 = \emptyset$ and $\Delta_1 \cup \Delta_2 \cup O = \Sigma$.

Branch Decompositions and Carving Decompositions. A *branch decomposition* $\langle T, \mu \rangle$ of a graph G consists of an un-rooted ternary (i.e. all internal vertices of degree three) tree T and a bijection $\mu : L \rightarrow E(G)$ between the set L of leaves of T to the edge set of G . We define for every edge e of T the *middle set* $\text{mid}(e) \subseteq V(G)$ as follows: Let T_1 and T_2 be the two connected components of $T \setminus \{e\}$. Then let G_i be the graph induced by the edge set $\{\mu(f) : f \in L \cap V(T_i)\}$ for $i \in \{1, 2\}$. The *middle set* is the intersection of the vertex sets of G_1 and G_2 , i.e., $\text{mid}(e) := V(G_1) \cap V(G_2)$. The *width* bw of $\langle T, \mu \rangle$ is the maximum order of the middle sets over all edges of T , i.e., $\text{bw}(\langle T, \mu \rangle) := \max\{|\text{mid}(e)| : e \in T\}$. An optimal branch decomposition of G is defined by the tree T and the bijection μ which together provide the minimum width, the *branchwidth* $\text{bw}(G)$.

A *carving decomposition* $\langle T, \mu \rangle$ is similar to a branch decomposition, only with the difference that μ is the bijection between the leaves of the tree and the *vertex set* of the graph. For an edge e of T , the counterpart of the middle set, called *cut set* $\text{cut}(e)$, contains the edges of the graph with end vertices in the leaves of both subtrees. The counterpart of branchwidth is *carvingwidth*.

We will need the following result.

Proposition 1 ([12]). *For any planar graph G , $\text{bw}(G) \leq \sqrt{4.5n} \leq 2.122\sqrt{n}$.*

Sphere Cut Branch Decompositions. For a Σ -plane graph G , we define a *sphere cut branch decomposition* or *sc-branch decomposition* $\langle T, \mu, \pi \rangle$ as a branch decomposition such that for every edge e of T there exists a noose O_e bounding the two open discs Δ_1 and Δ_2 such that $G_i \subseteq \Delta_i \cup O_e$, $1 \leq i \leq 2$. Thus O_e meets G only in $\text{mid}(e)$ and its length is $|\text{mid}(e)|$. Clockwise traversal of O_e in the drawing of G defines the cyclic ordering π of $\text{mid}(e)$. We always assume that the vertices of every middle set $\text{mid}(e) = V(G_1) \cap V(G_2)$ are enumerated according to π .

The following theorem provides us with the main technical tool. It follows almost directly from the results of Seymour & Thomas [22] and Gu & Tamaki [13]. Since this result is not explicitly mentioned in [22], we provide some explanations below.

Theorem 1. *Let G be a connected Σ -plane graph of branchwidth $\leq \ell$ without vertices of degree one. There exists an sc-branch decomposition of G of width $\leq \ell$ and such a branch decomposition can be constructed in time $O(n^3)$.*

Proof. Let G be a Σ -plane graph of branchwidth $\leq \ell$ and with minimal vertex degree ≥ 2 . Then, $I(G)$ is the simple bipartite graph with vertex $V(G) \cup E(G)$, in which $v \in V(G)$ is adjacent to $e \in E(G)$ if and only if v is an end of e in G . The *medial graph* M_G of G has vertex set $E(G)$, and for every vertex $v \in V(G)$ there is a cycle C_v in M_G with the following properties:

1. The cycles C_v of M_G are mutually edge-disjoint and have union M_G ;
2. For each $v \in V(G)$, let the neighbors of v in $I(G)$ be w_1, \dots, w_t enumerated according to the cyclic order of the edges $\{v, w_1\}, \dots, \{v, w_t\}$ in the drawing of $I(G)$; then C_v has vertex set $\{w_1, \dots, w_t\}$ and w_{i-1} is adjacent to w_i ($1 \leq i \leq t$), where w_0 means w_t .

In a *bond carving decomposition* of a graph, every cut set is a bond of the graph, i.e., every cut set is a minimal cut. Seymour & Thomas ((5.1) and (7.2) [22]) show that a Σ -plane graph G without vertices of degree one is of branch-width $\leq \ell$ if and only if M_G has a bond carving decomposition of width $\leq 2\ell$. They also show (Algorithm (9.1) in [22]) how to construct an optimal bond carving decompositions of the medial graph M_G in time $O(n^4)$. A refinement of the algorithm in [13] give running time $O(n^3)$. A bond carving decomposition $\langle T, \mu \rangle$ of M_G is also a branch decomposition of G (vertices of M_G are the edges of G) and it can be shown (see the proof of (7.2) in [22]) that for every edge e of T if the cut set $\text{cut}(e)$ in M_G is of size $\leq 2\ell$, then the middle set $\text{mid}(e)$ in G is of size $\leq \ell$. It is well known that the edge set of a minimal cut forms a cycle in the dual graph. The dual graph of a medial graph M_G is the *radial graph* R_G . In other words, R_G is a bipartite graph with the bipartition $F(G) \cup V(G)$. A vertex $v \in V(G)$ is adjacent in R_G to a vertex $f \in F(G)$ if and only if the vertex v is incident to the face f in the drawing of G . Therefore, a cycle in R_G forms a noose in G .

To summarize, for every edge e of T , $\text{cut}(e)$ is a minimal cut in M_G , thus $\text{cut}(e)$ forms a cycle in R_G (and a noose O_e in G). Every vertex of M_G is in one of the open discs Δ_1 and Δ_2 bounded by O_e . Since O_e meets G only in vertices, we have that $O_e \cap V(G) = \text{mid}(e)$. Thus for every edge e of T and the two subgraphs G_1 and G_2 of G formed by the leaves of the subtrees of $T \setminus \{e\}$, O_e bounds the two open discs Δ_1 and Δ_2 such that $G_i \subseteq \Delta_i \cup O_e$, $1 \leq i \leq 2$.

Finally, with a given bond carving decomposition $\langle T, \mu \rangle$ of the medial graph M_G , it is straightforward to construct cycles in R_G corresponding to $\text{cut}(e)$, $e \in E(T)$, and afterwards to compute ordering π of $\text{mid}(e)$ in linear time. \square

Non-crossing Matchings. Together with sphere cut branch decompositions, non-crossing matchings give us the key to our later dynamic programming approach. A *non-crossing partitions (ncp)* is a partition $P(n) = \{P_1, \dots, P_m\}$ of the set $S = \{1, \dots, n\}$ such that there are no numbers $a < b < c < d$ where $a, c \in P_i$, and $b, d \in P_j$ with $i \neq j$. A partition can be visualized by a circle with n equidistant vertices on it's border, where every set of the partition is represented by the convex polygon with it's elements as endpoints. A partition is non-crossing if these polygons do not overlap. Non-crossing partitions were introduced by Kreweras [16], who showed that the number of non-crossing partitions over n vertices is equal to the n -th Catalan number:

$$\text{CN}(n) = \frac{1}{n+1} \binom{2n}{n} \sim \frac{4^n}{\sqrt{\pi n^3}} \approx 4^n \tag{1}$$

A *non-crossing matching (ncm)* is a special case of a ncp, where $|P_i| = 2$ for every element of the partition. A ncm can be visualized by placing n vertices on a straight line, and connecting matching vertices with arcs at one fixed side of the line. A matching is non-crossing if these arcs do not cross. The number of non-crossing matchings over n vertices is given by:

$$M(n) = \text{CN}\left(\frac{n}{2}\right) \sim \frac{2^n}{\sqrt{\pi\left(\frac{n}{2}\right)^{\frac{3}{2}}}} \approx 2^n \tag{2}$$

3 Planar Hamiltonian Cycle

In this section we show how sc-branch decompositions in combination with ncm’s can be used to design subexponential parameterized algorithms. In the PLANAR HAMILTONIAN CYCLE problem we are given a weighted Σ -plane graph $G = (V, E)$ with weight function $w: E(G) \rightarrow \mathbb{N}$ and we ask for a cycle of minimum weight through all vertices of V . We can formulate the problem in a different way: A labelling $\mathcal{H}: E(G) \rightarrow \{0, 1\}$ is *Hamiltonian* if the subgraph $G_{\mathcal{H}}$ of G formed by the edges with positive labels is a spanning cycle. Find a Hamiltonian labelling \mathcal{H} minimizing $\sum_{e \in E(G)} \mathcal{H}(e) \cdot w(e)$. For an edge labelling \mathcal{H} and vertex $v \in V(G)$ we define the \mathcal{H} -degree $\text{deg}_{\mathcal{H}}(v)$ of v as the sum of labels assigned to the edges incident to v . Though the use of labellings makes the algorithm more sophisticated, it is necessary for the understanding of the latter approach for PLANAR GRAPH TSP. Let $\langle T, \mu, \pi \rangle$ be a sc-branch decomposition of G of width ℓ . We root T by arbitrarily choosing an edge e , and subdivide it by inserting a new node s . Let e', e'' be the new edges and set $\text{mid}(e') = \text{mid}(e'') = \text{mid}(e)$. Create a new node *root* r , connect it to s and set $\text{mid}(\{r, s\}) = \emptyset$. Each internal node v of T now has one adjacent edge on the path from v to r , called *parent edge* e_P , and two adjacent edges towards the leaves, called *left child* e_L and *right child* e_R . For every edge e of T the subtree towards the leaves is called the *lower part* and the rest the *residual part* with regard to e . We call the subgraph G_e induced by the leaves of the lower part of e the *subgraph rooted at e* . Let e be an edge of T and let O_e be the corresponding noose in Σ . The noose O_e partitions Σ into two discs, one of which, Δ_e , contains G_e .

We call a labelling $\mathcal{P}[e]: E(G_e) \rightarrow \{0, 1\}$ a *partial Hamiltonian labelling* if the subgraph $G_{\mathcal{P}[e]}$ induced by the edges with positive labels satisfies the following properties:

- For every vertex $v \in V(G_e) \setminus O_e$, the $\mathcal{P}[e]$ -degree $\text{deg}_{\mathcal{P}[e]}(v)$, i.e. the sum of labels assigned by $\mathcal{P}[e]$ to the edges incident to v , is two.
- Every connected component of $G_{\mathcal{P}[e]}$ has two vertices in O_e with $\text{deg}_{\mathcal{P}[e]}(v) = 1$ for $e \neq \{r, s\}$; For $e = \{r, s\}$, $G_{\mathcal{P}[e]}$ is a cycle.

Observe that $G_{\mathcal{P}[e]}$ forms a collection of disjoint paths, and note that every partial Hamiltonian labelling of $G_{\{r,s\}}$ is also a Hamiltonian labelling.

For dynamic programming we need to keep for every edge e of T the information on which vertices of the disjoint paths of $G_{\mathcal{P}[e]}$ of all possible partial

Hamiltonian labellings $\mathcal{P}[e]$ hit $O_e \cap V(G)$ and for every vertex $v \in O_e \cap V(G)$ the information if $\deg_{\mathcal{P}[e]}(v)$ is either 0, or 1, or 2.

And here the geometric properties of sc-branch decompositions in combination with ncm's come into play. For a partial Hamiltonian labelling $\mathcal{P}[e]$ let P be a path of $G_{\mathcal{P}[e]}$. We scan the vertices of $V(P) \cap O_e$ according to the ordering π and mark with '1_l' the first and with '1_r' the last vertex of P on O_e . We also mark by '2' the other 'inner' vertices of $V(P) \cap O_e$. If we mark in such a way all vertices of $V(G_{\mathcal{P}[e]}) \cap O_e$, then given the obtained sequence with marks '1_l', '1_r', '2', and '0', one can decode the complete information on which vertices of each path of $V(G_{\mathcal{P}[e]})$ hit O_e . This is possible because O_e bounds the disc Δ_e and the graph $G_{\mathcal{P}[e]}$ is in Δ_e . The sets of endpoints of every path are the elements of an ncm. Hence, with the given ordering π the '1_l' and '1_r' encode an ncm.

For an edge e of T and corresponding noose O_e , the state of dynamic programming is specified by an ordered ℓ -tuple $\mathbf{t}_e := (v_1, \dots, v_\ell)$. Here, the variables v_1, \dots, v_ℓ correspond to the vertices of $O_e \cap V(G)$ taken according to the cyclic order π with an arbitrary first vertex. This order is necessary for a well-defined encoding for the states when allowing v_1, \dots, v_ℓ to have one of the four values: 0, 1_l, 1_r, 2. Hence, there are at most $O(4^\ell |V(G)|)$ states. For every state, we compute a value $W_e(v_1, \dots, v_\ell)$. This value is equal to W if and only if W is the minimum weight of a partial Hamiltonian labelling $\mathcal{P}[e]$ such that:

1. For every path P of $G_{\mathcal{P}[e]}$ the first vertex of $P \cap O_e$ in π is represented by 1_l and the last vertex is represented by 1_r. All other vertices of $P \cap O_e$ are represented by 2.
2. Every vertex $v \in (V(G_e) \cap O_e) \setminus G_{\mathcal{P}[e]}$ is marked by 0.

We put $W = +\infty$ if no such labelling exists. For every vertex v the numerical part of its value gives $\deg_{\mathcal{P}[e]}(v)$.

To compute an optimal Hamiltonian labelling we perform dynamic programming over middle sets $\text{mid}(e) = O(e) \cap V(G)$, starting at the leaves of T and working bottom-up towards the root edge. The first step in processing the middle sets is to initialize the leaves with values $(0, 0)$, $(1_{l}, 1_{r})$. Then, bottom-up, update every pair of states of two child edges e_L and e_R to a state of the parent edge e_P assigning a finite value W_P if the state corresponds to a feasible partial Hamiltonian labelling.

Let O_L, O_R , and O_P be the nooses corresponding to edges e_L, e_R and e_P . Due to the definition of branch decompositions, every vertex must appear in at least two of the three middle sets and we can define the following partition of the set $(O_L \cup O_R \cup O_P) \cap V(G)$ into sets $I := O_L \cap O_R \cap V(G)$ and $D := O_P \cap V(G) \setminus I$ (I stands for 'Intersection' and D for 'symmetric Difference'). The disc Δ_P bounded by O_P and including the subgraph rooted at e_P contains the union of the discs Δ_L and Δ_R bounded by O_L and O_R and including the subgraphs rooted at e_L and e_R . Thus $|O_L \cap O_R \cap O_P \cap V(G)| \leq 2$. The vertices of $O_L \cap O_R \cap O_P \cap V(G)$ are called *portal vertices*.

We compute all valid assignments to the variables $\mathbf{t}_P = (v_1, v_2, \dots, v_p)$ corresponding to the vertices $\text{mid}(e_P)$ from all possible valid assignments to the

variables of \mathbf{t}_L and \mathbf{t}_R . For a symbol $x \in \{0, 1_{\lceil}, 1_{\lfloor}, 2\}$ we denote by $|x|$ its 'numerical' part. We say that an assignment c_P is *formed* by assignments c_L and c_R if for every vertex $v \in (O_L \cup O_R \cup O_P) \cap V(G)$:

1. $v \in D$: $c_P(v) = c_L(v)$ if $v \in O_L \cap V(G)$, or $c_P(v) = c_R(v)$ otherwise.
2. $v \in I \setminus O_P$: $(|c_L(v)| + |c_R(v)|) = 2$.
3. v portal vertex: $|c_P(v)| = |c_L(v)| + |c_R(v)| \leq 2$.

We compute all ℓ -tuples for $\text{mid}(e_P)$ that can be formed by tuples corresponding to $\text{mid}(e_L)$ and $\text{mid}(e_R)$ and check if the obtained assignment corresponds to a labelling without cycles. For every encoding of \mathbf{t}_P , we set $W_P = \min\{W_P, W_L + W_R\}$.

For the root edge $\{r, s\}$ and its children e' and e'' note that $(O_{e'} \cup O_{e''}) \cap V(G) = I$ and $O_{\{r,s\}} = \emptyset$. Hence, for every $v \in V(G_{\mathcal{P}[\{r,s\}]})$ it must hold that $\text{deg}_{\mathcal{P}[\{r,s\}]}(v)$ is two, and that the labellings form a cycle. The optimal Hamiltonian labelling of G results from $\min_{\mathbf{t}_{\{r,s\}}} \{W_r\}$.

Analyzing the algorithm, we obtain the following lemma.

Lemma 1. PLANAR HAMILTONIAN CYCLE on a graph G with branchwidth ℓ can be solved in time $O(2^{3 \cdot 292\ell} \ell n + n^3)$.

Proof. By Theorem 1, an sc-branch decomposition T of width $\leq \ell$ of G can be found in $O(n^3)$.

Assume we have three adjacent edges $e_P, e_L,$ and e_R of T with $|O_L| = |O_R| = |O_P| = \ell$. Without loss of generality we limit our analysis to even values for ℓ , and for simplicity assume there are no portal vertices. This can only occur if $|I| = |D \cap O_L| = |D \cap O_R| = \frac{\ell}{2}$.

By just checking every combination of ℓ -tuples from O_L and O_R we obtain a bound of $O(\ell 4^{2\ell})$ for our algorithm.

Some further improvement is apparent, as for the vertices $u \in I$ we want the sum of the $\{0, 1_{\lceil}, 1_{\lfloor}, 2\}$ assignments from both sides to be 2.

We start by giving an expression for $Q(\ell, m)$: the number of ℓ -tuples over ℓ vertices where the $\{0, 1_{\lceil}, 1_{\lfloor}, 2\}$ assignments for vertices from I is fixed and contains m 1_{\lceil} 's and 1_{\lfloor} 's. The only freedom is thus in the $\ell/2$ vertices in $D \cap O_L$ and $D \cap O_R$, respectively:

$$Q(\ell, m) = \sum_{i=m\%2}^{\frac{\ell}{2}, 2} \binom{\frac{\ell}{2}}{i} 2^{\frac{\ell}{2}-i} \text{CN}\left(\frac{i+m}{2}\right) \tag{3}$$

This expression is a summation over the number of 1_{\lceil} 's and 1_{\lfloor} 's in $D \cap O_L$ and $D \cap O_R$, respectively. The starting point is $m\%2$ (m modulo 2), and the 2 at the top of the summation indicates that we increase i with steps of 2, as we want $i+m$ to be even (the 1_{\lceil} 's and 1_{\lfloor} 's have to form a ncm). The term $\binom{\frac{\ell}{2}}{i}$ counts the possible locations for the 1_{\lceil} 's and 1_{\lfloor} 's, the $2^{\frac{\ell}{2}-i}$ counts the assignment of $\{0, 2\}$ to the remaining $\ell/2 - i$ vertices, and the $\text{CN}(\frac{i+m}{2})$ term counts the ncm's over the 1_{\lceil} 's and 1_{\lfloor} 's. As we are interested in exponential behaviour for large values of ℓ we ignore that $i+m$ is even, and use that $\text{CN}(n) \approx 4^n$:

$$Q(\ell, m) \approx \sum_{i=0}^{\frac{\ell}{2}} \binom{\frac{\ell}{2}}{i} 2^{\frac{\ell}{2}-i} 2^{i+m} = 2^{\frac{\ell}{2}+m} \sum_{i=0}^{\frac{\ell}{2}} \binom{\frac{\ell}{2}}{i} = 2^{\ell+m} \tag{4}$$

We can now count the total cost of forming an ℓ -tuple from O_P . We sum over i : the number of 1_{\lceil} 's and 1_{\rfloor} 's in the assignment for I :

$$C(\ell) = \sum_{i=0}^{\frac{\ell}{2}} \binom{\frac{\ell}{2}}{i} 2^{\frac{\ell}{2}-i} Q(\ell, i)^2 \tag{5}$$

Straightforward calculation by approximation yields:

$$C(\ell) \approx \sum_{i=0}^{\frac{\ell}{2}} \binom{\frac{\ell}{2}}{i} 2^{\frac{\ell}{2}-i} 2^{2\ell+2i} = 2^{\frac{5\ell}{2}} \sum_{i=0}^{\frac{\ell}{2}} \binom{\frac{\ell}{2}}{i} 2^i = 2^{\frac{5\ell}{2}} 3^{\frac{\ell}{2}} = (4\sqrt{6})^\ell \tag{6}$$

By Proposition 1 and Lemma 1 we achieve the running time $O(2^{6.987\sqrt{n}} n^{3/2} + n^3)$ for PLANAR HAMILTONIAN CYCLE.

3.1 Forbidding Cycles

We can further improve upon the previous bound by only forming encodings that don't create a cycle. As cycles can only be formed at the vertices in I with numerical part 1 in both O_L and O_R , we only consider these vertices. Note that within these vertices both in O_L and O_R every vertex with a 1_{\rfloor} has to be paired with a 1_{\lceil} , whereas a 1_{\lceil} could be paired with a 1_{\rfloor} that lies in D . We encode every vertex by a $\{1_{\lceil}, 1_{\rfloor}\}^2$ assignment, where the first corresponds to the state in O_L , and the second to the state in O_R . For example $|1_{\lceil}1_{\lceil}, 1_{\rfloor}1_{\rfloor}\rangle$ corresponds to a cycle over two vertices. We obtain the following combinatorial problem: given n vertices with a $\{1_{\lceil}, 1_{\rfloor}\}^2$ assignment to every vertex, how many combinations can be formed that induce no cycles and pair every 1_{\rfloor} with a 1_{\lceil} at the same side?

Exact counting is complex, so we use a different approach. We try to find some small z such that $|B(b)|$ is $O(z^n)$. Let $B(i)$ denote the set of all feasible combinations over i vertices: $B(0) = \emptyset$, $B(1) = \{|1_{\lceil}1_{\lceil}\rangle\}$, $B(2) = \{|1_{\lceil}1_{\lceil}, 1_{\lceil}1_{\lceil}|, |1_{\lceil}1_{\lceil}, 1_{\rfloor}1_{\lceil}|, |1_{\lceil}1_{\lceil}, 1_{\rfloor}1_{\rfloor}\rangle\}$, etc. Note that $|1_{\lceil}1_{\lceil}, 1_{\rfloor}1_{\rfloor}\rangle$ is not included in $B(2)$ as this is a cycle. We map all items of $B(i)$ to a fixed number of classes C_1, \dots, C_m and define $x_i = \{x_{i1}, \dots, x_{im}\}^T$ as the number of elements in each class.

Suppose we use two classes: C_1 contains all items $|\dots, 1_{\lceil}1_{\lceil}\rangle$, and C_2 contains all other items. Note that adding $1_{\rfloor}1_{\rfloor}$ to items from C_1 is forbidden, as this will lead to a cycle. Addition of $1_{\lceil}1_{\lceil}$ to items from C_2 gives us items of class C_1 . Addition of $1_{\lceil}1_{\rfloor}$ or $1_{\rfloor}1_{\lceil}$ to either class leads to items of class C_2 , or can lead to infeasible encodings. These observations show that $A = \begin{pmatrix} 1 & 1 \\ 2 & 3 \end{pmatrix}$. As the largest real eigenvalue of A is $2 + \sqrt{3}$, we have $z \leq 3.73205$.

Using these two classes eliminated all cycles over two consecutive vertices. By using more classes we can prevent larger cycles, and obtain tighter bounds for z . With only three classes we obtain $z \leq 3.68133$. This bound is definitely not tight, but computational research suggests that z is probably larger than 3.5. By using the value 3.68133 for z we obtain the following result:

Theorem 2. PLANAR HAMILTONIAN CYCLE is solvable in $O(2^{6.903\sqrt{n}}n^{3/2} + n^3)$.

4 Variants

In this section we will discuss results on other non-local problems on planar graphs.

Longest Cycle on Planar Graphs. Let C be a cycle in G . For an edge e of sc-branch decomposition tree T , the noose O_e can affect C in two ways: Either cycle C is partitioned by O_e such that in G_e the remains of C are disjoint paths, or C is not touched by O_e and thus is completely in G_e or $G \setminus E(G_e)$.

With the same encoding as for PLANAR HAMILTONIAN CYCLE, we add a counter for all states t_e which is initialized by 0 and counts the maximum number of edges over all possible vertex-disjoint paths represented by one t_e . In contrast to PLANAR HAMILTONIAN CYCLE, we allow for every vertex $v \in I$ that $|c_L(v)| + |c_R(v)| = 0$ in order to represent the isolated vertices. A cycle as a connected component is allowed if all other components are isolated vertices. Then all other vertices in $V(G) \setminus V(G_P)$ of the residual part of T must be of value 0. Implementing a counter Z for the actual longest cycle, a state in t_P consisting of only 0's represents a collection of isolated vertices with Z storing the longest path in G_P without vertices in $\text{mid}(e)$. At the root edge, Z gives the size of the longest cycle. Thus, PLANAR LONGEST CYCLE is solvable in time $O(2^{7.223\sqrt{n}}n^{3/2} + n^3)$.

k -Cycle on Planar graphs is the problem of finding a cycle of length at least a parameter k . The algorithm on LONGEST CYCLE can be used for obtaining parameterized algorithms by adopting the techniques from [8, 11].

Before we proceed, let us remind the notion of a minor. A graph H obtained by a sequence of edge-contractions from a graph G is said to be a *contraction* of G . H is a *minor* of G if H is the subgraph of a some contraction of G . Let us note that if a graph H is a minor of G and G contains a cycle of length $\geq k$, then so does G .

We need the following combination of statements (4.3) in [21] and (6.3) in [20].

Theorem 3 ([20]). Let $k \geq 1$ be an integer. Every planar graph with no $(k \times k)$ -grid as a minor has branchwidth $\leq 4k - 3$.

It is easy to check that every $(\sqrt{k} \times \sqrt{k})$ -grid, $k \geq 2$, contains a cycle of length $\geq k - 1$. This observation combined with Theorem 3 suggests the following parameterized algorithm. Given a planar graph G and integer k , first compute the branchwidth of G . If the branchwidth of G is at least $4\sqrt{k+1} - 3$ then by Theorem 3, G contains a $(\sqrt{k+1} \times \sqrt{k+1})$ -grid as a minor and thus contains a cycle of length $\geq k$. If the branchwidth of G is $< 4\sqrt{k+1} - 3$ we can find the longest cycle in G in time $O(2^{13.6\sqrt{k}}\sqrt{k}n + n^3)$. We conclude with the following theorem.

Theorem 4. PLANAR k -CYCLE is solvable in time $O(2^{13.6\sqrt{k}}\sqrt{k}n + n^3)$.

By standard techniques (see for example [9]) the recognition algorithm for PLANAR k -CYCLE can easily be turned into a constructive one.

Planar Graph TSP. In the PLANAR GRAPH TSP we are given a weighted Σ -plane graph $G = (V, E)$ with weight function $w: E(G) \rightarrow N$ and we are looking for a shortest closed walk that visits all vertices of G at least once. Equivalently, this is TSP with distance metric the shortest path metric of G . The algorithm for PLANAR GRAPH TSP is very similar to the algorithm for PLANAR HAMILTONIAN CYCLE so we mention here only the most important details. Every shortest closed walk in G corresponds to the minimum Eulerian subgraph in the graph G' obtained from G by adding to each edge a parallel edge. Since every vertex of an Eulerian graph is of even degree we obtain an *Eulerian* labelling $\mathcal{E}: E(G) \rightarrow \{0, 1, 2\}$ with the subgraph $G_{\mathcal{E}}$ of G formed by the edges with positive labels is a connected spanning subgraph and for every vertex $v \in V$ the sum of labels assigned to edges incident to v is even. Thus the problem is equivalent to finding an Eulerian labelling \mathcal{E} minimizing $\sum_{e \in E(G)} \mathcal{E}(e) \cdot w(e)$.

In contrast to the approach for PLANAR HAMILTONIAN CYCLE, the parity plays an additional role in dynamic programming, and we obtain a bit more sophisticated approach.

Theorem 5. PLANAR GRAPH TSP is solvable in time $O(2^{10.8224\sqrt{n}}n^{3/2} + n^3)$.

5 Conclusive Remarks

In this paper we introduce a new algorithm design technique based on geometric properties of branch decompositions. Our technique can be also applied to constructing $2^{O(\sqrt{n})} \cdot n^{O(1)}$ -time algorithms for a variety of cycle, path, or tree subgraph problems in planar graphs like HAMILTONIAN PATH, LONGEST PATH, and CONNECTED DOMINATING SET, and STEINER TREE among others. An interesting question here is if the technique can be extended to more general problems, like SUBGRAPH ISOMORPHISM. For example, Eppstein [10] showed that PLANAR SUBGRAPH ISOMORPHISM problem with pattern of size k can be solved in time $2^{O(\sqrt{k} \log k)}n$. Can we get rid of the logarithmic factor in the exponent (maybe in exchange to a higher polynomial degree)?

The results of Cook & Seymour [6] on using branch decomposition to obtain high-quality tours for (general) TSP show that branch decomposition based algorithms work much faster than their worst case time analysis shows.

Together with our preliminary experience on the implementation of a similar algorithm technique for solving PLANAR VERTEX COVER in [2], we conjecture that sc-branch decomposition based algorithms perform much faster in practice.

References

- [1] J. ALBER, H. L. BODLAENDER, H. FERNAU, T. KLOKS, AND R. NIEDERMEIER, *Fixed parameter algorithms for dominating set and related problems on planar graphs*, Algorithmica, 33 (2002), pp. 461–493.
- [2] J. ALBER, F. DORN, AND R. NIEDERMEIER, *Experimental evaluation of a tree decomposition-based algorithm for vertex cover on planar graphs*, Discrete Applied Mathematics, 145 (2005), pp. 219–231.

- [3] J. ALBER, H. FERNAU, AND R. NIEDERMEIER, *Graph separators: a parameterized view*, Journal of Computer and System Sciences, 67 (2003), pp. 808–832.
- [4] ———, *Parameterized complexity: exponential speed-up for planar graph problems*, Journal of Algorithms, 52 (2004), pp. 26–56.
- [5] H. L. BODLAENDER, *A tourist guide through treewidth*, Acta Cybernet., 11 (1993), pp. 1–21.
- [6] W. COOK AND P. SEYMOUR, *Tour merging via branch-decomposition*, INFORMS Journal on Computing, 15 (2003), pp. 233–248.
- [7] V. G. DEĀNEKO, B. KLINZ, AND G. J. WOEGINGER, *Exact algorithms for the Hamiltonian cycle problem in planar graphs*, Oper. Res. Lett., (2005), p. to appear.
- [8] E. D. DEMAINE, F. V. FOMIN, M. T. HAJIAGHAYI, AND D. M. THILIKOS, *Subexponential parameterized algorithms on graphs of bounded genus and H -minor-free graphs*, in 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), New York, 2004, ACM, pp. 823–832.
- [9] R. G. DOWNEY AND M. R. FELLOWS, *Parameterized complexity*, Springer-Verlag, New York, 1999.
- [10] D. EPPSTEIN, *Subgraph isomorphism in planar graphs and related problems*, Journal of Graph Algorithms and Applications, 3 (1999), pp. 1–27.
- [11] F. FOMIN AND D. THILIKOS, *Dominating sets in planar graphs: Branch-width and exponential speed-up*, in 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), New York, 2003, ACM, pp. 168–177.
- [12] ———, *A simple and fast approach for solving problems on planar graphs*, in 21st Annual Symposium on Theoretical Aspects of Computer Science (STACS), vol. 2996 of Lecture Notes in Comput. Sci., Berlin, 2004, Springer, pp. 56–67.
- [13] Q.-P. GU AND H. TAMAKI, *Optimal branch-decomposition of planar graphs in $O(n^3)$* , in 32nd International Colloquium on Automata, Languages and Programming (ICALP), 2005, p. to appear.
- [14] M. HELD AND R. M. KARP, *A dynamic programming approach to sequencing problems*, Journal of SIAM, 10 (1962), pp. 196–210.
- [15] R. Z. HWANG, R. C. CHANG, AND R. C. T. LEE, *The searching over separators strategy to solve some NP-hard problems in subexponential time*, Algorithmica, 9 (1993), pp. 398–423.
- [16] G. KREWERAS, *Sur les partition non croisées d'un cercle*, Discrete Mathematics, 1 (1972), pp. 333–350.
- [17] E. L. LAWLER, J. K. LENSTRA, AND A. H. G. RINNOOY KAN, eds., *The traveling salesman problem*, John Wiley & Sons Ltd., Chichester, 1985.
- [18] R. J. LIPTON AND R. E. TARJAN, *A separator theorem for planar graphs*, SIAM Journal on Applied Mathematics, 36 (1979), pp. 177–189.
- [19] ———, *Applications of a planar separator theorem*, SIAM Journal on Computing, 9 (1980), pp. 615–627.
- [20] N. ROBERTSON, P. SEYMOUR, AND R. THOMAS, *Quickly excluding a planar graph*, Journal of Combinatorial Theory Series B, 62 (1994), pp. 323–348.
- [21] N. ROBERTSON AND P. D. SEYMOUR, *Graph minors. X. Obstructions to tree-decomposition*, Journal of Combinatorial Theory Series B, 52 (1991), pp. 153–190.
- [22] P. SEYMOUR AND R. THOMAS, *Call routing and the ratcatcher*, Combinatorica, 15 (1994), pp. 217–241.
- [23] W. D. SMITH AND N. C. WORMALD, *Geometric separator theorems and applications*, in The 39th Annual IEEE Symposium on Foundations of Computer Science (FOCS 1998), IEEE Computer Society, 1998, pp. 232–243.

An Algorithm for the SAT Problem for Formulae of Linear Length

Magnus Wahlström*

Department of Computer and Information Science,
Linköping University, SE-581 83 Linköping, Sweden
magwa@ida.liu.se

Abstract. We present an algorithm that decides the satisfiability of a CNF formula where every variable occurs at most k times in time $O(2^{N(1-c/(k+1)+O(1/k^2))})$ for some c (that is, $O(\alpha^N)$ with $\alpha < 2$ for every fixed k). For $k \leq 4$, the results coincide with an earlier paper where we achieved running times of $O(2^{0.1736N})$ for $k = 3$ and $O(2^{0.3472N})$ for $k = 4$ (for $k \leq 2$, the problem is solvable in polynomial time). For $k > 4$, these results are the best yet, with running times of $O(2^{0.4629N})$ for $k = 5$ and $O(2^{0.5408N})$ for $k = 6$. As a consequence of this, the same algorithm is shown to run in time $O(2^{0.0926L})$ for a formula of length (*i.e.* total number of literals) L . The previously best bound in terms of L is $O(2^{0.1030L})$. Our bound is also the best upper bound for an exact algorithm for a 3SAT formula with up to six occurrences per variable, and a 4SAT formula with up to eight occurrences per variable.

1 Introduction

The boolean satisfiability problem, and its restricted variants, is one of the most well-studied classes of NP-complete problems. In the general case of formulae in conjunctive normal form with no restrictions on the clauses, or the number of appearances of the variables, no algorithm that decides satisfiability in time $O(\alpha^N)$ with $\alpha < 2$ for N variables is believed to exist. Nevertheless, there are a number of algorithms that improve on the trivial $O(2^N)$ in this case. With N variables and M clauses, an algorithm was recently published [6] that runs in expected time $O(2^{N(1-1/\alpha)})$ where $\alpha = \log(M/N) + O(\log \log M)$, and previously an algorithm that runs in expected time $O(2^{N(1-c/\sqrt{N})})$ for a constant c was known [4]. We see that the former bound is stronger when M is polynomial in N , while the latter bound is stronger for formulae with very many clauses. For a deterministic algorithm, there is one that runs in time $O(2^{N(1-1/\log_2 2M)})$ [5, 13]. Additionally, there is an algorithm by Hirsch [9] that runs in time $O(2^{0.3090M})$, which is a better bound when $M \leq 3N$.

In addition to this, there are of course a large number of results for restricted versions of the problem. Most notable of these restricted versions is k -SAT where

* The research is supported by CUGS – National Graduate School in Computer Science, Sweden.

a clause may have at most k literals. This is polynomial for $k = 2$ and NP-complete for $k > 2$ [8], and the best results for general k are a probabilistic algorithm with running time in $O((2 - 2/k)^N)$ [12] and a deterministic algorithm in time $O((2 - 2/(k + 1))^N)$ [3]. For $k = 3$, there are stronger results, with a probabilistic algorithm which runs in time $O(1.3238^N)$ [10] and a deterministic one which runs in time $O(1.473^N)$ [1].

In this paper, we increase the toolbox with an algorithm that has the strongest bound so far for formulae where the total length L (*i.e.* the total number of literals in the formula) is linear in N . Our algorithm is simple and deterministic, and with $k = L/N$ it runs in time $O(2^{N(1-c/(k+1)+O(1/k^2))})$ for a constant c , for any CNF formula (with the exception that no variables with a single occurrence are allowed when $L < 4N$). This result is an extension of the work in [14], where a running time of $O(2^{0.1736(L-2N)})$ is proven. The precise upper bounds for low k are $O(2^{0.1736N})$ when $k = 3$, $O(2^{0.3472N})$ when $k = 4$, $O(2^{0.4629N})$ when $k = 5$, and $O(2^{0.5408N})$ when $k = 6$; after this, the times follow the general pattern closely, with $c \approx 3$. The previously strongest bound in terms of l was by Hirsch, with an algorithm that runs in time $O(2^{0.1030L})$ [9]. Our new bounds are stronger than Hirsch's for every value of L and N , and especially so when L/N increases, as our bound tends to 2^N while Hirsch's keeps increasing. In terms of L alone, our analysis proves a bound of $O(2^{0.0926L})$ for any CNF formula.

The key to these results is the method of analysis, introduced in the #2SAT analysis in [2]. By analysing the behaviour of the algorithm in terms of both N and L , simultaneously, we are able to prove stronger bounds on the running time, even in terms of N or L alone, than what is easily achieved by a more classical analysis. This improvement is due to the extra information contained in the relative values of L and N , which allows us to distinguish cases with short and long formulae in the analysis. In the current algorithm, we use the algorithm from [14] for a strong bound when L is low (strongest when every variable occurs exactly three times, and $L = 3N$), and derive bounds progressively closer to $O(2^N)$ when L is longer, by measuring the speedup that is caused by having [14] as a base case (in essence, the lower the quotient L/N is, the faster we will reach a situation where the algorithm of [14] can be usefully applied).

In Section 2, we present some standard concepts used in the paper. In Section 3 we present the algorithm. Section 4 contains precise descriptions of the method of analysis and the upper bound on the running time expressed in L and N , and Section 5 contains the analysis where we prove that this bound on the running time is valid. Finally, Section 6 concludes the paper with some discussion about the results.

Due to the length restriction, some proofs have been omitted.

2 Preliminaries

A SAT instance is a boolean CNF formula F , with no restrictions on the clause lengths. A k -SAT instance is a SAT instance where no clause contains more than k literals; a clause with exactly k literals is a k -clause. The problem instances that

are considered in this paper belong to the general SAT class, without restrictions on clause lengths. $Vars(F)$ is the set of all variables that occur in F .

Regarding notation, if v is a variable then v and $\neg v$ are its positive and negative literals, respectively, and if l is the literal $\neg v$ then $\neg l$ is the literal v . $|C|$ for a clause C denotes the number of literals in C (also called the length of C), and a clause $(l \vee C)$ for some literal l and clause C is the clause that contains all literals of C plus the literal l . Similarly, $(l \vee C \vee D)$ for literal l and clauses C and D would be the clause containing l plus every literal occurring in C or D . Unless explicitly stated otherwise, the clauses C, D must be non-empty.

If l is a literal of a variable occurring in F , $F[l]$ is the formula one gets if every clause C in F that contains l , and every occurrence of $\neg l$ in other clauses, is removed. For a set A of literals, $F[A]$ is the result of performing the same set of operations for every literal $l' \in A$. Note that $F[l]$ and $F[A]$ may contain empty clauses.

For a variable $v \in Vars(F)$, define the *degree* $d(v, F)$ of v in F to be the number of occurrences of either v or $\neg v$ in the clauses of F . Usually, the formula is understood from the context, and $d(v)$ is used. $d(F)$ is the maximum degree of any $v \in Vars(F)$, and F is *k-regular* if $d(v) = k$ for every $v \in Vars(F)$. A variable v where v occurs in a clauses and $\neg v$ occurs in b clauses is an (a, b) -variable, in which case v is an a -literal and $\neg v$ a b -literal. If $b = 0$, then v is a *pure literal* (similarly, if $a = 0$ then $\neg v$ is a pure literal). We will usually assume, by symmetry, that $a \geq b$, so that *e.g.* a 1-literal is usually a negative literal $\neg v$. If $d(v) = 1$, then v is a *singleton*.

The *neighbours* of a literal l are all literals l' such that some clause C in F contains both l and l' . If a clause C contains a literal of both variables a and b , then a and b *co-occur* in C .

We write $L = L(F)$ for the length of F and $N = N(F)$ for the number of variables in F (*i.e.* $L(F) = \sum_{v \in Vars(F)} d(v, F)$ and $N(F) = |Vars(F)|$).

We use the classic concept of *resolution* [7]. For clauses $C = (a \vee l_1 \vee \dots \vee l_d)$ and $D = (\neg a \vee m_1 \vee \dots \vee m_e)$, the *resolvent* of C and D by a is the clause $(l_1 \vee \dots \vee l_d \vee m_1 \vee \dots \vee m_e)$, shortened to remove duplicate literals. If this new clause contains both v and $\neg v$ for some variable v , it is said to be a *trivial resolvent*.

For a formula F and a variable v occurring in F , $DP_v(F)$ is the formula where all non-trivial resolvents by v have been added to F and all clauses containing the variable v have been removed from F . Resolution is the process of creating $DP_v(F)$ from F .

We also use *backwards resolution*. If a formula F contains two clauses $C_1 = (C \vee D)$, $C_2 = (C \vee E)$ where D and E share no literals, DP_{C_1, C_2}^{-1} is the formula where C_1 and C_2 have been replaced by clauses $(\neg a \vee C)$, $(a \vee D)$, $(a \vee E)$ for a fresh variable a .

3 The Algorithm

The algorithm *MiddegSAT* used in this article is based on *LowdegSAT* from [14], modified to give better worst-case performance for formulae with $L > 4N$.

The algorithm is given in Figure 1. It is shown as a list of cases, where the earliest case that applies is used, *e.g.* case 8 is only used if none of cases 0–7 apply. Case 0 is a base case. Cases 1–7 are reductions, and case 8 is a branching. Case 9, finally, calls the algorithm from [14] when L/N is low enough.

Algorithm MiddegSAT(F)

Case 0: If $F = \emptyset$, return 1. If $\emptyset \in F$, return 0.

Case 1: If F is not in standard form (see text), standardise it, return $MiddegSAT(F)$.

Case 2: If there is some 1-clause $(l) \in F$, return $MiddegSAT(F[l])$.

Case 3: If there is a pure literal l in F , return $MiddegSAT(F[l])$.

Case 4: a) If there is a 2-clause $(l_1 \vee l_2)$ and a clause $D = (l_1 \vee \neg l_2 \vee C)$ in F for some possibly empty C , construct F' from F by deleting $\neg l_2$ from D and return $MiddegSAT(F')$.

b) If there are 2-clauses $C_1 = (l_1 \vee l_2)$ and $C_2 = (\neg l_1 \vee \neg l_2)$, create F' from F by replacing all occurrences of l_2 by $\neg l_1$ and all occurrences of $\neg l_2$ by l_1 , and removing C_1 and C_2 . Return $MiddegSAT(F')$.

Case 5: If there is a variable x in F with at most one non-trivial resolvent, return $MiddegSAT(DP_x(F))$.

Case 6: If there is a variable x in F with $d(x) = 3$ such that resolution on x is admissible (see def. 1), return $MiddegSAT(DP_x(F))$.

Case 7: If there are two clauses $C_1 = (C \vee D)$, $C_2 = (C \vee E)$ such that backwards resolution on C_1, C_2 is admissible (see def. 1), return $MiddegSAT(DP_{C_1, C_2}^{-1}(F))$.

Case 8: If $L(F) > 4N(F)$, pick a variable x of maximum degree. If some literal of x , assume $\neg x$, occurs only in a single clause $(\neg x \vee l_1 \vee \dots \vee l_k)$, return $MiddegSAT(F[x]) \vee MiddegSAT(F[\{\neg x, \neg l_1, \dots, \neg l_k\}])$. If both x and $\neg x$ occur in at least two clauses, return $MiddegSAT(F[x]) \vee MiddegSAT(F[\neg x])$.

Case 9: Otherwise, return $LowdegSAT(F)$.

Fig. 1. Algorithm for SAT when most variables have few occurrences

We say that a formula F' is the *step t -reduced version* of F if F' is the result of applying the reductions in cases 0– t , in the order in which they are listed, until no such reduction applies anymore. F' is called *step t -reduced* (without reference to F) if no reduction in case t or earlier applies (*i.e.* F is step t -reduced if $MiddegSAT(F)$ reaches case $t + 1$ without applying any reduction). A synonym to step 7-reduced is fully reduced.

Standardising a CNF formula F refers to applying the following reductions as far as possible:

1. Subsumption: If there are two clauses C, D in F , and if every literal in C also occurs in D , then D is *subsumed* by C . Remove D from F .
2. Trivial or duplicate clauses: If F contains several copies of some clause C , then C is a duplicate clause. If there is a clause C in F such that both literals v and $\neg v$ occur in C for some variable v , then C is a trivial clause. In both cases, remove C from F .
3. Multi-occurring literals: If there is a clause C in F where some literal l occurs more than once, remove all but one of the occurrences of l from C .

A formula F where none of these reductions apply is said to be in *standard form*.

Definition 1. Let F be a step 5-reduced CNF formula, and let F' be the step 5-reduced version of $DP_x(F)$, for some variable x in F . Let $k = \lceil L(F)/N(F) \rceil$, $\Delta L = L(F) - L(F')$ and $\Delta N = N(F) - N(F')$. We say that resolution on x in F is admissible if $k \leq 4$ and $\Delta L \geq 2\Delta N$, or if $k = 5$ and $\Delta L \geq \Delta N$, or if $k > 5$ and $\Delta L \geq 0$. For backwards resolution, if there are two clauses $C_1 = (C \vee D)$, $C_2 = (C \vee E)$ in F , let F' be the step 5-reduced version of $DP_{C_1, C_2}^{-1}(F)$. Backwards resolution on C_1, C_2 is admissible if $k \leq 4$ and $\Delta L > 2\Delta N$, or if $k = 5$ and $\Delta L > \Delta N$.

Once the function $f(F)$ is defined in Section 4, it will be clear that this definition guarantees that $f(F) \geq f(F')$ when resolution is admissible, and that $f(F) > f(F')$ when backwards resolution is admissible.

Lemma 1. *MiddegSAT(F) correctly determines the satisfiability of a CNF formula F.*

4 Method of Analysis

For the analysis of the worst-case running time of *MiddegSAT*, we use an adaptation of the method used for the #2SAT algorithm in [2]. At its core is the method of Kullmann [11], described below, using a measure $f(F)$ of the complexity of a formula F , but the construction of f is special enough that it deserves attention of its own. This measure is described in the rest of this section.

Regarding Kullmann’s method, in the form we use it here, let F be a fully reduced formula, and assume that *MiddegSAT* applied to F branches into formulas F_1, \dots, F_d . Let F'_i be the fully reduced version of F_i for $i = 1, \dots, d$. The *branching number* of this branching is $\tau(f(F) - f(F'_1), \dots, f(F) - f(F'_d))$, where $\tau(t_1, \dots, t_d)$ is the unique positive root to the equation $\sum_i x^{-t_i} = 1$. If α is the biggest branching number for all branchings that can occur in *MiddegSAT*, then the running time of the algorithm for a formula F is $O(\alpha^{f(F)})$. For a more detailed explanation, see Kullmann’s paper.

Normally, when using a measure such as $N(F)$ or $L(F)$, this is fairly straightforward: You locate the case of the algorithm that gets the highest branching number α , and thus prove that the running time is in $O(\alpha^N)$ or $O(\alpha^L)$. This worst-case branching would usually be independent of the value of the parameter; as long as $N(F)$ is large enough, the exact value of it contains very little information about the branchings that can occur.

However, in this paper, $f(F) = f(L(F), N(F))$ depends on both the total length and the number of variables in a formula, and the combination of $L(F)$ and $N(F)$ does contain information about the possible branchings: when $L(F) > k \cdot N(F)$, we know that at least one variable has at least $k + 1$ occurrences. Since we choose a variable of maximum degree in the algorithm, this means that we get not one global worst case, but a sequence of worst cases, depending on the degree we can guarantee. We want the bound $O(\alpha^{f(F)})$ for some constant α to be as tight as possible for every combination of large enough values for L and N . We also want the measure to be easy to use.

To get this, we let $f(L, N)$ be a piecewise linear function, so that for any worst-case situation, $f(F) - f(F') = a(N(F) - N(F')) + b(L(F) - L(F'))$, where the parameters a, b depend on the situation (with the natural restriction that f must be continuous). In essence, we define that every worst-case branching number for a formula F with L literals and N variables is 2, so that the running time for any F is in $O(2^{f(L(F), N(F))})$, and we then derive the proper measure f so that this holds.

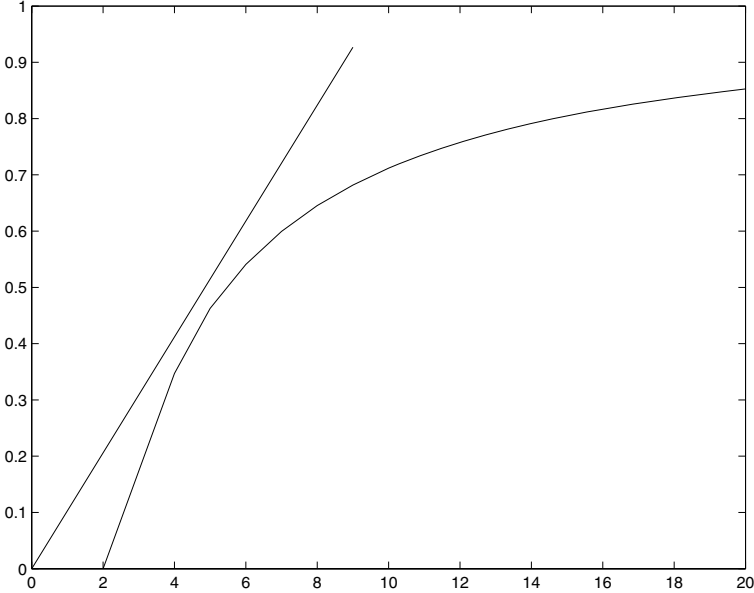


Fig. 2. Worst-case running time expressed as $O(2^{cN})$ depending on L/N (c on vertical axis, L/N on horizontal axis) for Hirsch’s algorithm (top line) and *MiddegSAT* (bottom line)

In this particular case, we use the algorithm *LowdegSAT* [14] to get a running time of $O(1.1279^{L-2N})$ when $L \leq 4N$, or $O(2^{0.1736N})$ when F is 3-regular and $O(2^{0.3472N})$ when $L = 4N$ (these bounds are proven in [14]), and in the rest of the analysis, the improvement over 2^N depends on the distance to this easy section. That is, defining the section of F to be $\lceil L(F)/N(F) \rceil$, when the section of F is k , the components of f are N and $L - (k - 1)N$, where the latter is the distance to the next lower (easier) section. The definitions are as follows:

$$f(L, N) = f_k(L, N) \text{ if } k - 1 \leq L/N \leq k \tag{1}$$

$$f_k(L, N) = \chi_{k-1}N + (L - (k - 1)N)b_k \tag{2}$$

$$\chi_0 = 0 \tag{3}$$

$$\chi_k = \chi_{k-1} + b_k \tag{4}$$

For convenience, $f(F) = f(L(F), N(F))$ for a formula F , and when branching from a formula F to a fully reduced formula F' , we sometimes write $\Delta f(F)$ for $f(F) - f(F')$ (and correspondingly for ΔL and ΔN).

The values of b_k are chosen so that the worst-case branching number for a fully reduced formula F with $\lceil L/N \rceil = k$ is 2, as mentioned. For $k \leq 4$, *LowdegSAT* from [14] is used, with a worst-case running time of $O(\alpha^{L(F)-2N(F)})$ where $\alpha = \tau(4, 8) \approx 1.1279$, so b_3 and b_4 must be chosen so that this coincides with $O(2^{f(F)})$. For $k \geq 5$, b_k are chosen according to the worst-case branchings identified in Section 5. However, the definitions of b_k are given here, before the analysis, to simplify the presentation. Note that no cyclic reference occurs, as these definitions are presented without reference to the running time. In other words, $f(F)$ is defined here, and the running time of $O(2^{f(F)})$ for a fully reduced formula F is proven in Section 5.

The definitions of b_k are as follows:

$$b_1 = b_2 = 0 \tag{5}$$

$$\tau(4b_3, 8b_3) = 2 \tag{6}$$

$$\tau(4b_4, 8b_4) = 2 \tag{7}$$

$$\tau(\chi_4 + 3b_5, 3\chi_4 + 3b_5) = 2 \tag{8}$$

$$\tau(\chi_{k-1} + 5b_k, \chi_{k-1} + (2k - 3)b_k) = 2 \text{ for } k \geq 6 \tag{9}$$

For b_3 to b_5 , there are analytical solutions. For b_k with $k \geq 6$, we will need numerical approximations. Both of these are given later in this section.

As stated, $f(L, N)$ is a piecewise linear function. Each section $(k - 1)N < L \leq kN$ is associated with a linear function f_k with a parameter b_k , and the worst-case running time for any F within the section is $O(2^{f(kN, N)}) = O(2^{\chi_k N})$. If a form $f_k(L, N) = a_k N + b_k L$ is desired, it can easily be derived from the definitions that $a_k = \chi_{k-1} - (k - 1)b_k$. We will proceed to give some limits on the size of the parameters, but first we need to state a lemma on the behaviour of the τ function.

Lemma 2. *If $0 < d < a \leq b$, $\tau(a - d, b + d) > \tau(a, b)$.*

Proof. This is a direct consequence of Lemma 8.5 of [11].

Table 1. Approximate values for the parameters in $f_k(L, N) = a_k N + b_k L$ and $\chi_k = \sum_{i=1}^k b_i$, and worst-case running time in each section

k	a_k	b_k	χ_k	Running time
3	-0.347121	0.173560	0.173560	$O(2^{0.1736N}) \approx O(1.1279^N)$
4	-0.347121	0.173560	0.347121	$O(2^{0.3472N}) \approx O(1.2721^N)$
5	-0.115707	0.115707	0.462828	$O(2^{0.4629N}) \approx O(1.3783^N)$
6	0.073130	0.077940	0.540768	$O(2^{0.5408N}) \approx O(1.4548^N)$
7	0.188505	0.058710	0.599478	$O(2^{0.5995N}) \approx O(1.5152^N)$
8	0.278738	0.045820	0.645298	$O(2^{0.6453N}) \approx O(1.5641^N)$
9	0.352328	0.036621	0.681920	$O(2^{0.6820N}) \approx O(1.6043^N)$
10	0.411685	0.030026	0.711946	$O(2^{0.7120N}) \approx O(1.6381^N)$

Lemma 3. *The following hold for the parameters a_k , b_k and χ_k .*

- $b_3 = b_4 = (\log_2(\sqrt{5} + 1) - 1)/4$ and $a_3 = a_4 = -2b_3$
- $b_5 = 2b_3/3$ and $a_5 = -b_5$
- For $k \geq 4$, $a_k < a_{k+1} < 1$, $b_k > b_{k+1} > 0$, $\chi_k < \chi_{k+1} < 1$, and $a_k > 0$ for $k \geq 6$.

The following lemma allows us to use Δf_k as a replacement for Δf when looking for worst cases.

Lemma 4. *If $L \geq 2N$, then $f(L, N) \leq f_k(L, N)$ for all k . Thus, if $\lceil L/N \rceil = k$, and if $L \geq 2N$ and $L' \geq 2N'$, then $f(L, N) - f(L', N') \geq f_k(L, N) - f_k(L', N')$.*

Next, we give the asymptotic growth of χ_k .

Lemma 5. $\chi_k = 1 - c/(k + 1) + O(1/k^2)$ for some c .

Finally, we show the relation between the bound $O(2^{\chi_k N})$ and a bound of the form $O(2^{\alpha L})$.

Lemma 6. $f(L, N) \leq 0.0926L$ for all L and N , with a maximum $f(L, N)/L$ value occurring when $L = 5N$.

Approximate values for the various variables, and the running time, for sections up to $k = 10$, are given in Table 1. The running times are also illustrated graphically in Figure 2, along with the previously best bound for a comparable algorithm, Hirsch's algorithm with an $O(2^{0.10297L})$ running time [9], included for comparison. The proof that $\text{MiddegSAT}(F)$ has a running time in $O(2^{f(F)})$ is in Section 5.

5 The Analysis

In this section, we analyse the running time of the algorithm, for increasing values of $\lceil L/N \rceil$ (*i.e.* increasing sections). More properly, we prove that every branching that can occur in MiddegSAT has a branching number of at most 2, when using the measure $f(L, N)$ defined in the previous section. In Section 5.1 we prove this for $L \leq 4N$ (which coincides with the LowdegSAT algorithm of [14]). In Section 5.2, we take care of the case $4N < L \leq 5N$, and in Section 5.3, we prove it for every $L > 5N$.

5.1 Up to Four Occurrences per Variable

Theorem 1 (*LowdegSAT*). *If F is a CNF formula with $L(F) \leq 4N(F)$ and no singletons, then $\text{LowdegSAT}(F)$ decides the satisfiability of F in time $O(2^{f(F)})$.*

Proof. Theorem 1 of [14] proves this for $O(1.1279^{L(F)-2N(F)})$, which coincides with $O(2^{f(F)})$ when $L(F) \leq 4N(F)$.

Lemma 7. *If F is a fully reduced CNF formula with N variables and L literals, where $L \leq 3N$, then $\text{MiddegSAT}(F)$ decides the satisfiability of F in time $O(2^{\chi_3 N})$. If $L \leq 4N$, the time is $O(2^{\chi_4 N})$.*

Proof. MiddegSAT applies algorithm LowdegSAT in this case, and the running time for LowdegSAT is known from Theorem 1.

5.2 Five Occurrences per Variable

In this section, $f_5(L, N) = a_5N + b_5L$ is used, with $a_5 = -b_5$ and $b_5 = 2/3 \cdot b_3 \approx 0.115707$. We will need to prove that $\tau(\chi_4 + 3b_5, 3\chi_4 + 3b_5) = 2$ is the worst-case branching.

Lemma 8. *If F is a step 5-reduced formula with $\lceil L/N \rceil = 5$, the following hold:*

1. *If there is a clause $(l \vee C)$ in F , where l is a 1-literal, the variable of l is of degree 3, and $|C| \leq 2$, then resolution on l is admissible.*
2. *If there are two clauses $(l_1 \vee l_2 \vee C)$ and $(l_1 \vee l_2 \vee D)$ in F , where l_1 or l_2 is a literal of a variable of degree 3, then backwards resolution on these clauses is admissible.*

Since $a_5 + b_5 = 0$, the only pitfall when evaluating branchings in this section is a variable that has all its occurrences within the clauses removed by an assignment x or $\neg x$ (or assignments $\neg x, \neg l_1, \dots, \neg l_d$ if $\neg x$ is a 1-literal). This limits the number of cases that we have to consider in the following result.

Lemma 9. *If F is a fully reduced CNF formula and $\lceil L/N \rceil = 5$, the worst-case branching number for MiddegSAT when applied to F is $\tau(\chi_4 + 3b_5, 3\chi_4 + 3b_5) = 2$.*

Proof. Let x be the variable we branch on ($d(x) \geq 5$). It can be realised that if there is a 2-clause $(x \vee l)$, then at least two literals of l occur in clauses without $\neg x$. That is, as a_5 and b_5 cancel each other out, a 2-clause contributes at least b_5 to both branches, while a 3-clause contributes $2b_5$ to one branch. By Lemma 2, a 2-clause only occurs in a worst case if it causes a higher imbalance in Δf .

Regarding the complications arising from having $a_5 < 0$, the only case where Δf could decrease due to an unexpectedly high ΔN is when a variable has all its occurrences among literals already accounted for, and thus disappears from F without an assignment. For most of this proof, this can only occur if some variable v , $d(v) \geq 4$, has all its occurrences with the literal x (or $\neg x$), which requires that x occurs in at least four 3-clauses, with a minimum Δf of $(d(x) + 8)b_5 + 3a_5$ in the branch x , while the minimum Δf when x is a 3-literal occurring in no 2-clauses is $(d(x) + 6)b_5 + a_5$. No new worst-case branchings are introduced by this case. When there are other cases where a variable can disappear, these are addressed specifically.

To start with, assume that $\neg x$ is a 1-literal. In the branch with assignment $\neg x$, x contributes at least $5b_5 + a_5$, and each neighbour of $\neg x$ with degree d contributes at least $db_5 + a_5$. Let C be the clause with $\neg x$. If $|C| \geq 3$, Δf in this branch is at least $8b_5$. Otherwise, assume that $C = (\neg x \vee y)$ and let D be a clause where $\neg y$ appears. D contains at least one literal not of variable x or y . If $d(y) = 3$, $|D| \geq 4$; otherwise, D may contain only one further literal. In either case, the minimum reduction in $f(F)$ is again $8b_5$, and no further variables can disappear without also increasing ΔL . x is at least a 4-literal, and as noted above, the minimum reduction is $10b_5$. $\tau(8b_5, 10b_5) = \tau((5 + 1/3)b_3, (6 + 2/3)b_3) < 2$.

Next, assume that $\neg x$ is a 2-literal. If $\neg x$ is involved in two 2-clauses, branch $\neg x$ will reduce $f(F)$ by at least $6b_5$, and branch x by at least $12b_5$, counting the contributions from the variables of the 2-clauses and the neighbours of x that do not appear in these 2-clauses. $\tau(6b_5, 12b_5) = \tau(4b_3, 8b_3) = 2$. If $\neg x$ is involved in one 2-clause, the branching number is $\tau(7b_5, 11b_5) < 2$, and with no 2-clauses, $\tau(8b_5, 10b_5) < 2$. Having $d(x) > 5$ will not yield any harder cases. Expressing the worst case in χ_4 and b_5 , we have $\tau(\chi_4 + 3b_5, 3\chi_4 + 3b_5) = 2$.

5.3 Six or More Occurrences per Variable

In all further sections, a_k and b_k are both positive, and the worst-case branchings all appear when there are no 2-clauses. For $L \leq 10N$, we need to prove this section by section. When $L > 10N$, we can give a general proof.

Lemma 10. *For a fully reduced CNF formula F , let $k = \lceil L/N \rceil$. If $k > 5$, then the worst-case branching number for MiddegSAT when applied to F is $\tau(\chi_{k-1} + 5b_k, \chi_{k-1} + (2k - 3)b_k) = 2$.*

Proof. Assume that we are branching on variable x , in section k , and $d(x) = k$. First, let $\neg x$ be a 2-literal, involved in zero, one or two 2-clauses. With no 2-clauses, we get a reduction of $(k + 4)b_k + a_k$ in branch $\neg x$, and $(3k - 4)b_k + a_k$ in branch x . With $a_k = \chi_{k-1} - (k - 1)b_k$, we get a branching of $(\chi_{k-1} + 5b_k, \chi_{k-1} + (2k - 3)b_k)$, which has a branching number of 2 by definition of b_k .

For a 2-clause $(\neg x \vee y)$, the statements in Lemma 9 still hold, and assignment y in branch x adds a further $2b_k + a_k$, independently of reductions due to other assignments and clauses containing x . We find that with $\neg x$ being involved in one 2-clause the branching is $(\chi_{k-1} + 4b_k, 2\chi_{k-1} + kb_k)$ and with two 2-clauses, $(\chi_{k-1} + 3b_k, 3\chi_{k-1} + 3b_k)$.

If $\neg x$ is instead a 1-literal, assume that the clause containing $\neg x$ is $(\neg x \vee l \vee C)$ for some possibly empty C , where l is a literal of the variable y . If $d(y) \geq 4$, the reduction from variables x and y alone in the $\neg x$ branch is at least $(k + 4)b_k + 2a_k$. If $d(y) = 3$, and l is a 2-literal, then the clause D where $\neg l$ appears is at least a 5-clause, with at least three literals not of variables x and y , for a reduction of at least $(k + 6)b_k + 2a_k$. If $d(y) = 3$ and l is a 1-literal, then $|C| \geq 3$ and counting only the assignments we have a reduction of at least $(k + 12)b_k + 5a_k$. In all of these cases, the reduction in the $\neg x$ branch is stronger than the $(k + 3)b_k + a_k$ we get from a 2-literal $\neg x$ not involved in any 2-clauses, and the reduction in the branch x is no weaker than in this case, as x must appear in at least one more clause.

We have three remaining branchings for each value of k . Note that $\tau(\chi_7, 3\chi_7) < 2$ already, so when $L > 7N$, the case with two 2-clauses can be excluded as a worst case. Similarly, $\tau(\chi_{10}, 2\chi_{10}) < 2$, so when $L > 10N$, we can see immediately that the case where $\neg x$ is a 2-literal and not involved in any 2-clauses is the worst case. For the sections up to those points, we need to evaluate the branching number case by case. Doing so, we find that $\tau(\chi_{k-1} + 4b_k, 2\chi_{k-1} + kb_k) < 2$ and $\tau(\chi_{k-1} + 3b_k, 3\chi_{k-1} + 3b_k) < 2$ for every $k \geq 6$. This concludes the proof.

Now that we know that the branching number is at most 2 for every section of L/N , we can give the general theorem.

Theorem 2. *If F is a CNF formula where either $L(F) \geq 4N(F)$ or F is free of singletons, then the running time of $\text{MiddegSAT}(F)$ is in $O(2^{f(F)})$, where f is the function defined in Section 4.*

Proof. For a fully reduced F , it follows from the lemmas in this section. If $L(F) < 4N(F)$ and F contains no singletons, it follows from Theorem 1. If $L(F) \geq 4N(F)$, we see that the process of applying the reductions never increases $f_k(F)$ (where k is the section that F belongs to).

Corollary 1. *The running time of $\text{MiddegSAT}(F)$, without any restrictions on F , is in $O(2^{0.0926L(F)})$ regardless of the value for $N(F)$.*

Proof. By Lemma 6 in Section 4, $f(L, N) \leq 0.0926L$ for all L and N . If F' is the step 3-reduced version of F , $L(F') \leq L(F)$, and by Theorem 2, the running time of $\text{MiddegSAT}(F')$ will be in $O(2^{f(F')})$, and by extension also in $O(2^{0.0926L(F)})$.

6 Conclusions

We have presented an algorithm for determining the satisfiability of a CNF formula F of length L and with N variables in time $O(2^{0.4629N})$ if $\lceil L/N \rceil = 5$, in time $O(2^{0.5408N})$ if $\lceil L/N \rceil = 6$, and in general in time $O(2^{N(1-c/(k+1)+O(1/k^2))})$ if $\lceil L/N \rceil = k$. This builds on our previous result, where an algorithm for the same problem, with the restriction that F contains no single-occurring variables, was presented, running in time $O(2^{0.1736N})$ when $\lceil L/N \rceil = 3$ and time $O(2^{0.3472N})$ when $\lceil L/N \rceil = 4$. We also showed the limit of $O(2^{0.0926L})$ for the running time for every value of L/N . These are all the strongest known bounds for this problem, improving on the previous bound $O(2^{0.1030L})$ by Hirsch [9].

These results provide further proof of the usefulness of our method of analysis, introduced in [2], where the running time of an algorithm is analysed in terms of two properties (here, N and L) in such a way that the upper bound on the total running time is stronger.

This paper does not address the topic of an algorithm with an upper bound on the running time characterised by N and M , but better than $O(2^N)$ (where M is the number of clauses in a formula). Something in this vein should be possible.

References

- [1] Tobias Brueggemann and Walter Kern. An improved deterministic local search algorithm for 3-SAT. *Theoretical Computer Science*, 329(1–3):303–313, 2004.
- [2] Vilhelm Dahllöf, Peter Jonsson, and Magnus Wahlström. Counting models for 2SAT and 3SAT formulae. *Theoretical Computer Science*, 332(1–3):265–291, 2005.

- [3] Evgeny Dantsin, Andreas Goerdt, Edward A. Hirsch, Ravi Kannan, Jon M. Kleinberg, Christos H. Papadimitriou, Prabhakar Raghavan, and Uwe Schöning. A deterministic $(2 - 2/(k+1))^n$ algorithm for k -SAT based on local search. *Theoretical Computer Science*, 289(1):69–83, 2002.
- [4] Evgeny Dantsin, Edward A. Hirsch, and Alexander Wolpert. Algorithms for SAT based on search in Hamming balls. In *Proceedings of the 21st Annual Symposium on Theoretical Aspects of Computer Science (STACS 2004)*, volume 2996 of *Lecture Notes in Computer Science*, pages 141–151. Springer, 2004.
- [5] Evgeny Dantsin and Alexander Wolpert. Derandomization of Schuler’s algorithm for SAT. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, pages 69–75, 2004.
- [6] Evgeny Dantsin and Alexander Wolpert. An improved upper bound for SAT. Technical Report TR05-030, Electronic Colloquium on Computational Complexity, 2005.
- [7] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.
- [8] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.
- [9] Edward A. Hirsch. New worst-case upper bounds for SAT. *Journal of Automated Reasoning*, 24(4):397–420, 2000.
- [10] Kazuo Iwama and Suguru Tamaki. Improved upper bounds for 3-SAT. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2004)*, page 328, 2004.
- [11] Oliver Kullmann. New methods for 3-SAT decision and worst-case analysis. *Theoretical Computer Science*, 223:1–72, 1999.
- [12] Uwe Schöning. A probabilistic algorithm for k -SAT based on limited local search and restart. *Algorithmica*, 32(4):615–623, 2002.
- [13] Rainer Schuler. An algorithm for the satisfiability problem of formulas in conjunctive normal form. *Journal of Algorithms*, 54(1):40–44, 2005.
- [14] Magnus Wahlström. Faster exact solving of SAT formulae with a low number of occurrences per variable. In *Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT-2005)*, 2005.

Linear-Time Enumeration of Isolated Cliques

Hiro Ito, Kazuo Iwama, and Tsuyoshi Osumi

Department of Communications and Computer Engineering, School of Informatics,
Kyoto University, Kyoto 606-8501, Japan
{itohiro, iwama, osumi}@kuis.kyoto-u.ac.jp

Abstract. For a given graph G of n vertices and m edges, a clique S of size k is said to be c -isolated if there are at most ck outgoing edges from S . It is shown that this parameter c is an interesting measure which governs the complexity of finding cliques. In particular, if c is a constant, then we can enumerate all c -isolated maximal cliques in linear time, and if $c = O(\log n)$, then we can enumerate all c -isolated maximal cliques in polynomial time. Note that there is a graph which has a superlinear number of c -isolated cliques if c is not a constant, and there is a graph which has a superpolynomial number of c -isolated cliques if $c = \omega(\log n)$. In this sense our algorithm is optimal for the linear-time and polynomial-time enumeration of c -isolated cliques.

1 Introduction

Clique-related problems are all hard: For example, finding a maximum clique is not only NP-hard but also inapproximable within a factor of $n^{(1-\epsilon)}$ [10]. The problem is W[1]-hard and hence it is probably not fixed-parameter tractable [6]. The problem is of course solvable if we can enumerate all maximal cliques, but there are too many maximal cliques in general [14]. The situation is much the same if we relax cliques to dense subgraphs [3,4,13]. Why is this so hard? One intuitive reason is that a boundary between inside and outside of a clique is not clear. Then, otherwise, the problem should be easier.

In this paper, we consider *isolated* cliques and *isolated* dense subgraphs. For a given graph G , a vertex subset S of size k (and also its induced subgraph $G(S)$) is said to be c -isolated if $G(S)$ is connected to its outside via at most ck edges. The number c is sometimes called the *isolation factor*. Apparently, the subgraph is more isolated if the isolation factor is smaller. Our main result in this paper shows that for a fixed constant c , we can enumerate all c -isolated maximal cliques (including a maximum one if any) in linear time.

The notion of isolation appears in several different contexts. For example, clustering Web pages into “communities” has been drawing a lot of research attention recently [8,9,11,15]. Among others, Flake, Lawrence, and Giles [8] define a community as a set of members (Web pages) such that each member has more links to other members than to non-members. If we use our definition (isolated cliques) for a community, then it has to be not only isolated from the outside but also tightly coupled inside. Presumably members in such a community have

some uniqueness in common, such as being engaged in an unusual (even criminal) business; it would help a lot in a certain situation if we can enumerate all such communities quickly.

Our Contribution. As mentioned earlier, the most important result in this paper is a linear-time enumeration of c -isolated cliques for a fixed constant c . Note that we can prove that there is a graph which has a superlinear number of c -isolated cliques if the isolation factor is not a constant and therefore the condition that the isolation factor be a constant is tight for linear-time enumeration. For a nonconstant c , our algorithm still works as it is but its time complexity increases, e.g., in polynomial time for $c = O(\log n)$. We also show that there is a graph which has a superpolynomial number of c -isolated cliques if $c = \omega(\log n)$. Thus the isolation factor can be regarded as a measure of the complexity for enumerating cliques.

For a dense subgraph, we consider what we call a *pseudo-clique* whose average and minimum degrees must be at least some designed values. Enumerating isolated pseudo-cliques can be done in polynomial time for a wide range of parameter values. We also show a fairly tight condition on the parameter values for the polynomial enumeration.

Related Work. Here is a brief review of clique-related results: Recall that the problem of finding a maximum clique is hard to approximate within a factor of $n^{1-\varepsilon}$ [10]. For heuristic algorithms, see [12]. For the enumeration of maximal cliques, [5] includes a nice survey. For the most recent results, see [17]. For clique covers and clique partitions, see [16]. The problem of finding dense subgraphs is also called the maximum edge subgraph problem. This problem has been also popular, but known upper bounds for its approximation factor are not small either, for instance, $O(n^{\frac{1}{3}} \log n)$ [3] and $(\frac{n}{2k} + \frac{1}{2})^2$ (k is the size of the subgraph) [4]. However, if the original graph is dense, then there is a PTAS for this problem [2]. The problems are in P for restricted subclasses like constant-degree graphs and bounded tree-width graphs.

2 Definitions and Summary of Results

For a graph $G = (V, E)$ such that $|V| = n$ and $|E| = m$, a *clique* is a subgraph S of G such that every pair of vertices in S has an edge between them. A clique S is said to be *c-isolated* if $|E(S, G - S)| < ck$, where $c \geq 1$, k is the number of vertices in S and $E(G_A, G_B)$ denotes the set of edges connecting the two subgraphs G_A and G_B directly, i.e., the set of edges (v_1, v_2) such that v_1 is in G_A and v_2 in G_B . Edges in $E(S, G - S)$ are called *outgoing* edges. For a vertex v , $d(v)$ denotes the degree of v and $N(v)$ denotes the set of vertices which are adjacent to v . $N[v] = N(v) \cup \{v\}$. Clearly $d(v) = |N(v)| = |N[v]| - 1$ for any $v \in V$. Sometimes, $N[v]$ also denotes the subgraph induced by $N[v]$. Unless otherwise stated, a clique in this paper means a maximal clique.

Theorem 1. *For a given graph G of n vertices and m edges and an integer $c \geq 1$, all c -isolated cliques can be enumerated in time $O(c^5 2^{2c} m)$.*

Corollary 1. *All c -isolated cliques can be enumerated in linear time if c is constant, and in polynomial time if $c = O(\log n)$.*

Theorem 2. *Suppose that $f(n)$ is an increasing function not bounded by any constant. Then there is a graph of n vertices and m edges for which the number of $f(n)$ -isolated cliques is super-linear in $n + m$. Furthermore if $f(n) = \omega(\log n)$, there is a graph of n vertices and m edges for which the number of $f(n)$ -isolated cliques is super-polynomial in $n + m$.*

Both proofs are given in Sec. 3. Note that the graph G may include cliques which are not c -isolated (not included in the output of the algorithm).

A *pseudo-clique* with average degree α and minimum degree β , denoted by $PC(\alpha, \beta)$, is a set $V' \subseteq V$ such that the subgraph induced by V' has an average degree of at least α and a minimum degree of at least β . We always use k for the size of the subgraph and thus a pseudo clique is given like $PC(0.9k, 0.5k)$. The c -isolation condition is the same as before, i.e., the number of outgoing edges must be less than ck . We assume the isolation factor is always a constant for pseudo-cliques. Also a pseudo-clique usually means a maximal pseudo-clique.

Theorem 3. *Suppose that $f(n)$ is an increasing function not bounded by any constant and $0 < \varepsilon < 1$ is a constant. Then there are two graphs G_A and G_B of n vertices such that the number of 1-isolated $PC(k - f(k), k^\varepsilon)$ for G_A and the number of 1-isolated $PC(k - k^\varepsilon, \frac{k}{f(k)})$ for G_B are both super-polynomial.*

If we use a bit stronger condition for α of G_A and β of G_B , then it is not hard to show a polynomial-time enumeration: If we have $\beta = c_1 k$ for a constant $c_1 (< 1)$, then we have a polynomial-time enumeration for any α (but at least $c_1 k$ of course). A polynomial-time enumeration is also possible if $\alpha = k - c_2$ for a constant $c_2 (\geq 1)$ and $\beta = k^\varepsilon$. Notice that the restrictions for β of G_A and α of G_B are rather weak, which requires the other parameter to be almost equal to k for polynomial-time enumeration. Then what happens if the restrictions for α and β are both fairly strong? As the following theorems show, polynomial-time enumeration becomes possible for a wider range of parameters.

Theorem 4. *For any $\varepsilon > 0$ there is a graph of n vertices for which the number of 1-isolated $PC\left(k - (\log k)^{1+\varepsilon}, \frac{k}{(\log k)^{1+\varepsilon}}\right)$ is super-polynomial.*

Theorem 5. *There is a polynomial-time algorithm which enumerates all c -isolated $PC(k - \log k, \frac{k}{\log k})$.*

Proofs of Theorems 3–5 are omitted here from the space limitation.

3 Enumeration of Isolated Cliques

3.1 Basic Ideas

Before giving the proof of Theorem 1, we describe the basic ideas behind our algorithm. The following lemma is easy but important.

Lemma 1. *For any c -isolated clique C , there is a vertex (called a pivot) in the clique which has less than c edges outgoing from C . (obvious from the definition of c -isolation.)*

For notational simplicity let \check{c} be the maximum integer less than c , i.e., $\check{c} = c - 1$ if c is an integer and $\check{c} = \lfloor c \rfloor$ otherwise. If we can spend polynomial (instead of linear) time, then the proof is easy: By Lemma 1 it is enough to examine, for each vertex v , whether or not v is a pivot. To do so, we consider every set $C(v) \subseteq N[v]$ such that $|N[v]| - |C(v)| \leq \check{c}$, and examine whether $C(v)$ induces a c -isolated clique. Note that the number of such $C(v)$'s is polynomial since $\check{c} \leq c$ is a constant.

Our linear time algorithm also adopts the same strategy, i.e., we scan each vertex v_i and enumerate all c -isolated cliques whose pivots are v_i . Namely our goal is to enumerate all c -isolated cliques by removing at most $c - 1$ vertices from $N(v_i)$. We call this subroutine PIVOT(v_i). In order to make the algorithm run in linear time, we have to overcome two major difficulties:

(i) **How to delete \check{c} vertices:** The above naive algorithm does this blindly, but in fact, we can enumerate all maximal cliques that are obtained by removing at most constant number of vertices, in linear time, by using an FTP technique for the vertex-cover problem.

(ii) **How to save vertex scan:** It is important how many times the adjacency list of each vertex is scanned, which often dominates the computation time. Although our algorithm may scan $\omega(1)$ times for some vertices, they can be amortized to be $O(1)$ on average. For this purpose, we remove apparently useless vertices from $N[v_i]$, by using only degree information, before scanning their adjacency lists.

PIVOT(v_i) consists of the following three stages (see Section 3.2 for details).

- **Trimming stage (T-stage):** We remove vertices which are apparently unqualified for components of desired c -isolated cliques. In this stage, (i) we first trim vertices by using degree data only ($O(1)$ time for every vertex), and then (ii) scan adjacency lists ($O(d(v))$ time for every vertex v). During this stage, we exit from PIVOT(v_i), whenever we find out that v_i never be a pivot, for several reasons like more than \check{c} vertices have been removed. After the first trimming, each of the remaining vertices has $O(cd(v_i))$ degree; and the set of vertices having at most h th lowest indices in the remaining vertices of $N[v_i]$ has $O(c^3 h)$ outgoing edges. These conditions are crucial to guarantee that each adjacency list is scanned $O(1)$ (amortized) times in the entire algorithm.
- **Enumerating stage (E-stage):** We enumerate all maximal cliques by removing at most \check{c} vertices except v_i from the remaining vertex set, and obtain at most a constant number of cliques.
- **Screening stage (S-stage):** We test all maximal cliques obtained above and select legal c -isolated cliques. For checking maximality of them, some c -isolated cliques that have been already found must be used. However, we will see that the number of such c -isolated cliques is also constant and thus linearity of the algorithm is not damaged.

3.2 Proof of Theorem 1

For a given graph $G = (V, E)$, we assume that all vertices are sorted by their degrees as $d(v_1) \leq d(v_2) \leq \dots \leq d(v_n)$. We also assume that the adjacency list of each vertex v_i is sorted by the indices of the adjacent vertices of v_i in an increasing order, i.e., the adjacent vertices of v_i appear in an increasing order of their indices by tracing the adjacency list of v_i . Note that if a given graph does not satisfy these conditions, we can make the graph satisfy them in linear time by using the bucket sort. Note that a c -isolated clique may have more than one pivots. However by introducing the above assumption, we can restrict the pivot to be the vertex that has the minimum index in the clique.

Let $k = |N[v_i]|$ for notational simplicity. Let $N^-(v_i) \subseteq N(v_i)$ be the set of vertices with indices lower than i in $N(v_i)$. Since the pivot has the minimum index in the clique, $N^-(v_i)$ is removed from the candidate of elements of c -isolated cliques whose pivot is v_i . Moreover from Lemma 1, if $|N^-(v_i)| > \check{c}$, then v_i cannot be a pivot of any c -isolated clique, and it can be skipped, i.e., we immediately exit from PIVOT(v_i).

In the following, we always use $C = \{v_{i_1}, v_{i_2}, \dots, v_{i_{k'}}\}$ ($k' \leq k$) to denote the set of candidates for vertices of c -isolated cliques whose pivots are v_i , where $i = i_1 < i_2 < \dots < i_{k'}$. Let $C^h \subseteq C$ be $\{v_{i_1}, v_{i_2}, \dots, v_{i_h}\}$ for $h \leq k'$. We start from $C = N[v_i] - N^-(v_i)$. We can still remove at most $c' = \check{c} - (k - k')$ vertices from C for a possible c -isolated clique.

Trimming Stage: We can remove vertices that violate one of the following conditions without losing any c -isolated cliques:

- (a) $d(v_{i_j}) < (c + 1)k'$,
- (b) $v_{i_j} \in C$ has less than ck' edges outgoing from C ,
- (c) $v_{i_j} \in C$ has at least $k' - c' (= k - \check{c})$ adjacent vertices in C .

The necessity of (b) is obvious from the definition of c -isolated cliques. The necessity of (c) is obtained from that we want to get cliques by deleting at most $c' - 1$ vertices from C . Condition (a) is implied from (b), and thus test (a) may seem meaningless. However, it is important since test (a) can be done by using only the value of degrees, i.e., it is done in $O(1)$ time for each vertex, and all remaining vertices after applying test (a) have degree $O(ck')$. Thus we apply test (a) first for all vertices in C .

Furthermore, we should not apply tests (b) and (c) hastily. Because C^h may have $\Omega(k'h)$ (i.e., $\Omega(k')$ per vertex) outgoing edges even if it satisfies (a)–(c). We want to limit the number of outgoing edges by $O(h)$ ($O(1)$ per vertex), which is again important to guarantee the linear computation time. We can delete at most \check{c} vertices, each of which may have less than ck' outgoing edges. Thus we can estimate that the upper-bound of the number of outgoing edges is $\check{c}ck' + ck' = c(\check{c} + 1)k' < c(c + 1)k' = O(k')$. Thus we introduce the following test.

- (d) $\sum_{v_{i_j} \in C} d(v_{i_j}) < c(c + 1)k'$.

Thus after test (a), we apply test (d), and then apply tests (b) and (c) for $v_{i_2}, v_{i_3}, \dots, v_{i_{k'}}$ in this order. If the number of deleted vertices becomes c , then we break PIVOT(v_i), since v_i cannot be a pivot.

Enumerating Stage: Now, $C = \{v_{i_1}, v_{i_2}, \dots, v_{i_{k'}}\}$ is the set of vertices remaining after the above tests, where $i = i_1 < i_2 < \dots < i_{k'}$ ($k - c < k' \leq k$). We try to find (maximal) cliques by removing at most $c' = \check{c} - (k - k') < c$ vertices from C . We can observe the next:

(A) C has $\Omega(k'(k' - c))$ edges.

This means that C is “dense,” and hence we can construct the complement graph \overline{C} of C in $O(\|C\| + ck')$ time, where $\|C\|$ is the number of edges in C . Here is a fundamental graph property:

(B) If $C' \subseteq C$ is a clique, $C - C'$ is a vertex-cover of \overline{C} .

From the property (B), enumerating maximal cliques $C' \subseteq C$ that can be obtained by removing at most \check{c} vertices is equivalent to enumerating minimal vertex-covers with at most \check{c} vertices in \overline{C} . Moreover, enumerating vertex-cover with at most \check{c} vertices in \overline{C} can be done in $O((1.39)^{\check{c}}\check{c}^2 + \|\overline{C}\|) = O((1.39)^c c^2 + ck' + \|C\|)$ time (e.g. see, [7]). Note that v_i must not be removed, since we are looking for c -isolated cliques whose pivots are v_i .

Screening Stage: Let \mathcal{Q}_i be the set of obtained cliques in the enumerating stage. We should check for each clique in \mathcal{Q}_i whether it satisfies the condition of a c -isolated clique and maximality. Checking c -isolation is trivial, since the number of cliques in \mathcal{Q}_i is constant (at most $2^{\check{c}}$). For checking maximality, it may seem that we must compare every $C' \in \mathcal{Q}_i$ with all other $C'' \in \mathcal{Q}_j$ ($j \leq i$) which have been already obtained. Fortunately, we do not need such brute force comparison, since we have the following property (Let $N^-(v_i) = \{v_{i'_1}, v_{i'_2}, \dots, v_{i'_p}\}$):

(C) $C' \in \mathcal{Q}_i$ is not included in $C'' \in \mathcal{Q}_j$ if $j \notin \{i'_1, i'_2, \dots, i'_p\}$.

The proof is straightforward since $C' \in \mathcal{Q}_i$ has v_i but $C'' \in \mathcal{Q}_j$ ($j \notin \{i'_1, i'_2, \dots, i'_p\}$) does not. From the property (C), it is enough to compare $C' \in \mathcal{Q}_i$ with $C'' \in \mathcal{Q}_j$ ($j \in \{i'_1, i'_2, \dots, i'_p\}$). The number of the latter cliques is less than $\check{c}2^{\check{c}}$ (note that $p = |N^-(v_i)| \leq \check{c}$), and hence the comparison can be done in $O(c2^{2\check{c}}\|C\|)$ time.

Now, a pseudo-code of this algorithm is shown as Table 1.

Estimation of Computation Time: First, we investigate the computation time of the trimming stage. Tests (a) and (d) uses $O(1)$ time for every $v_{i_j} \in C \subseteq N[v_i]$, i.e., $O(d(v_i))$ time for every PIVOT(v_i), and hence tests (a) and (d) requires at most linear time in the entire algorithm.

Tests (b) and (c) are more complicated, since they require scanning adjacency lists of $v_{i_j} \in C$. Let $C(v_i)$ be the set of vertices remaining after test (a). For estimating how many times we need the scan, we must note that tests (b) and

Table 1. The algorithm which enumerates all c -isolated cliques

```

procedure I-CLIQUE( $c, G$ )
1   Sort all vertices by their degrees and renumber their indices;
2   Sort their adjacency lists by their indices;
3   for  $i := 1$  to  $n$  do; call PIVOT( $v_i$ ); end for;
end procedure;

procedure PIVOT( $v_i$ )
11  if  $|N^-(v_i)| \leq \check{c}$  then
12    Construct  $C = N[v_i] - N^-(v_i) = \{v_{i_1}, \dots, v_{i_k}\}$  ( $i = i_1 < \dots < i_k$ );
13 T-stage  $\forall v_{i_j} \in C$ , apply test (a) and remove it it violates the condition;
14 if  $C$  doesn't satisfy (d) then return;
15 for  $j := 2$  to  $k$  do
16   if  $v_{i_j}$  breaks conditions (b) or (c) then
17     Remove  $v_{i_j}$  from  $C$ ;
18     if  $|C| < k - \check{c}$  then return;
19   end if;
20 end for;
21 E-stage Construct the complement graph  $\overline{C}$  of  $C$ ;
22 Enumerate all vertex covers  $C'' \subseteq C$  of  $\overline{C}$  with  $|C''| \leq c'$ ;
23 comment  $c' = \check{c} - (d(v_i) - |C|)$ ;
24  $\mathcal{Q}_i := \{\text{clique } C' = C - C'' \mid C'' \text{ is obtained in Line 15}\}$ ;
25 S-stage Check all cliques in  $\mathcal{Q}_i$  and remove cliques which don't satisfy the
26 condition of  $c$ -isolation, from  $\mathcal{Q}_i$ ;
27 Compare  $\forall C, C' \in \mathcal{Q}_i$  and remove not maximal ones from  $\mathcal{Q}_i$ 
28 for  $j := 1$  to  $p$  do comment  $N^-(v_i) = \{v_{i_1}, v_{i'_2}, \dots, v_{i_p}\}$ ;
29   Compare all elements in  $\mathcal{Q}_i$  with  $\mathcal{Q}_{i'_j}$  and remove not maximal
30   ones from  $\mathcal{Q}_i$ 
31 end for;
32 Output all elements in  $\mathcal{Q}_i$ ;
33 end if;
34 return;
end procedure;

```

(c) may be broken in the middle of the for-loop at Line 18. Let $P(v_i)$ be the set of vertices to which tests (a) and (b) are applied. $P(v_i)$ can be divided into the following two subsets: $P_g(v_i) \subseteq P(v_i)$ (reps., $P_b(v_i) \subseteq P(v_i)$) is the set of **good** vertices which pass the tests (b) and (c) (resp., **bad** vertices which are rejected by tests (b) or (c)). Clearly $|P_b(v_i)| \leq \check{c}$. Furthermore, we let $P_p(v_i) \subseteq P_g(v_i)$ be the set of vertices having 1st, 2nd, ..., and (2c)th smallest indices in $P_g(v_i)$. For notational simplicity, we sometimes express as $C(v_i) = C$, $P_p(v_i) = P_b$, $P_b(v_i) = P_b$, $|C(v_i)| = k'$, $|P_g(v_i)| = g$, and $|P_b(v_i)| = b$. Fig. 1 illustrates the relation of them (circles and crosses represent vertices in V , and crosses mean vertices that are not adjacent to v_i or removed before tests (b) and (c)).

Clearly for every $v_{i_j} \in P(v_i)$, its adjacency list is scanned constant times, and for the other vertices $v \notin P(v_i)$ their adjacency lists are never scanned, in PIVOT(v_i). Let P^{-1} be the reverse function of P , i.e., $P^{-1}(v_j) = \{v_i \mid v_j \in$

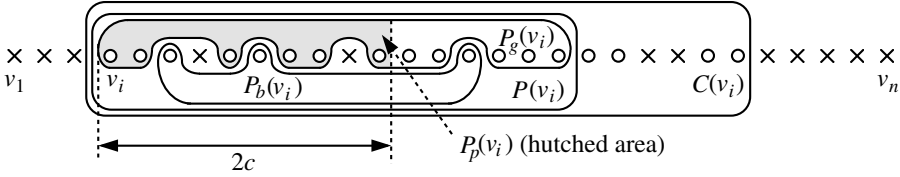


Fig. 1. Relation of $C(v_i)$, $P(v_i)$, $P_g(v_i)$, $P_b(v_i)$, and $P_p(v_i)$

$P(v_i)$ }, and $D(v_j) = |P^{-1}(v_j)|$. Namely, $D(v_j)$ shows how many times v_j appears in $P(v_1), P(v_2), \dots, P(v_n)$. Thus the value of

$$D = \sum_{i=1}^n \sum_{v_j \in P(v_i)} d(v_j) = \sum_{j=1}^n D(v_j)d(v_j) \tag{1}$$

expresses the computation time required for tests (b) and (c).

Next, we estimate computation time of E-stage and S-stage. As mentioned before, they require $O(c2^{2c}||P_g|| + 1.39c^2 + cg + ||P_g||)$ time for each PIVOT(v_i), where $||P_g||$ means the number of edges of the induced subgraph by P_g .

Here, we also let $P_g^{-1}(v_j) = \{v_i \mid v_j \in P_g(v_i)\}$, $D_g(v_j) = |P_g^{-1}(v_j)|$, and

$$D_g = \sum_{i=1}^n \sum_{v_j \in P_g(v_i)} d(v_j) = \sum_{j=1}^n D_g(v_j)d(v_j). \tag{2}$$

Clearly $||P_g|| \leq D_g$, and hence the computation time required in E-stage and T-stage is $O(c2^{2c}D_g)$ through the entire algorithm. Thus together with (1), we obtain the following lemma.

Lemma 2. *The computation time of I-CLIQUE(c, G) is $O(D + c2^{2c}D_g)$. (Obvious from the above discussion.)* □

We first compute the value of D .

Lemma 3. *For a vertex $v_i \in V$ and a pair of vertices $v_j, v_{j'} \in P(v_i)$, we have $d(v_{j'}) \leq (c + 1)d(v_j)$ and $d(v_j) \leq (c + 1)d(v_{j'})$. (Obvious from that v_j and $v_{j'}$ passed test (a).)*

Lemma 4. $D < c(c + 1)m + D_g$.

Proof:

$$D = \sum_{j=1}^n D(v_j)d(v_j) = \sum_{i=1}^n \sum_{v_j \in P(v_i)} d(v_j) = \sum_{i=1}^n \left(\sum_{v_j \in P_g(v_i)} d(v_j) + \sum_{v_j \in P_b(v_i)} d(v_j) \right).$$

From Lemma 3 and the fact that $|P_b(v_i)| < c$, we have $\sum_{v_j \in P_b(v_i)} d(v_j) < c(c + 1)d(v_i)$. Thus

$$D = \sum_{j=1}^n D(v_j)d(v_j) < \sum_{i=1}^n \sum_{v_j \in P_g(v_i)} d(v_j) + c(c + 1) \sum_{i=1}^n d(v_i) = D_g + c(c + 1)m.$$

□

We next compute the value of D_g . The next lemma shows how many edges are outgoing from $P_g(v_i)$.

Lemma 5. $|E(P_g(v_i), V - C(v_i))| < c(2c + 3) \max\{|P_g(v_i)|, c\}$.

Proof: If $k' < 2c$, from condition (d) we obviously obtain $|E(P_g(v_i), V - C(v_i))| < 2c^2(c + 1) < c^2(2c + 3)$, which satisfies the desired inequality. Thus we suppose $k' \geq 2c$ and $|E(P_g, V - C)| \geq c(2c + 3)g$. Then by considering the condition (c), $\sum_{v_j \in P_g} d(v_j) \geq g(k' - c) + c(2c + 3)g = g\{k' + 2c(c + 1)\}$, and hence $d(v_{i_g}) \geq k' + 2c(c + 1)$, where v_{i_g} is the vertex having the largest index in P_g . This implies that each of $v_j \in C - P_g$ has also degree at least $k' + 2c(c + 1)$, i.e., has at least $k' + 2c(c + 1) - (k' - 1) = k' + 2c^2 + 2c + 1$ edges outgoing from C . Thus

$$\begin{aligned} |E(C, V - C)| &\geq |E(C - P, V - C)| + |E(P_g, V - C)| \\ &\geq (k' - g - b)(2c^2 + 2c + 1) + g(k' + 2c^2 + 2c) \\ &= 2c(c + 1)(k' - b) + g(k' - 1) + (k' - b) \geq 2c(c + 1)(k' - c) \\ &\geq 2c(c + 1)\frac{k'}{2} \quad (\text{since } k' \geq 2c) \\ &= c(c + 1)k', \end{aligned}$$

contradicting the condition (d). \square

Lemma 6. $\sum_{v_j \in P_g(v_i)} D_g(v_j) < 2c^2(c + 3)|P_g(v_i)|$.

Proof:

$$\begin{aligned} D_g(v_i) = |P_g^{-1}(v_j)| &\leq |P_g^{-1}(v_j) \cap (C(v_i) - P(v_i))| + |P_g^{-1}(v_j) \cap P_g(v_i)| \\ &\quad + |P_g^{-1}(v_j) \cap P_b(v_i)| + |P_g^{-1}(v_j) \cap (V - C(v_i))|. \end{aligned}$$

The values of the four terms are computed as follows.

$P_g^{-1}(v_j) \cap (C(v_i) - P(v_i))$: All vertices $v_{j'}$ in $C(v_i) - P(v_i)$ have larger indices than any v_j in $P(v_i)$. The pivot has the smallest index in the c -isolated clique, and thus any vertex $v_{j'}$ in $C(v_i) - P(v_i)$ cannot be a pivot of a c -isolated clique including any v_j in $P(v_i)$. Therefore $|P_g^{-1}(v_j) \cap (C(v_i) - P(v_i))| = 0$.

$P_g^{-1}(v_j) \cap P_g(v_i)$: All vertices in $P_g(v_i)$ passed test (c) and has at least $k - c$ adjacent vertices in $N[v_i]$. Moreover, if v has more than c adjacent vertices with indices lower than v 's index, it is skipped at Line 11. From this we can observe that only vertices in $P_p(v_i)$ can be pivots. Therefore $|P_g^{-1}(v_j) \cap P_g(v_i)| \leq 2c$.

$P_g^{-1}(v_j) \cap P_b(v_i)$: $|P_b(v_i)| \leq c$, which obviously means $|P_g^{-1}(v_j) \cap P_b(v_i)| \leq c$.

$P_g^{-1}(v_j) \cap (V - C(v_i))$: From Lemma 5,

$$\sum_{j=1}^n |P_g^{-1}(v_j) \cap (V - C(v_i))| < c(2c + 3) \max\{|P_g(v_i)|, c\}.$$

Thus we obtain: $\sum_{v_j \in P_g(v_i)} D_g(v_j) < (2c + c)|P_g(v_i)| + c(2c + 3) \max\{|P_g(v_i)|, c\} \leq 2c(c + 3) \max\{|P_g(v_i)|, c\} \leq 2c^2(c + 3)|P_g(v_i)|$. \square

We now introduce $S = \{v_j \mid D_g(v_j) \leq 4c^2(c + 3)\}$. The following property holds on S . (Notice that $4c^2(c + 3)$ is the twice the value of the coefficient of the right-hand side of the inequality in Lemma 6.)

Lemma 7. For any $v_i \in V$, at least a half of elements in $P_g(v_i)$ are in S .

Proof: Assume that less than a half of $P_g(v_i)$ are in S for a vertex v_i , i.e., more than a half of vertices v_j in $P_g(v_i)$ has $D_g(v_j)$ value greater than $4c^2(c+3)$. Thus $\sum_{v_j \in P_g(v_i)} D_g(v_j) > 2c^2(c+3)|P_g(v_i)|$, contradicting Lemma 6. \square

Now we are ready to bound the value of D_g .

Lemma 8. $D_g \leq 4c^2(c+2)(c+3)m$

Proof:

$$\begin{aligned} D_g &= \sum_{j=1}^n D_g(v_j)d(v_j) = \sum_{i=1}^n \sum_{v_j \in P_g(v_i)} d(v_j) \\ &\leq \sum_{i=1}^n \sum_{v_j \in P_g(v_i) \cap S} (c+2)d(v_j) \quad (\text{since Lemmas 3 and 7}) \\ &= (c+2) \sum_{s \in S} D_g(s)d(s) \leq (c+2)\{4c^2(c+3)\} \sum_{s \in S} d(s) \quad (\text{since Lemma 7}) \\ &\leq 4c^2(c+2)(c+3)m. \end{aligned}$$

\square

From Lemmas 2, 4, and 8, the time complexity of I-CLIQUE(c, G) is

$$\begin{aligned} O(D + c^{2^2c}D_g) &= O(c(c+1)m + (1 + c^{2^{2c}})D_g) \\ &= O((1 + c^{2^{2c}})4c^2(c+2)(c+3)m) = O(c^5 2^{2^2c}m). \end{aligned}$$

Therefore, Theorem 1 is proved.

3.3 Proof of Theorem 2

Let $f(n)$, or simply f , be an arbitrary unbounded increasing function of n . We consider the following graph G_f of n vertices. G_f consists of a number of connected components, which are called *blocks*. All blocks are isomorphic, and each block has f vertices (thus the number of blocks is n/f). Let x and y be non-decreasing functions of n such that $f = xy$. Each block (see Fig. 2) is a complete y -partite graph, where each part $(X_i, i = 1, \dots, y)$ consists of x vertices.

We can easily verify that the following vertex set C is a maximum clique:

$$C = \{x_1, x_2, \dots, x_y \mid x_1 \in X_1, x_2 \in X_2, \dots, x_y \in X_y\}.$$

C consists of $k = y$ vertices. The number of edges between C and $V - C$ is

$$(x-1)(y-1)y \leq (xy)y = fk.$$

Thus C is an f -isolated clique. The number of such C 's in a block is $Q = x^y$, and the number of edges in a block is $m < \frac{1}{2}(xy)^2 = \frac{1}{2}f^2$. Thus the ratio of the number of f -isolated cliques to the input data size of G_f is

$$\frac{Q}{m} > \frac{2x^y}{f^2}. \tag{3}$$

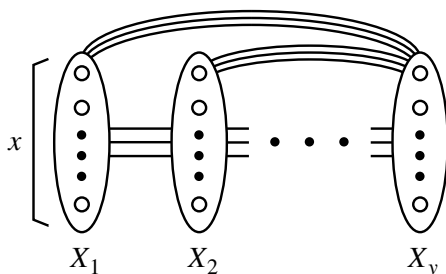


Fig. 2. A block of the graph G_f in the proof of Theorem 2

Since f is an unbounded increasing function, we can make (3) be an increasing function, i.e., Q is superliner by letting $x = y = \sqrt{f}$. Furthermore, if $f = \omega(\log n)$, by letting $y = \log n$ and $x = f/\log n$ we can make $Q/m \geq x^{\log n}/f^2$ be superpolynomial. Therefore Theorem 2 is proved.

4 Concluding Remarks

It is probably possible to improve our enumeration algorithm so that it can save space usage. To this end, we need to assume that we can make a random access to the input data and also need to avoid sorting. We have started experiments although they are preliminary at this moment. Our test data is so-called the inter-site graph consisting of some 6M Web pages and (much more) links among them. We have found 6974 1-isolated cliques of size from 2 to 32 fairly quickly.

References

1. J. Abello, A. L. Buchsbaum, and J. R. Westbrook: A functional approach to external graph algorithms, *Algorithmica*, 32 (2002) 437–458.
2. S. Arora, D. R. Karger and M. Karpinski: Polynomial time approximation schemes for dense instances of NP-hard problems, In *Proceedings of the 27th ACM Symposium on Theory of Computing (1995)* 284–293.
3. Y. Asahiro, R. Hassin and K. Iwama: Complexity of finding dense subgraph, *Discrete Applied Mathematics*, 121, Issue 1-3 (2002) 15–26.
4. Y. Asahiro, K. Iwama, H. Tamaki, and T. Tokuyama: Greedily finding a dense subgraph, *J. Algorithms*, 34 (2000) 203–221.
5. I. Bomze: The Maximum Clique Problem, *Handbook of Combinatorial Optimization (Supplement Volume A)*, Kluwer, 1999.
6. R. Downey, M. Fellows: Fixed-Parameter Tractability and Completeness II: On Completeness for $W[1]$. *Theor. Comput. Sci.* 141(1 & 2) (1995) 109-131.
7. R. G. Downey and M. R. Fellows: *Parametrized Complexity*, Springer, 1999.
8. G. W. Flake, S. Lawrence, and C. L. Giles: Efficient identification of web communities, In *Proceedings of the Sixth International Conference on Knowledge Discovery and Data Mining (ACM SIGKDD-2000)*, 150–160, Boston, MA, ACM Press (2000).
9. D. Gibson, J. M. Kleinberg and P. Raghavan: Inferring web communities from link topology, *UK Conference on Hypertext (1998)* 225–234.

10. J. Håstad: Clique is hard to approximate within $n^{1-\epsilon}$, *Acta Mathematica* 182 (1999) 105–142.
11. X. He, H. Zha, C. Ding, and H. Simon: Web document clustering using hyperlink structures, Tech. Rep. CSE-01-006, Department of Computer Science and Engineering, Pennsylvania State University (2001).
12. D. Johnson and M. Trick (Eds.): Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 26, American Mathematical Society (1996).
13. G. Kortsarz and D. Peleg: On choosing a dense subgraph, In *Proceedings of the 34th Annual IEEE Symposium on Foundation of Computer Science* (1993) 692–701.
14. J. Moon and L. Moser: On cliques in graphs, *Israel Journal of Mathematics*, 3 (1965) 23–28.
15. P. K. Reddy and M. Kitsuregawa: An approach to relate the web communities through bipartite graphs, In *Proceedings of The Second International Conference on Web Information Systems Engineering* (2001) 301–310.
16. H. U. Simon: On approximate solutions for combinatorial optimization problems, *SIAM J. Disc. Math.* 3 (1990) 294–310.
17. K. Makino and T. Uno: New algorithms for enumerating all maximal cliques, *Proceedings of the 9th Scandinavian Workshop on Algorithm Theory*, LNCS, 3111 (2004) 260–272.

Finding Shortest Non-separating and Non-contractible Cycles for Topologically Embedded Graphs

Sergio Cabello^{1,*} and Bojan Mohar^{2,**}

¹ Department of Mathematics, Institute for Mathematics,
Physics and Mechanics, Slovenia
`sergio.cabello@imfm.uni-lj.si`

² Department of Mathematics, Faculty of Mathematics and Physics,
University of Ljubljana, Slovenia
`bojan.mohar@fmf.uni-lj.si`

Abstract. We present an algorithm for finding shortest surface non-separating cycles in graphs with given edge-lengths that are embedded on surfaces. The time complexity is $O(g^{3/2}V^{3/2} \log V + g^{5/2}V^{1/2})$, where V is the number of vertices in the graph and g is the genus of the surface. If $g = o(V^{1/3-\epsilon})$, this represents a considerable improvement over previous results by Thomassen, and Erickson and Har-Peled. We also give algorithms to find a shortest non-contractible cycle in $O(g^{O(g)}V^{3/2})$ time, improving previous results for fixed genus.

This result can be applied for computing the (non-separating) face-width of embedded graphs. Using similar ideas we provide the first near-linear running time algorithm for computing the face-width of a graph embedded on the projective plane, and an algorithm to find the face-width of embedded toroidal graphs in $O(V^{5/4} \log V)$ time.

1 Introduction

Cutting a surface for reducing its topological complexity is a common technique used in geometric computing and topological graph theory. Erickson and Har-Peled [9] discuss the relevance of cutting a surface to get a topological disk in computer graphics. Colin de Verdière [5] describes applications that algorithmical problems involving curves on topological surfaces have in other fields.

Many results in topological graph theory rely on the concept of face-width, sometimes called representativity, which is a parameter that quantifies local planarity and density of embeddings. The face-width is closely related to the edge-width, the minimum number of vertices of any shortest non-contractible cycle of an embedded graph [17]. Among some relevant applications, face-width

* Partially supported by the European Community Sixth Framework Programme under a Marie Curie Intra-European Fellowship.

** Supported in part by the Ministry of Higher Education, Science and Technology of Slovenia, Research Project L1-5014-0101-04 and Research Program P1-0297.

plays a fundamental role in the graph minors theory of Robertson and Seymour, and large face-width implies that there exists a collection of cycles that are far apart from each other, and after cutting along them, a planar graph is obtained. By doing so, many computational problems for locally planar graphs on general surfaces can be reduced to corresponding problems on planar graphs. See [17, Chapter 5] for further details. The efficiency of algorithmical counterparts of several of these results passes through the efficient computation of face-width.

The same can be said for the non-separating counterparts of the width parameters, where the surface non-separating (i.e., nonzero-homologous) cycles are considered instead of non-contractible ones. In this work, we focus on what may be considered the most natural problem for graphs embedded on surfaces: finding a shortest non-contractible and a shortest surface non-separating cycle. Our results give polynomial-time improvements over previous algorithms for low-genus embeddings of graphs (in the non-separating case) or for embeddings of graphs in a fixed surface (in the non-contractible case). In particular, we improve previous algorithms for computing the face-width and the edge-width of embedded graphs. In our approach, we reduce the problem to that of computing the distance between a few pairs of vertices, what some authors have called the *k-pairs shortest path problem*.

1.1 Overview of the Results

Let G be a graph with V vertices and E edges embedded on a (possibly non-orientable) surface Σ of genus g , and with positive weights on the edges, representing edge-lengths. Our main contributions are the following:

- We find a shortest surface non-separating cycle of G in $O(g^{3/2}V^{3/2} \log V + g^{5/2}V^{1/2})$ time, or $O(g^{3/2}V^{3/2})$ if $g = O(V^{1-\varepsilon})$ for some constant $\varepsilon > 0$. This result relies on a characterization of the surface non-separating cycles given in Section 3. The algorithmical implications of this characterization are described in Section 4.
- For any fixed surface, we find a shortest non-contractible cycle in $O(V^{3/2})$ time. This is achieved by considering a small portion of the universal cover. See Section 5.
- We compute the non-separating face-width and edge-width of G in $O(g^{3/2}V^{3/2} + g^{5/2}V^{1/2})$ time. For fixed surfaces, we can also compute the face-width and edge-width of G in $O(V^{3/2})$ time. For graphs embedded on the projective plane or the torus we can compute the face-width in near-linear or $O(V^{5/4} \log V)$ time, respectively. This is described in Section 6.

Although the general approach is common in all our results, the details are quite different for each case. The overview of the technique is as follows. We find a set of generators either for the first homology group (in the non-separating case) or the fundamental group (in the non-contractible case) that is made of a few geodesic paths. It is then possible to show that shortest cycles we are interested in (non-separating or non-contractible ones) intersect these generators according to

certain patterns, and this allows us to reduce the problem to computing distances between pairs of vertices in associated graphs.

We next describe the most relevant related work, and in Section 2 we introduce the basic background. The rest of the sections are as described above.

1.2 Related Previous Work

Thomassen [20] was the first to give a polynomial time algorithm for finding a shortest non-separating and a shortest non-contractible cycle in a graph on a surface; see also [17, Chapter 4]. Although Thomassen does not claim any specific running time, his algorithm tries a quadratic number of cycles, and for each one it has to decide if it is non-separating or non-contractible. This yields a rough estimate $O(V(V+g)^2)$ for its running time. More generally, his algorithm can be used for computing in polynomial time a shortest cycle in any class \mathcal{C} of cycles that satisfy the so-called *3-path-condition*: if u, v are vertices of G and P_1, P_2, P_3 are internally disjoint paths joining u and v , and if two of the three cycles $C_{i,j} = P_i \cup P_j$ ($i \neq j$) are not in \mathcal{C} , then also the third one is not in \mathcal{C} . The class of one-sided cycles for embedded graphs is another relevant family of cycles that satisfy the 3-path-condition.

Erickson and Har-Peled [9] considered the problem of computing a planarizing subgraph of minimum length, that is, a subgraph $C \subseteq G$ of minimum length such that $\Sigma \setminus C$ is a topological disk. They show that the problem is NP-hard when genus is not fixed, provide a polynomial time algorithm for fixed surfaces, and provide efficient approximation algorithms. More relevant for our work, they show that a shortest non-contractible (resp. non-separating) loop through a fixed vertex can be computed in $O(V \log V + g)$ (resp. $O((V+g) \log V)$) time, and therefore a shortest non-contractible (resp. non-separating) cycle can be computed in $O(V^2 \log V + Vg)$ (resp. $O(V(V+g) \log V)$) time. They also provide an algorithm that in $O(g(V+g) \log V)$ time finds a non-separating (or non-contractible) cycle whose length is at most twice the length of a shortest one.

Several other algorithmical problems for graphs embedded on surfaces have been considered. Colin de Verdière and Lazarus [6,7] considered the problem of finding a shortest cycle in a given homotopy class, a system of loops homotopic to a given one, and finding optimal pants decompositions. Eppstein [8] discusses how to use the tree-cotree partition for dynamically maintaining properties from an embedded graph under several operations. Very recently, Erickson and Whittlesey [10] present algorithms to determine a shortest set of loops generating the fundamental group. Other known results for curves embedded on topological surfaces include [2,3,16,21]; see also [18,19] and references therein.

2 Background

Topology. We consider *surfaces* Σ that are connected, compact, Hausdorff topological spaces in which each point has a neighborhood that is homeomorphic to \mathbb{R}^2 ; in particular, they do not have boundary. A *loop* is a continuous function of the circle S^1 in Σ . Two loops are *homotopic* if there is a continuous deformation

of one onto the other, that is, if there is a continuous function from the cylinder $S^1 \times [0, 1]$ to Σ such that each boundary of the cylinder is mapped to one of the loops. A loop is *contractible* if it is homotopic to a constant (a loop whose image is a single point); otherwise it is *non-contractible*. A loop is *surface separating* (or *zero-homologous*) if it can be expressed as the symmetric difference of boundaries of topological disks embedded in Σ ; otherwise it is *non-separating*. In particular, any non-separating loop is a non-contractible loop. We refer to [13] and to [17, Chapter 4] for additional details.

Representation of Embedded Graphs. We will assume the Heffter-Edmonds-Ringel representation of embedded graphs: it is enough to specify for each vertex v the circular ordering of the edges emanating from v and for each edge $e \in E(G)$ its *signature* $\lambda(e) \in \{+1, -1\}$. The negative signature of e tells that the selected circular ordering around vertices changes from clockwise to anti-clockwise when passing from one end of the edge to the other. For orientable surfaces, all the signatures can be made positive, and there is no need to specify it. This representation uniquely determines the embedding of G , up to homeomorphism, and one can compute the set of facial walks in linear time.

Let V denote the number of vertices in G and let g be the (Euler) genus of the surface Σ in which G is embedded. It follows from Euler's formula that G has $\Theta(V + g)$ edges. Asymptotically, we may consider $V + g$ as the measure of the size of the input.

We use the notation $G\mathcal{A}C$ for the surface obtained by cutting G along a cycle C . Each vertex $v \in C$ gives rise to two vertices v', v'' in $G\mathcal{A}C$. If C is a two-sided cycle, then it gives rise to two cycles C' and C'' in $G\mathcal{A}C$ whose vertices are $\{v' \mid v \in V(C)\}$ and $\{v'' \mid v \in V(C)\}$, respectively. If C is one-sided, then it gives rise to a cycle C' in $G\mathcal{A}C$ whose length is twice the length of C , in which each vertex v of C corresponds to two diagonally opposite vertices v', v'' on C' . The notation $G\mathcal{A}C$ naturally generalizes to $G\mathcal{A}\mathcal{C}$, where \mathcal{C} is a set of cycles.

Distances in Graphs. In general, we consider simple graphs with *positive* edge-weights, that is, we have a function $w : E \rightarrow \mathbb{R}^+$ describing the length of the edges. In a graph G , a *walk* is a sequence of vertices such that any two consecutive vertices are connected by an edge in G ; a *path* is a walk where all vertices are distinct; a *loop* is a walk where the first and last vertex are the same; a *cycle* is a loop without repeated vertices; a *segment* is a subwalk. The *length* of a walk is the sum of the weights of its edges, counted with multiplicity.

For two vertices $u, v \in V(G)$, the *distance* in G , denoted $d_G(u, v)$, is the minimum length of a path in G from u to v . A *shortest-path tree* from a vertex v is a tree T such that for any vertex u we have $d_G(v, u) = d_T(v, u)$; it can be computed in $O(V \log V + E) = O(V \log V + g)$ time [12], or in $O(V)$ time if $g = O(V^{1-\varepsilon})$ for any positive, fixed ε [14]. When all the edge-weights are equal to one, a breadth-first-search tree is a shortest-path tree.

We assume non-negative real edge-weights, and our algorithms run in the comparison based model of computation, that is, we only add and compare (sums of) edge weights. For integer weights and word-RAM model of computation, some logarithmic improvements may be possible.

Width of Embeddings. The *edge-width* $\text{ew}(G)$ (*non-separating edge-width* $\text{ew}_0(G)$) of a graph G embedded in a surface is defined as the minimum number of vertices in a non-contractible (resp. surface non-separating) cycle. The *face-width* $\text{fw}(G)$ (*non-separating face-width* $\text{fw}_0(G)$) is the smallest number k such that there exist facial walks W_1, \dots, W_k whose union contains a non-contractible (resp. surface non-separating) cycle.

2.1 k-Pairs Distance Problem

Consider the *k-pairs distance problem*: given a graph G with positive edge-weights and k pairs $(s_1, t_1), \dots, (s_k, t_k)$ of vertices of G , compute the distances $d_G(s_i, t_i)$ for $i = 1, \dots, k$. Djidjev [4] and Fakcharoenphol and Rao [11] (slightly improved by Klein [15] for non-negative edge-lengths) describe data structures for shortest path queries in planar graphs. We will need the following special case.

Lemma 1. *For a planar graph of order V , the k -pairs distance problem can be solved in $O(\min\{V^{3/2} + k\sqrt{V}, V \log^2 V + k\sqrt{V} \log^2 V\})$ time.*

For a graph G embedded on a surface of genus g , there exist a set $S \subset V(G)$ of size $O(\sqrt{gV})$ such that $G - S$ is planar. It can be computed in time linear in the size of the graph [8]. Since $G - S$ is planar, we can then use the previous lemma to get the following result.

Lemma 2. *The k -pairs distance problem can be solved in $O(\sqrt{gV}(V \log V + g + k))$ time, and in $O(\sqrt{gV}(V + k))$ time if $g = O(V^{1-\epsilon})$ for some $\epsilon > 0$.*

Proof. (Sketch) We compute in $O(V + g)$ time a vertex set $S \subset V(G)$ of size $O(\sqrt{gV})$ such that $G - S$ is a planar graph. Making a shortest path tree from each vertex $s \in S$, we compute all the values $d_G(s, v)$ for $s \in S, v \in V(G)$. We define the restricted distances $d_G^S(s_i, t_i) = \min_{s \in S} \{d_G(s, s_i) + d_G(s, t_i)\}$, and compute for each pair (s_i, t_i) the value $d_G^S(s_i, t_i)$

If s_i and t_i are in different connected components of $G - S$, it is clear that $d_G(s_i, t_i) = d_G^S(s_i, t_i)$. If s_i, t_i are in the same component G_j of $G - S$ we have $d_G(s_i, t_i) = \min\{d_{G_j}(s_i, t_i), d_G^S(s_i, t_i)\}$. We can compute $d_{G_j}(s_i, t_i)$ for all the pairs (s_i, t_i) in a component G_j using Lemma 1, and the lemma follows because each pair (s_i, t_i) is in one component. □

3 Separating vs. Non-separating Cycles

In this section we characterize the surface non-separating cycles using the concept of crossing. Let $Q = u_0u_1 \dots u_ku_0$ and $Q' = v_0v_1 \dots v_l v_0$ be cycles in the embedded graph G . If Q, Q' do not have any common edge, for each pair of common vertices $u_i = v_j$ we count a crossing if the edges $u_{i-1}u_i, u_iu_{i+1}$ of Q and the edges $v_{j-1}v_j, v_jv_{j+1}$ of Q' alternate in the local rotation around $u_i = v_j$; the resulting number is $cr(Q, Q')$. If Q, Q' are distinct and have a set of edges E' in common, then $cr(Q, Q')$ is the number of crossings after contracting G along

E' . If $Q = Q'$, then we define $cr(Q, Q') = 0$ if Q is two-sided, and $cr(Q, Q') = 1$ if Q is one-sided; we do this for consistency in later developments.

We introduce the concept of (\mathbb{Z}_2) -homology; see any textbook of algebraic topology for a comprehensive treatment. A set of edges E' is a *1-chain*; it is a *1-cycle* if each vertex has even degree in E' ; in particular, every cycle in the graph is a 1-cycle, and also the symmetric difference of 1-cycles is a 1-cycle. The set of 1-cycles with the symmetric difference operation $+$ is an Abelian group, denoted by $\mathcal{C}_1(G)$. This group can also be viewed as a vector space over \mathbb{Z}_2 and is henceforth called the *cycle space* of the graph G . If f is a closed walk in G , the edges that appear an odd number of times in f form a 1-cycle. For convenience, we will denote the 1-cycle corresponding to f by the same symbol f .

Two 1-chains E_1, E_2 are *homologically equivalent* if there is a family of facial walks f_1, \dots, f_t of the embedded graph G such that $E_1 + f_1 + \dots + f_t = E_2$. Being homologically equivalent is an equivalence relation compatible with the symmetric difference of sets. The 1-cycles that are homologically equivalent to the empty set, form a subgroup $\mathcal{B}_1(G)$ of $\mathcal{C}_1(G)$. The quotient group $H_1(G) = \mathcal{C}_1(G)/\mathcal{B}_1(G)$ is called the homology group of the embedded graph G .

A set L of 1-chains generates the homology group if for any loop l in G , there is a subset $L' \subset L$ such that l is homologically equivalent with $\sum_{l' \in L'} l'$. There are sets of generators consisting of g 1-chains. It is known that any generating set of the fundamental group is also a generating set of the homology group $H_1(G)$.

If $\mathcal{L} = \{L_1, \dots, L_g\}$ is a set of 1-cycles that generate $H_1(G)$, then every L_i ($1 \leq i \leq g$) contains a cycle Q_i such that the set $\mathcal{Q} = \{Q_1, \dots, Q_g\}$ generates $H_1(G)$. This follows from the exchange property of bases of a vector space since $H_1(G)$ can also be viewed as a vector space over \mathbb{Z}_2 .

A cycle in G is surface non-separating if and only if it is homologically equivalent to the empty set. We have the following characterization of non-separating cycles involving parity of crossing numbers.

Lemma 3. *Let $\mathcal{Q} = \{Q_1, \dots, Q_g\}$ be a set of cycles that generate the homology group $H_1(G)$. A cycle Q in G is non-separating if and only if there is some cycle $Q_i \in \mathcal{Q}$ such that Q and Q_i cross an odd number of times, that is, $cr(Q, Q_i) \equiv 1 \pmod{2}$.*

Proof. Let f_0, \dots, f_r be the 1-cycles that correspond to the facial walks. Then $f_0 = f_1 + \dots + f_r$ and $\mathcal{Q} \cup \{f_1, \dots, f_r\}$ is a generating set of $\mathcal{C}_1(G)$. If C is a 1-cycle, then $C = \sum_{j \in J} Q_j + \sum_{i \in I} f_i$. We define $cr_C(Q)$ as the modulo 2 value of

$$\sum_{j \in J} cr(Q, Q_j) + \sum_{i \in I} cr(Q, f_i) = \sum_{j \in J} cr(Q, Q_j) \pmod{2}.$$

It is easy to see that $cr_C : \mathcal{C}_1(G) \rightarrow \mathbb{Z}_2$ is a homomorphism. Since $cr(Q, f_i) = 0$ for every facial walk f_i , cr_C determines also a homomorphism $H_1(G) \rightarrow \mathbb{Z}_2$.

If Q is a surface separating cycle, then it corresponds to the trivial element of $H_1(G)$, so every homomorphism maps it to 0. In particular, for every j , $cr(Q, Q_j) = cr_{Q_j}(Q) = 0 \pmod{2}$.

Let Q be a non-separating cycle and consider $\tilde{G} = G \# Q$. Take a vertex $v \in Q$, which gives rise to two vertices $v', v'' \in \tilde{G}$. Since Q is non-separating, there is a simple path P in \tilde{G} connecting v', v'' . The path P is a loop in G (not necessarily a cycle) that crosses Q exactly once.

Since \mathcal{Q} generates the homology group, there is a subset $\mathcal{Q}' \subset \mathcal{Q}$ such that the loop P and $\sum_{Q_i \in \mathcal{Q}'} Q_i$ are homological. But then $1 = cr_P(Q) = \sum_{Q_i \in \mathcal{Q}'} cr(P, Q_i) \pmod 2$, which means that for some $Q_i \in \mathcal{Q}'$, it holds $cr(P, Q_i) \equiv 1 \pmod 2$. □

4 Shortest Non-separating Cycle

We use the tree-cotree decomposition for embedded graphs introduced by Epstein [8]. Let T be a spanning tree of G rooted at $x \in V(G)$. For any edge $e = uv \in E(G) \setminus T$, we denote by $loop(T, e)$ the closed walk in G obtained by following the path in T from x to u , the edge uv , and the path in T from v to x ; we use $cycle(T, e)$ for the cycle obtained by removing the repeated edges in $loop(T, e)$. A subset of edges $C \subseteq E(G)$ is a *cotree* of G if $C^* = \{e^* \in E(G^*) \mid e \in C\}$ is a spanning tree of the dual graph G^* . A *tree-cotree* partition of G is a triple (T, C, X) of disjoint subsets of $E(G)$ such that T forms a spanning tree of G , C is cotree of G , and $E(G) = T \cup C \cup X$. Euler's formula implies that if (T, C, X) is a tree-cotree partition, then $\{loop(T, e) \mid e \in X\}$ contains g loops and it generates the fundamental group of the surface; see, e.g., [8]. As a consequence, $\{cycle(T, e) \mid e \in X\}$ generates the homology group H_1 .

Let T_x be a shortest-path tree from vertex $x \in V(G)$. Let us fix any tree-cotree partition (T_x, C_x, X_x) , and let $\mathcal{Q}_x = \{cycle(T_x, e) \mid e \in X_x\}$. For a cycle $Q \in \mathcal{Q}_x$, let \mathcal{Q}_Q be the set of cycles that cross Q an odd number of times. Since \mathcal{Q}_x generates the homology group, Lemma 3 implies that $\bigcup_{Q \in \mathcal{Q}_x} \mathcal{Q}_Q$ is precisely the set of non-separating cycles. We will compute a shortest cycle in \mathcal{Q}_Q , for each $Q \in \mathcal{Q}_x$, and take the shortest cycle among all them; this will be a shortest non-separating cycle.

We next show how to compute a shortest cycle in \mathcal{Q}_Q for $Q \in \mathcal{Q}_x$. Firstly, we use that T_x is a shortest-path tree to argue that we only need to consider cycles that intersect Q exactly once; a similar idea is used by Erickson and Har-Peled [9] for their 2-approximation algorithm. Secondly, we reduce the problem of finding a shortest cycle in \mathcal{Q}_Q to an $O(V)$ -pairs distance problem.

Lemma 4. *Among the shortest cycles in \mathcal{Q}_Q , where $Q \in \mathcal{Q}_x$, there is one that crosses Q exactly once.*

Proof. (Sketch) Let Q_0 be a *shortest* cycle in \mathcal{Q}_Q for which the number $Int(Q, Q_0)$ of connected components of $Q \cap Q_0$ is minimum. We claim that $Int(Q, Q_0) \leq 2$, and therefore $cr(Q, Q_0) = 1$ because \mathcal{Q}_Q is the set of cycles crossing Q an odd number of times, and each crossing is an intersection. Using the 3-path-condition and that the cycle Q is made of two shortest paths, it is not difficult to show that $Int(Q, Q_0) \geq 3$ cannot happen. □

Lemma 5. *For any $Q \in \mathcal{Q}_x$, we can compute a shortest cycle in \mathcal{Q}_Q in $O((V \log V + g)\sqrt{gV})$ time, or $O(V\sqrt{gV})$ time if $g = O(V^{1-\varepsilon})$.*

Proof. Consider the graph $\tilde{G} = G \# Q$, which is embedded in a surface of Euler genus $g - 1$ (if Q is a 1-sided curve in Σ) or $g - 2$ (if Q is 2-sided). Each vertex v on Q gives rise to two copies v', v'' of v in \tilde{G} .

In G , a cycle that crosses Q exactly once (at vertex v , say) gives rise to a path in \tilde{G} from v' to v'' (and vice versa). Therefore, finding a shortest cycle in \mathcal{Q}_Q is equivalent to finding a shortest path in \tilde{G} between pairs of the form (v', v'') with v on Q . In \tilde{G} , we have $O(V)$ pairs (v', v'') with v on Q , and using Lemma 2 we can find a closest pair (v'_0, v''_0) in $O((V \log V + g)\sqrt{gV})$ time, or $O(V\sqrt{gV})$ if $g = O(V^{1-\varepsilon})$. We use a single source shortest path algorithm to find in \tilde{G} a shortest path from v'_0 to v''_0 , and hence a shortest cycle in \mathcal{Q}_Q . \square

Theorem 1. *Let G be a graph with V vertices embedded on a surface of genus g . We can find a shortest surface non-separating cycle in $O((gV \log V + g^2)\sqrt{gV})$ time, or $O((gV)^{3/2})$ time if $g = O(V^{1-\varepsilon})$.*

Proof. Since $\bigcup_{Q \in \mathcal{Q}_x} \mathcal{Q}_Q$ is precisely the set of non-separating cycles, we find a shortest non-separating cycle by using the previous lemma for each $Q \in \mathcal{Q}_x$, and taking the shortest among them. The running time follows because \mathcal{Q}_x contains $O(g)$ loops. \square

Observe that the algorithm by Erickson and Har-Peled [9] outperforms our result for $g = \Omega(V^{1/3} \log^{2/3} V)$. Therefore, we can recap concluding that a shortest non-separating cycle can be computed in $O(\min\{(gV)^{3/2}, V(V + g) \log V\})$ time.

5 Shortest Non-contractible Cycle

Like in the previous section, we consider a shortest-path tree T_x from vertex $x \in V(G)$, and we fix a tree-cotree partition (T_x, C_x, X_x) . Consider the set of loops $L_x = \{\text{loop}(T_x, e) \mid e \in X_x\}$, which generates the fundamental group with base point x . By increasing the number of vertices to $O(gV)$, we can assume that L_x consists of cycles (instead of loops) whose pairwise intersection is x . This can be shown by slightly modifying G in such a way that L_x can be transformed without harm.

Lemma 6. *The problem is reduced to finding a shortest non-contractible cycle in an embedded graph \tilde{G} of $O(gV)$ vertices with a given set of cycles \mathcal{Q}_x such that: \mathcal{Q}_x generates the fundamental group with basepoint x , the pairwise intersection of cycles from \mathcal{Q}_x is only x , and each cycle from \mathcal{Q}_x consists of two shortest paths from x plus an edge. This reduction can be done in $O(gV)$ time.*

Proof. (Sketch) The first goal is to change the graph G in such a way that the loops in L_x will all become cycles. Then we handle the pairwise intersections between them. The procedure is as follows. Consider a non-simple loop l_0 in

L_x whose repeated segment P_0 is shortest, and replace the vertices in P_0 in the graph as shown in Figure 1. We skip a detailed description since it involves much notation, but the idea should be clear from the figure. Under this transformation, the rest of loops (or cycles) in L_x remain the same except that their segment common with P_0 is replaced with the corresponding new segments. We repeat this procedure until L_x consists of only cycles; we need $O(g)$ repetitions. This achieves the first goal, and the second one can be achieved doing a similar transformation if we consider at each step the pair of cycles that have a longest segment in common. \square

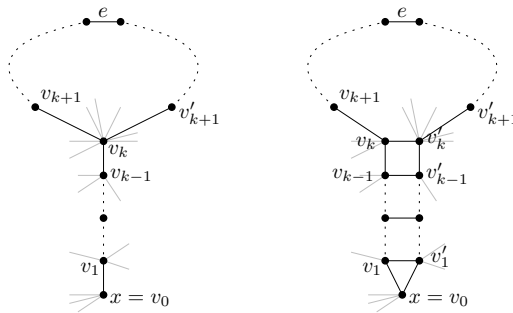


Fig. 1. Changing G such that a loop $l_0 \in L_x$ becomes a cycle. The edges $v_i v'_i$ have length 0.

Therefore, from now on, we only consider scenarios as stated in Lemma 6. Let \mathcal{Q}^* be the set of shortest non-contractible cycles in \tilde{G} . Using arguments similar to Lemma 4, we can show the following.

Lemma 7. *There is a cycle $Q \in \mathcal{Q}^*$ that crosses each cycle in \mathcal{Q}_x at most twice.*

Consider the set $D = \Sigma \mathcal{Q}_x$ and the corresponding graph $G_P = \tilde{G} \mathcal{Q}_x$. Since \mathcal{Q}_x is a set of cycles that generate the fundamental group and they only intersect at x , it follows that D is a topological disk, and G_P is a planar graph. We can then use D and G_P as building blocks to construct a portion of the universal cover where a shortest non-contractible cycle has to lift.

Theorem 2. *Let G be a graph with V vertices embedded on a surface of genus g . We can find a shortest non-contractible cycle in $O(g^{O(g)} V^{3/2})$ time.*

Proof. According to Lemma 6, we assume that \tilde{G} has $O(gV)$ vertices and we are given a set of cycles \mathcal{Q}_x that generate the fundamental group with base point x , whose pairwise intersection is x , and such that each cycle of \mathcal{Q}_x consists of two shortest paths plus an edge. Moreover, because of Lemma 7, there is a shortest non-contractible cycle crossing each cycle of \mathcal{Q}_x at most twice.

Consider the topological disk $D = \Sigma \mathcal{Q}_x$ and let U be the universal cover that is obtained by gluing copies of D along the cycles in \mathcal{Q}_x . Let G_U be the

universal cover of the graph \tilde{G} that is naturally embedded in U . The graph G_U is an infinite planar graph, unless Σ is the projective plane \mathbb{P}^2 , in which case G_U is finite.

Let us fix a copy D_0 of D , and let U_0 be the portion of the universal cover U which is reachable from D_0 by visiting at most $2g$ different copies of D . Since each copy of D is adjacent to $2|\mathcal{Q}_x| \leq 2g$ copies of D , U_0 consists of $(2g)^{2g} = g^{O(g)}$ copies of D . The portion G_{U_0} of the graph G_U that is contained in U_0 can be constructed in $O(g^{O(g)}gV) = O(g^{O(g)}V)$ time. We assign to the edges in G_{U_0} the same weights they have in G .

A cycle is non-contractible if and only if its lift in U finishes in different copies of the same vertex. Each time that we pass from a copy of D to another copy we must intersect a cycle in \mathcal{Q}_x . Using the previous lemma, we conclude that there is a shortest non-contractible cycle whose lift intersects at most $2|\mathcal{Q}_x| = O(g)$ copies of D . That is, there exists a shortest non-contractible cycle in G whose lifting to U starts in D_0 and is contained G_{U_0} .

We can then find a shortest non-contractible cycle by computing, for each vertex $v \in D_0$, the distance in G_{U_0} from the vertex v to all the other copies of v that are in G_{U_0} . Each vertex $v \in D_0$ has $O(g^{O(g)})$ copies in G_{U_0} . Therefore, the problem reduces to computing the shortest distance in G_{U_0} between $O(g^{O(g)}V)$ pairs of vertices. Since G_{U_0} is a planar graph with $O(g^{O(g)}V)$ vertices, we can compute these distances using Lemma 1 in $O(g^{O(g)}V\sqrt{g^{O(g)}V}) = O(g^{O(g)}V^{3/2})$ time. \square

Observe that, for a fixed surface, the running time of the algorithm is $O(V^{3/2})$. However, for most values of g as a function of V (when $g \geq c\frac{\log V}{\log \log V}$ for a certain constant c), the near-quadratic time algorithm by Erickson and Har-Peled [9] is better.

6 Edge-Width and Face-Width

When edge-lengths are all equal to 1, shortest non-contractible and surface non-separating cycles determine combinatorial width parameters (cf. [17, Chapter 5]). Since their computation is of considerable interest in topological graph theory, it makes sense to consider this special case in more details.

6.1 Arbitrary Embedded Graphs

The (*non-separating*) *edge-width* $\text{ew}(G)$ (and $\text{ew}_0(G)$, respectively) of an embedded graph G is the minimum number of vertices in a non-contractible (surface non-separating) cycle, which can be computed by setting $w(e) = 1$ for all edges e in G and running the algorithms from previous sections. For computing the (*non-separating*) *face-width* $\text{fw}(G)$ (and $\text{fw}_0(G)$, respectively) of an embedded graph G , it is convenient to consider its vertex-face incidence graph Γ : a bipartite graph whose vertices are faces and vertices of G , and there is an edge between face f and vertex v if and only if v is on the face f . The construction of Γ takes linear time from an embedding of G , and it holds that $\text{fw}(G) = \frac{1}{2}\text{ew}(\Gamma)$

and $\mathbf{fw}_0(G) = \frac{1}{2}\mathbf{ew}_0(\Gamma)$ [17]. In this setting, since a breadth-first-search tree is a shortest-path tree, a log factor can be shaved off.

Theorem 3. *For a graph G embedded in a surface of genus g , we can compute its non-separating edge-width and face-width in $O(g^{3/2}V^{3/2} + g^{5/2}V^{1/2})$ time and its edge-width and face-width in $O(g^{O(g)}V^{3/2})$ time.*

Although in general it can happen that $\mathbf{ew}(G) = \Omega(V)$, there are non-trivial bounds on the face-width $\mathbf{fw}(G)$. Albertson and Hutchinson [1] showed that the edge-width of a *triangulation* is at most $\sqrt{2V}$. Since the vertex-face graph Γ has a natural embedding in the same surface as G as a quadrangulation, we can add edges to it to obtain a triangulation T , and conclude that $\mathbf{fw}(G) = \frac{1}{2}\mathbf{ew}(\Gamma) \leq \mathbf{ew}(T) \leq \sqrt{2V}$.

6.2 Face-Width in the Projective Plane and the Torus

For the special cases when G is embedded in the projective plane \mathbb{P}^2 or the torus \mathbb{T} , we can improve the running time for computing the face-width. The idea is to use an algorithm for computing the edge-width whose running time depends on the value $\mathbf{ew}(G)$. We only describe the technique for the projective plane.

Lemma 8. *Let G be a graph embedded in \mathbb{P}^2 . If $\mathbf{ew}(G) \leq t$, then we can compute $\mathbf{ew}(G)$ and find a shortest non-contractible cycle in $O(V \log^2 V + t\sqrt{V} \log^2 V)$ time.*

Proof. (Sketch) Since the sphere is the universal cover of the projective plane \mathbb{P}^2 , we can consider the cover of G on the sphere, the so-called double cover D_G of the embedding of G , which is a planar graph. Each vertex v of G gives rise to two copies v, v' in D_G , and a shortest non-contractible loop passing through a vertex $v \in V(G)$ is equivalent to a shortest path in D_G between the vertices v and v' .

We compute in $O(V \log V)$ time a non-contractible cycle Q of G of length at most $2\mathbf{ew}(G) \leq 2t$ [9]. Any non-contractible cycle in G has to intersect Q at some vertex, and therefore the problem reduces to find two copies $v, v' \in D_G$ of the same vertex $v \in Q$ that minimize their distance in d_G . This requires $|Q| \leq 2t$ pairs of distances in D_G , which can be solved using Lemma 1. \square

Like before, consider the vertex-face incidence graph Γ which can be constructed in linear time. From the bounds in Section 6.1, we know that the edge-width of Γ is $O(\sqrt{V})$, and computing the face-width reduces to computing the edge-width of a graph knowing a priori that $\mathbf{ew}(\Gamma) = 2\mathbf{fw}(G) = O(\sqrt{V})$. Using the previous lemma we conclude the following.

Theorem 4. *If G is embedded in \mathbb{P}^2 we can find $\mathbf{fw}(G)$ in $O(V \log^2 V)$ time.*

For the torus, we have the following result, whose proof we omit.

Theorem 5. *If G is embedded in \mathbb{T} we can find $\mathbf{fw}(G)$ in $O(V^{5/4} \log V)$ time.*

References

1. M. O. Albertson and J. P. Hutchinson. On the independence ratio of a graph. *J. Graph Theory*, 2:1–8, 1978.
2. T. K. Dey and S. Guha. Transforming curves on surfaces. *J. Comput. Syst. Sci.*, 58:297–325, 1999. Preliminary version in FOCS'95.
3. T. K. Dey and H. Schipper. A new technique to compute polygonal schema for 2-manifolds with application to null-homotopy detection. *Discrete Comput. Geom.*, 14:93–110, 1995.
4. H. Djidjev. On-line algorithms for shortest path problems on planar digraphs. In *WG'96*, volume 1197 of *LNCS*, pages 151–165, 1997.
5. É. Colin de Verdière. *Shortening of curves and decomposition of surfaces*. PhD thesis, University Paris 7, December 2003.
6. É. Colin de Verdière and F. Lazarus. Optimal system of loops on an orientable surface. In *FOCS 2002*, pages 627–636, 2002. To appear in *Discrete Comput. Geom.*
7. É. Colin de Verdière and F. Lazarus. Optimal pants decompositions and shortest homotopic cycles on an orientable surface. In *GD 2003*, volume 2912 of *LNCS*, 2004.
8. D. Eppstein. Dynamic generators of topologically embedded graphs. In *SODA 2003*, pages 599–608, 2003.
9. J. Erickson and S. Har-Peled. Optimally cutting a surface into a disk. *Discrete Comput. Geom.*, 31:37–59, 2004. Preliminary version in SoCG'02.
10. J. Erickson and K. Whittlesey. Greedy optimal homotopy and homology generators. In *SODA 2005*, 2005.
11. J. Fakcharoenphol and S. Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. In *FOCS 2001*, pages 232–242, 2001.
12. M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34:596–615, 1987. Preliminary version in FOCS'84.
13. A. Hatcher. *Algebraic Topology*. Cambridge University Press, 2001.
14. M. Henzinger, P. Klein, S. Rao, and S. Subramanian. Faster shortest-path algorithms for planar graphs. *J. Comput. Syst. Sci.*, 55(1):3–23, 1997.
15. P. N. Klein. Multiple-source shortest paths in planar graphs. In *SODA 2005*, 2005.
16. F. Lazarus, M. Pocchiola, G. Vegter, and A. Verroust. Computing a canonical polygonal schema of an orientable triangulated surface. In *SOCG 2001*, pages 80–89, 2001.
17. B. Mohar and C. Thomassen. *Graphs on Surfaces*. Johns Hopkins University Press, Baltimore, 2001.
18. A. Schrijver. Disjoint circuits of prescribed homotopies in a graph on a compact surface. *J. Combin. Theory Ser. B*, 51:127–159, 1991.
19. A. Schrijver. Paths in graphs and curves on surfaces. In *First European Congress of Mathematics, Vol. II*, volume 120 of *Progr. Math.*, pages 381–406. Birkhäuser, 1994.
20. C. Thomassen. Embeddings of graphs with no short noncontractible cycles. *J. Combin. Theory, Ser. B*, 48:155–177, 1990.
21. G. Vegter and C. K. Yap. Computational complexity of combinatorial surfaces. In *SOCG 1990*, pages 102–111, 1990.

Delineating Boundaries for Imprecise Regions*

Iris Reinbacher¹, Marc Benkert², Marc van Kreveld¹, Joseph S. B. Mitchell³,
and Alexander Wolff²

¹ Institute of Information and Computing Sciences, Utrecht University
`{iris, marc}@cs.uu.nl`

² Dept. of Comp. Science, Karlsruhe University
`i11www.ira.uka.de/algo/group`

³ Department of Applied Mathematics and Statistics,
State University of New York at Stony Brook
`jsbm@ams.sunysb.edu`

Abstract. In geographic information retrieval, queries often use names of geographic regions that do not have a well-defined boundary, such as “Southern France.” We provide two classes of algorithms for the problem of computing reasonable boundaries of such regions, based on evidence of given data points that are deemed likely to lie either inside or outside the region. Our problem formulation leads to a number of problems related to red-blue point separation and minimum-perimeter polygons, many of which we solve algorithmically. We give experimental results from our implementation and a comparison of the two approaches.

1 Introduction

Geographic information retrieval is concerned with information retrieval for spatially related data, including Web searching. Certain specialized search engines allow queries that ask for things (hotels, museums) in the (geographic) neighborhood of some named location. These search engines cannot use the standard term matching on a large term index, because the user is not interested in the term “neighborhood” or “near”. When a user asks for Web pages on museums near Utrecht, not only Web pages that contain the terms “museum” and “Utrecht” should be found, but also Web pages of museums in Amersfoort, a city about 20 kilometers from Utrecht. Geographic search engines require ontologies—geographic databases—to be able to answer such queries. The ontologies store all geographic information, including coordinates and geographic concepts. Geographic search engines also require a combined spatial and term index to retrieve the relevant Web pages efficiently.

Besides specifying neighborhoods, users of geographic search engines may also use containment and directional concepts in the query. Furthermore, the named location need not be a city name or region with well-specified, administrative boundaries. A typical query could ask for campgrounds in Northern Portugal, or

* This research is supported by the EU-IST Project No. IST-2001-35047 (SPIRIT) and by grant WO 758/4-2 of the German Science Foundation (DFG).

specify locations such as Central Mexico, the Bible Belt, or the British Midlands. The latter two are examples of named regions for which no exact boundaries exist. The extent of such a region is in a sense in the minds of the people. Every country has several such imprecise regions.

Since geographic queries may ask for Web pages about castles in the British Midlands, it is useful to have a reasonable boundary for this imprecise region. This enables us to find Web pages for locations in the British Midlands that mention castles, even if they do not contain the words British Midlands. We need to store a reasonable boundary for the British Midlands in the geographic ontology during preprocessing, and use query time for searching in the spatial part of the combined spatial and term index.

To determine a reasonable boundary for an imprecise region we can use the Web once again. The enormous amount of text on all Web pages can be used as a source of data; the idea of using the Web as a geo-spatial database has appeared before [11]. A possible approach is using so-called *trigger phrases*. For any reasonable-size city in the British Midlands, like Nottingham, it is quite likely that some Web page contains a sentence fragment like “. . . Nottingham, a city in the British Midlands, . . .”, or “Nottingham is located in the British Midlands. . .”. Such sentence fragments give a location that is most likely in the British Midlands, while other cities like London or Cardiff, which do not appear in similar sentence fragments, give locations that are not in the British Midlands. Details of using trigger phrases to determine locations inside or outside a region to be delineated can be found in [1]. Obviously the process is not very reliable, and false positives and false negatives are likely to occur.

We have arrived at the following computational problem: given a set of “inside” points (red) and a set of “outside” points (blue), determine a reasonable polygon that separates the two sets. Imprecise regions generally are not thought of as having holes or a tentacle-shaped boundary, but rather a compact shape. Therefore, possible criteria for such a polygon are: The red points are inside and the blue points are outside the polygon, which is simply connected and has small perimeter. Other shape measures for polygons that are used in geography [12], e.g. the compactness ratio, can also be applied.

In computational geometry, various red-blue separation algorithms exist; see [14] for a survey. Red-blue separation by a line can be solved by two-dimensional linear programming in $O(n)$ time for n points. Red-blue separation by a line with k misclassified points takes $O((n + k^2) \log k)$ expected time [5]. Other fixed separation shapes, e.g. strips, wedges, and sectors, have also been considered [14]. For polygonal separators, a natural choice is the minimum-perimeter polygon that separates the bichromatic point set. This problem is NP-hard (by reduction from Euclidean traveling salesperson [6]); polynomial-time approximation schemes follow from the m -guillotine method of Mitchell [10] and from Arora’s method [3]. Minimum-link separation has also received attention [2].

In this paper we present two approaches to determine a reasonable polygon for a set of red and blue points. Based on these approaches we define and solve various algorithmic problems. The first approach takes a red polygon with blue

and red points inside, and tries to adapt the polygon to get the blue points outside while keeping the red points inside. We show that for a red polygon with n vertices and only one blue point inside, the minimum perimeter adaptation can be computed in $O(n)$ time. For the case of one blue and $O(n)$ red points inside, an $O(n \log n)$ -time algorithm is presented. If there are m blue points but no red points inside, an $O(m^3 n^3)$ -time algorithm is given. If there are m red and blue points inside, we give an $O(C^{m \log m} \cdot n)$ -time algorithm, for some constant C . These results are given in Section 2. The second approach changes the color of points to obtain a better shape of the polygon. Different recoloring strategies and algorithms are presented in Section 3. The implementation and test results on several data sets for both approaches are given in Section 4.

2 Adaptation Method

In the adaptation method, we start with a polygon P and adapt it until all blue points inside P are no longer inside, or the shape has to be changed too dramatically. By choosing P initially as an α -shape (see [7] for a definition), with α chosen such that e.g. 90% of the red points lie inside P , we can determine an appropriate initial shape and remove red outliers (red points outside P) in the same step. The parameter α can also be chosen based on “jumps” in the function that maps α to the perimeter of the initial polygon that we would choose. Once P is computed, the remaining problem is to change P so that the blue points are no longer inside. The resulting polygon P^* should be contained in P and its perimeter should be minimum. In this section we discuss the algorithmic side of this problem. In practice it may be better for the final shape to allow some blue points inside, which then would be considered misclassified. Some of our algorithms can handle this extension.

2.1 One Blue Point Inside P

First, we assume that there is only one blue point b inside P . The optimal, minimum-perimeter polygon P^* inside P has the following structure:

Lemma 1. *An optimal polygon P^* is a—possibly degenerate—simple polygon that (i) has b on its boundary, and (ii) contains all edges of P , except one.*

We call a simple polygon *degenerate*, if there are vertices that appear more than once on its boundary. We consider two versions of the problem: the special case where P contains only one point (namely b), and the general case where P contains b and a number of red points.

One Blue and No Red Points Inside P . Let b be the only point in the interior of P and let $e = \overline{v_1 v_2}$ be the edge of P that is not an edge of P^* . The boundary of the polygon P^* contains a path F^* that connects v_1 and v_2 via b . The path F^* consists of a shortest geodesic path between b and v_1 , and between b and v_2 . Note that these paths can contain red points other than v_1 and v_2 which are concave vertices of P , see Figure 1 (left). In the optimal solution P^* , the path F^* and the edge e have the following additional properties.

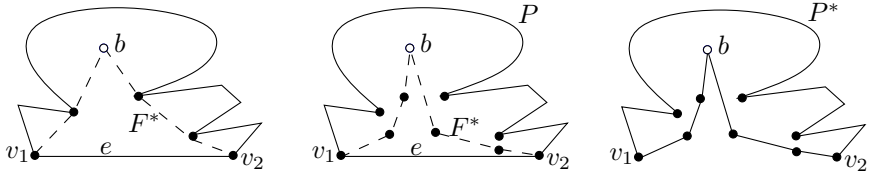


Fig. 1. Left: the path F^* if $R = \emptyset$. Middle: the case $R \neq \emptyset$. Right: P^* for $R \neq \emptyset$.

Lemma 2. (i) The path F^* is a simple funnel. (ii) The base e of the funnel F^* is partially visible from b .

We use the algorithm of Guibas et al. [8] to find the shortest path from the point b to every vertex v of the polygon. For every two adjacent vertices v_i and v_{i+1} of the polygon, we compute the shortest paths connecting them to b . The algorithm of Guibas et al. [8] can find all these paths in $O(n)$ time. For each possible base edge and corresponding funnel, we add the length of the two paths and subtract the length of the edge between v_i and v_{i+1} to get the value of the perimeter change for this choice. We obtain the following result.

Theorem 1. For a simple polygon P with n vertices and with a single point b inside, we can compute in $O(n)$ time the minimum-perimeter polygon $P^* \subseteq P$, that contains all vertices of P , and that has b on its boundary.

One Blue and Several Red Points Inside P . Let R be the set of the red points in the interior of P . Assume that its size is $O(n)$. We need to adapt the algorithm given before to take the red points into account. We first triangulate the polygon P . Ignoring the red points R , we compute all funnels F from b to every edge e of P . We get a partitioning of P into $O(n)$ funnels with disjoint interiors. In every funnel we do the following: If there are no red points inside F , we just store the length of the funnel without its base edge. Otherwise, we need to find a shortest path π_{\min} from one endpoint of the base edge to b and back to the other endpoint of the base edge, such that all red points in R still lie inside the resulting polygon P^* .

The shortest path π_{\min} inside some funnel F with respect to a set $R \cap F$ of red points consists of two chains which, together with the base edge e , again forms a funnel F^* , see Figure 1 (middle). This funnel is not allowed to contain points of $R \cap F$. We need to consider all possible ways of making such funnels, which involves partitioning the points of $R \cap F$ into two subsets. Fortunately, the red points of $R \cap F$ can only appear as reflex points on the funnel F^* , and therefore we can use an order of these points. For ease of description we let e be horizontal. Then F has a left chain and a right chain. The optimal funnel F^* also has a left chain and a right chain, such that all points of $R \cap F$ lie between the left chains of F and F^* and between the right chains of F^* and F . We extend the two edges of F incident to b , so that they end on the base edge e . This partitions F into three parts: a left part, a middle triangle, and a right part. In the same way as the first claim of Lemma 2, we can show that all points

of $R \cap F$ in the left part must be between the left chains of F and F^* , and all points of $R \cap F$ in the right part must be between the right chains of F^* and F . The points of $R \cap F$ in the middle triangle are sorted by angle around b . Let the order be r_1, \dots, r_h , counterclockwise.

Lemma 3. *There is an i such that the points r_1, \dots, r_i lie between the left chains of F and F^* , and r_{i+1}, \dots, r_h lie between the right chains of F^* and F .*

We iterate through the $h + 1$ possible partitions that can lead to an optimal funnel F^* , and maintain the two chains using a dynamic convex-hull algorithm [4]. Every next pair of chains requires a deletion of a point on one chain and an insertion of the same point on the other chain. We maintain the length of the path during these updates to find the optimal one.

As to the efficiency, finding all shortest paths from b to all vertices of P takes linear time for a polygon. Assigning the red points of R to the funnels takes $O(n \log n)$ time using either plane sweep or planar point location. Sorting the h red points inside F takes $O(h \log h)$ time, and the same amount of time is taken for the dynamic convex hull part. Since each red point of R appears in only one funnel, the overall running time is $O(n \log n)$.

Theorem 2. *For a simple polygon P with n vertices, a point b in P , and a set R of $O(n)$ red points in P , we can compute the minimum-perimeter polygon $P^* \subseteq P$ that contains all vertices of P and all red points of R , and that has b on its boundary, in $O(n \log n)$ time.*

2.2 Several Blue Points Inside P

When there are more blue points inside P , we use other algorithmic techniques. We first deal with the case of only blue points inside P and give a dynamic-programming algorithm. Then we assume that there is a constant number of blue and red points inside P , and we give a fixed-parameter tractable algorithm.

Several Blue and No Red Points Inside P . We sketch the dynamic-programming solution for the case that there are only blue points inside P . Details are given in the full version [13]. Let B be the set of blue points and let P^* be the optimal solution with no points of B inside, see Figure 2, left. The difference $P \setminus P^*$ defines a set of simple polygons called *pockets*. Each pocket contains one edge of P , which is called the *lid* of the pocket.

The structure of the solution is determined by a partitioning of the blue points into groups. Blue points are in the same group if they are on the boundary of the same pocket, or inside it. All blue points on the boundary of a pocket are convex for the pocket and concave for P^* .

The dynamic-programming solution is based on the idea that if \overline{uv} is a diagonal of P^* between two red vertices of P , then the solutions in the two subpolygons to either side of the diagonal are independent. If \overline{uv} is a diagonal of P^* between a red point u of P and a blue point v of B , and we know the lid of the pocket that v is part of, then the solutions in the two subpolygons are also independent.

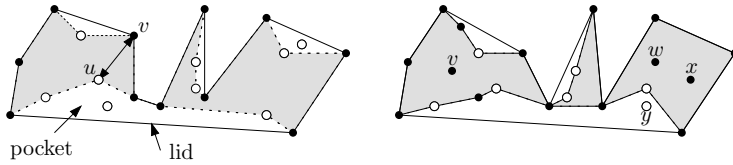


Fig. 2. Structure of an optimal subpolygon if only blue points are inside P (left), and if blue and red points are inside (right). Blue points are shown as white disks, red points as black disks.

Similarly, if u and v are both blue, we need to know both lids of both pockets. Therefore, optimal solutions to subproblems are characterized by a function with four parameters. This function gives the length of the optimal subpolygon up to the diagonal \overline{uv} (if applicable, with the specified pockets).

The recursion needed to set up dynamic programming takes the diagonal \overline{uv} and possibly, the lid specifications, and tries all possibilities for a point w such that triangle $\triangle uvw$ is part of a triangulation that gives an optimal subpolygon up to \overline{uv} for this set of parameters. If \overline{uv} and \overline{vw} are also diagonals, we need the smaller optimal solutions up to these two diagonals and add the lengths. To compute a function value, we must try $O(n)$ choices for w , and if w is a blue point, we also need to choose the lid of the pocket, giving another $O(n)$ choices. Hence, it takes $O(n^2)$ time to determine each of the $O(n^4)$ function values needed to compute the optimal polygon.

Theorem 3. *For a simple polygon P with n vertices and $O(n)$ blue points inside it, we can compute a minimum-perimeter polygon $P^* \subseteq P$ in $O(n^6)$ time. If there are $m = \Omega(n)$ blue points, the running time is $O(m^3 n^3)$.*

Several Blue and Red Points Inside P . We next give an algorithm that can handle k red and blue points inside a red polygon P with n vertices. The algorithm is fixed-parameter tractable: it takes $O(C^{k \log k} \cdot n)$ time, where C is some constant.

Let R and B be the sets of red and blue points inside P , respectively, and let $k = |R| + |B|$. The structure of the solution is determined by a partitioning of $R \cup B$ into groups, see Figure 2, right. One group contains points of $R \cup B$ that do not appear on the boundary of P^* . In Figure 2, right, this group contains v, w, x , and y . For the other groups, the points are in the same group if they lie on the same chain that forms a pocket, together with some lid. On this chain, points from B must be convex and points from R must be concave for the pocket. Besides points from $R \cup B$, the chain consists of geodesics between consecutive points from $R \cup B$. For all points in the group not used on chains, all points that come from B must be in pockets, and all points that come from R may not be in pockets (they must lie in P^*).

For a fixed-parameter tractable algorithm, we try all options for $R \cup B$. To this end, we generate all permutations of $R \cup B$, and all splits of each permutation into groups. The first group can be seen as the points that do not contribute to any chain. For any other group, we get a sequence of points (ordered) that lie

in this order on a chain. We compute the full chain by determining geodesics between consecutive points, and determine the best edge of P to replace by this chain (we need one more geodesic to connect to the first point and one to connect to the last point of the chain). We repeat this for every (ordered) group in the split permutation.

Theorem 4. *For a simple polygon P with n vertices, and k red and blue points inside it, we can compute a minimum-perimeter polygon $P^* \subseteq P$ in $O(C^k \log^k \cdot n)$ time, for some constant C .*

3 Recoloring Methods

In the adaptation method, we changed the boundary of the red region to bring blue points to the outside. However, if a blue point p is surrounded by red points, it may have been classified wrongly and recoloring it to red may lead to a more natural boundary of the red region. Similarly, red points surrounded by blue points may have been classified wrongly and we can recolor them to blue.

In this section we present methods for recoloring the given points, i.e. assigning a new inside-outside classification. The starting point for our methods is as follows: We are given a set P of n points, each of which is either red or blue. We first compute the Delaunay Triangulation $DT(P)$ of P . In $DT(P)$, we color edges red if they connect two red points, blue if they connect two blue points, and green otherwise. A red point is incident only to red and green edges, and a blue point is incident only to blue and green edges. To formalize that a point is surrounded by points of the other color, we define:

Definition 1. *Let the edges of $DT(P)$ be colored as above. Then the green angle ϕ of $p \in P$ is*

- 360° , if p is only incident to green edges,
- 0° , if p has at most one radially consecutive incident green edge,
- the maximum turning angle between two or more radially consecutive incident green edges otherwise.

We recolor points only if their green angle ϕ is at least some threshold value Φ . Note that if Φ has any value below 180° , there is a simple example where there is no termination. So we assume in any case that $\Phi \geq 180^\circ$; a suitable value for the application can be found empirically. After the algorithm has terminated, we define the regions as follows. Let M be the set of midpoints of the green edges. Then, each Delaunay triangle contains either no point or two points of M . In each triangle that contains two points of M , we connect the points by a straight line segment. These segments define the boundary between the red and the blue region. Note that each point of M on the convex hull of the point set is incident to one boundary segment while the other points of M are incident to exactly two boundary segments. Thus, the set of boundary segments consists of connected components that are either cycles, or chains that connect two points on the convex hull. We define the perimeter of the separation to be the total length of the boundary cycles and chains.

Observation 1. *If we can recolor a blue point to be red, then we do not destroy this option if we first recolor other blue points to be red. We also cannot create possibilities for coloring points red if we have only colored other points blue.*

We can now describe our first recoloring method, the *preferential scheme*. We first recolor all blue points with green angle $\phi \geq \Phi$ red, and afterwards, all red points with green angle $\phi \geq \Phi$ blue. It can occur that points that are initially blue become red, and later blue again. However, with our observation we can see that no more points can be recolored. Hence, this scheme leads to at most a linear number of recolorings. As this scheme gives preference of one color over the other, it is not fair and therefore not satisfactory. It would for example be better to recolor by decreasing green angle, since we then recolor points first that are most likely to be misclassified.

Another scheme is the true *adversary scheme*, where an adversary may recolor any point with green angle $\phi \geq 180^\circ$. For this scheme we do not have a termination proof, nor do we have an example where termination does not occur.

3.1 The Angle-and-Perimeter Scheme

In the angle-and-perimeter scheme, we require for a point to be recolored, in addition to its green angle being larger than $\Phi \geq 180^\circ$, that this recoloring decreases the perimeter of the separating polygon(s). Since we are interested in a small perimeter polygon, this is a natural choice. When there are several choices of recoloring a point, we select the one that has the largest green angle.

Theorem 5. *The number of recolorings in the angle-and-perimeter recoloring scheme is at least $\Omega(n^2)$ and at most $2^n - 1$.*

To implement the algorithm efficiently, we maintain the subset of points that can be recolored, sorted by decreasing green angle, in a balanced binary search tree. We extract the point p with largest green angle, recolor it, and recolor the incident edges. This can be done in time linear in the degree of p . We must also examine the neighbors of p . They may get a different green angle, which we must recompute. We must also test if recoloring a neighbor still decreases the perimeter length. This can be done for each neighbor in time linear in its degree.

Theorem 6. *The running time for the angle-and-perimeter recoloring algorithm is $O(Z \cdot n \log n)$, where Z denotes the actual number of recolorings.*

3.2 The Angle-and-Degree Scheme

In the angle-and-degree scheme we use the same angle condition as before, complemented by requiring that the number of green edges decreases. For any red (blue) point p , we define $\delta(p)$ to be the difference between the number of green edges and the number of red (blue) edges incident to p . We recolor a point p if its green angle ϕ is at least some threshold Φ and its δ -value is larger than some threshold $\delta_0 \geq 1$. We always choose the point with largest δ -value, and among these, the largest green angle. In every recoloring the number of green edges in $DT(P)$ decreases, so we get a linear number of recolorings.

Theorem 7. *The angle-and-degree recoloring algorithm requires $O(n^2 \log n)$ time.*

4 Experiments

In cooperation with our partners in the SPIRIT project [9] we got four data sets: Eastanglia (14, 57), Midlands (56, 52), Southeast (51, 49), and Wales (72, 54). The numbers in parentheses refer to the numbers of red and blue points in each data set, respectively. The red points were determined by Web searches using www.google.uk in Sept.'04 and trigger phrases such as “located in the Midlands” and then extracting the names of the corresponding towns and cities in the search results. The coordinates of the towns were looked up in the SPIRIT ontology. Finally the same ontology was queried with an axis-parallel rectangle 20% larger than the bounding box of the red points. The blue points were defined to be those points in the query result that were not red. The exact procedure is described in [1]. More experimental results are described in the full version [13].

We have implemented both the adaptation and the recoloring method, and we show and discuss a few screen shots. Figure 3 features the adaptation method for two different values of α . The corresponding radius- α disk can be found in the lower right corner of each subfigure. Regarding the recoloring method we give an example of the angle scheme and of the angle-and-degree scheme, see Figure 4. In each figure blue points are marked by white disks and red points by black disks. Due to the alpha shape that is used as initial region in the adaptation method, the area of the resulting region increases with increasing α , see Figure 3. We found that good values of α have to be determined by hand. For smaller values of α , the alpha shape is likely to consist of several components, which leads to strange results, see Figure 3 (left). Here, the largest component captures Wales quite well, but the other components seem to make not much sense. For larger values of α the results tend to change very little, because then the alpha shape becomes similar to the convex hull of the red points. However, the value of α may not be too large since then outliers are joined in the α -shape and cause strange effects. For example, in Figure 3 (right) Wales has an enormous extent. When the initial value of α was well-chosen, the results matched the region quite well for all data sets.

For the basic recoloring method we found the best results for values of Φ that were slightly larger than 180° , say in the range 185° – 210° . Larger values of Φ severely restrict color changes. This often results in a long perimeter of the red region, which is not very natural. However, the results strongly depended on the quality of the input data. Figure 4 shows this effect: Although Wales is contained in the resulting region, the region is too large. Too many points were falsely classified positive. The quality of the Eastanglia data was better, thus the resulting region nearly matches the typical extent of Eastanglia.

For a small degree threshold, say $\delta_0 \leq 4$, the angle-and-degree scheme yields nearly the same results as the angle scheme, as points having a green angle larger than 180° are likely to have a positive δ -value. For increasing values of δ_0 the

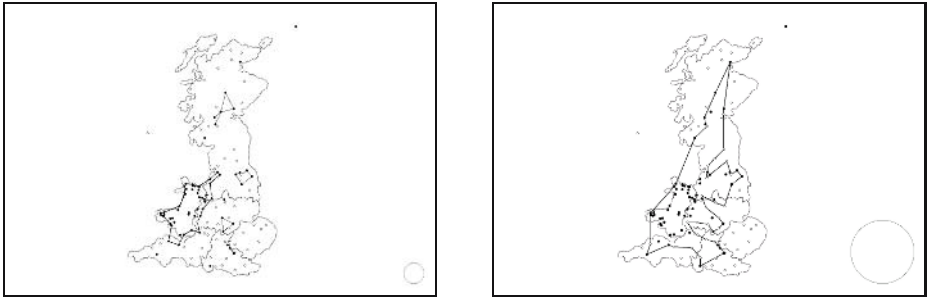


Fig. 3. Regions for *Wales* computed by the adaptation method. The α -values are shown as radius- α disks in the lower right corner.



Fig. 4. Region for *Wales* computed by the angle scheme ($\Phi = 185^\circ$, left) and for *Eastanglia* computed by the angle-and-degree scheme ($\Phi = 200^\circ$ and $\delta_0 = 4$, right)

results depend less and less on the angle threshold Φ . If a point has a δ -value above 4, then its green angle is usually large anyway. However, if the main goal is not the compactness of the region, or if the input is reasonably precise, larger values of δ_0 can also yield good results, see Figure 4 (right) where only the two light-shaded points were recolored.

For random data Figure 5 shows how the number of recolorings (y -axis) depends on the angle threshold Φ (left, x -axis) for sets of size $n = 800$ and on the number of points (right, x -axis) for a fixed angle threshold of $\Phi = 210^\circ$. In both graphs, a data point represents the average number of recolorings over 30 instances. The error bars show the minimum and maximum total number of recolorings that occurred among the 30 instances. For $n = 800$ the data sets were generated by picking 400 blue points and 100 red points uniformly distributed over the unit square. Then, another 300 red points were uniformly distributed over a radius-1/4 disk centered on the square. It is interesting to see that the number of recolorings seems to scale perfectly with the number of points.

The strength of the recoloring method is its ability to eliminate false positives provided that they are not too close to the target region. Since the differences between the various schemes we investigated seem to be small, a scheme that is easy to implement and terminates quickly can be chosen, e.g. the preferential-red or preferential-blue scheme.

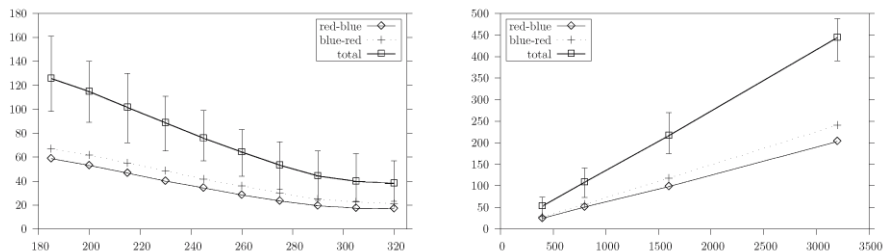


Fig. 5. Number of recolorings as a function of Φ (left) for random data sets of size $n = 800$ and as a function of the number of points (right) for random data sets and fixed angle threshold $\Phi = 210^\circ$

Comparing the adaptation method and the recoloring scheme shows that the two schemes behave similarly on the inner part, the “core”, of a point set. The main differences occur along the boundaries. The adaptation method may produce more fjord-like boundaries than the recoloring scheme. For example, compare Figure 3 (right) and Figure 4 (left).

The given real-world data appeared to be noisy, which influences the outcome, however, we think that the results are promising. Both improved data and more refined variations of our methods (e.g. using confidence values for inside-outside classification) should lead to better boundaries.

5 Conclusions

This paper discussed the problem of computing a reasonable boundary for an imprecise geographic region based on noisy data. Using the Web as a large database, it is possible to find cities and towns that are likely to be inside and cities and towns that are likely to be outside the imprecise region. We presented two basic approaches to determine a boundary. The first was to formulate the problem as a minimum perimeter polygon computation, based on an initial red polygon and additional points that must be outside (blue) or stay inside (red).

The second approach involved changing the color, or inside-outside classification of points if they are surrounded by points of the other color. We proved a few lower and upper bounds on the number of recolorings for different criteria of recoloring. An interesting open problem is whether the most general version of this recoloring method terminates or not. In tests we always had less than n recolorings. Furthermore, different schemes gave nearly the same result.

We also presented test results of our algorithms, based on real-world data, which included places obtained from trigger phrases for several British regions. Indeed the data appeared to be noisy, which is one reason why the boundaries determined were not always acceptable.

Acknowledgements. Iris Reinbacher was partially supported by a travel grant of the Netherlands Organization for Scientific Research (NWO). J. Mitchell is

partially supported by the NSF (CCR-0098172, ACI-0328930, CCF-0431030), NASA (NAG2-1620), Metron Aviation, and the US-Israel Binational Science Foundation (2000160).

We thank Subodh Vaid, Hui Ma, and Markus Völker for implementation, Paul Clough and Hideo Joho for providing the data, and Emo Welzl for useful ideas.

References

1. A. Arampatzis, M. van Kreveld, I. Reinbacher, C. B. Jones, S. Vaid, P. Clough, H. Joho, and M. Sanderson. Web-based delineation of imprecise regions. Manuscript, 2005.
2. E. M. Arkin, J. S. B. Mitchell, and C. D. Piatko. Minimum-link watchman tours. *Inform. Process. Lett.*, 86(4):203–207, May 2003.
3. S. Arora and K. Chang. Approximation schemes for degree-restricted MST and red-blue separation problem. *Algorithmica*, 40(3):189–210, 2004.
4. G. S. Brodal and R. Jacob. Dynamic planar convex hull. In *Proc. 43rd IEEE Sympos. Found. Comput. Sci.*, pages 617–626, 2002.
5. T. M. Chan. Low-dimensional linear programming with violations. In *Proc. 43rd IEEE Symp. on Foundations of Comput. Sci. (FOCS'02)*, pages 570–579, 2002.
6. P. Eades and D. Rappaport. The complexity of computing minimum separating polygons. *Pattern Recogn. Lett.*, 14:715–718, 1993.
7. H. Edelsbrunner. *Algorithms in Combinatorial Geometry*, volume 10 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Heidelberg, 1987.
8. L. J. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. E. Tarjan. Linear-time algorithms for visibility and shortest path problems inside triangulated simple polygons. *Algorithmica*, 2:209–233, 1987.
9. C. Jones, R. Purves, A. Ruas, M. Sanderson, M. Sester, M. van Kreveld, and R. Weibel. Spatial information retrieval and geographical ontologies – an overview of the SPIRIT project. In *Proc. 25th Annu. Int. Conf. on Research and Development in Information Retrieval (SIGIR 2002)*, pages 387–388, 2002.
10. J. S. B. Mitchell. Guillotine subdivisions approximate polygonal subdivisions: A simple polynomial-time approximation scheme for geometric TSP, k -MST, and related problems. *SIAM J. Comput.*, 28:1298–1309, 1999.
11. Y. Morimoto, M. Aono, M. Houle, and K. McCurley. Extracting spatial knowledge from the Web. In *Proc. IEEE Sympos. on Applications and the Internet (SAINT'03)*, pages 326–333, 2003.
12. D. O'Sullivan and D. J. Unwin. *Geographic Information Analysis*. John Wiley & Sons Ltd, 2003.
13. I. Reinbacher, M. Benkert, M. van Kreveld, J. S. Mitchell, and A. Wolff. Delineating boundaries for imprecise regions. Tech. Rep. UU-CS-2005-026, Utrecht University.
14. C. Seara. *On Geometric Separability*. PhD thesis, UPC Barcelona, 2002.

EXACUS: Efficient and Exact Algorithms for Curves and Surfaces*

Eric Berberich¹, Arno Eigenwillig¹, Michael Hemmer², Susan Hert³, Lutz Kettner¹,
Kurt Mehlhorn¹, Joachim Reichel¹, Susanne Schmitt¹, Elmar Schömer²,
and Nicola Wolpert¹

¹Max-Planck-Institut für Informatik, Saarbrücken, Germany

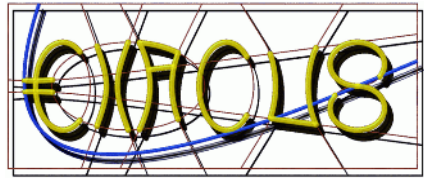
²Johannes-Gutenberg-Universität Mainz, Germany

³Serials Solutions, Seattle, WA., USA

Abstract. We present the first release of the EXACUS C++ libraries. We aim for systematic support of non-linear geometry in software libraries. Our goals are efficiency, correctness, completeness, clarity of the design, modularity, flexibility, and ease of use. We present the generic design and structure of the libraries, which currently compute arrangements of curves and curve segments of low algebraic degree, and boolean operations on polygons bounded by such segments.

1 Introduction

The EXACUS-project (*Efficient and Exact Algorithms for Curves and Surfaces*¹) aims to develop efficient, exact (the mathematically correct result is computed), and complete (for all inputs) algorithms and implementations for low-degree non-linear geometry.



Exact and complete methods are available in the algebraic geometry community, but they are not optimized for low-degree or large inputs. Efficient, but inexact and incomplete methods are available in the solid modeling community. We aim to show that exactness and completeness can be obtained at a moderate loss of efficiency. This requires theoretical progress in computational geometry, computer algebra, and numerical methods, and a new software basis. In this paper, we present the design of the EXACUS C++ libraries. We build upon our experience with CGAL [12,22] and LEDA [25].

CGAL, the *Computational Geometry Algorithms Library*, is the state-of-the-art in implementing geometric algorithms completely, exactly, and efficiently. It deals mostly with linear objects and arithmetic stays within the rational numbers.

Non-linear objects bring many new challenges with them: (1) Even single objects, e.g., a single algebraic curve, are complex, (2) there are many kinds of degeneracies, (3) and point coordinates are algebraic numbers. It is not clear yet, how to best cope with these challenges. The answer will require extensive theoretical and experimental work.

* Partially supported by the IST Programme of the EU as a Shared-cost RTD (FET Open) Project No IST-2000-26473 (ECG - Effective Computational Geometry for Curves and Surfaces).

¹ <http://www.mpi-inf.mpg.de/EXACUS/>

Therefore, we aim for a crisp and clear design, flexibility, modularity, and ease of use of our libraries.

The EXACUS C++ libraries compute arrangements of curves and curve segments of low algebraic degree, and boolean operations on polygons bounded by such segments. The functionality of our implementation is complete. We support arcs going to infinity and isolated points. We deal with all degeneracies, such as singularities and intersections of high multiplicity. We always compute the mathematically correct result. Important application areas are CAD, GIS and robotics. The recent Open Source release contains the full support for conics and conic segments, while existing implementations for cubic curves and a special family of degree four curves are scheduled for later releases. We work currently on extending our results to arrangements of general-degree curves in the plane [29] and quadric surfaces in space and boolean operations on them [4].

Here, we present the generic design and the structure of the EXACUS C++ libraries. The structure of the algorithms and algebraic methods is reflected in a layered architecture that supports experiments at many levels. We describe the interface design of several important levels. Two interfaces are improvements of interfaces in CGAL, the others are novelties. This structure and design has proven valuable in our experimental work and supports future applications, such as spatial arrangements of quadrics.

2 Background and Related Work

The theory behind EXACUS is described in the following series of papers: Berberich et al. [3] computed arrangements of conic arcs based on the LEDA [25] implementation of the Bentley-Ottmann sweep-line algorithm [2]. Eigenwillig et al. [10] extended the sweep-line approach to cubic curves. A generalization of Jacobi curves for locating tangential intersections is described by Wolpert [31]. Berberich et al. [4] recently extended these techniques to special quartic curves that are projections of spatial silhouette and intersection curves of quadrics and lifted the result back into space. Seidel et al. [29] describe a new method for general-degree curves. Eigenwillig et al. [9] extended the Descartes algorithm for isolating real roots of polynomials to bit-stream coefficients.

Wein [30] extended the CGAL implementation of planar maps to conic arcs with an approach similar to ours. However, his implementation is restricted to bounded curves (i.e. ellipses) and bounded arcs (all conic types), and CGAL's planar map cannot handle isolated points. He originally used Sturm sequences with separation bounds and switched now to the Expr number type of CORE [21] to handle algebraic numbers.

Emiris et al. [11] proposed a design for a support of non-linear geometry in CGAL focusing on arrangements of curves. Their implementation was limited to circles and had preliminary results for ellipsoidal arcs. Work on their kernel has continued but is not yet available. Their approach is quite different from ours: They compute algebraic numbers for both coordinates of an intersection point, while we need only the x -coordinate as algebraic number. On the other hand, they use statically precomputed Sturm sequences for algebraic numbers of degree up to four, an approach comparing favorably with our algebraic numbers. However, their algorithm for arrangement computations requires evaluations of the curve at points with algebraic numbers as coordinates.

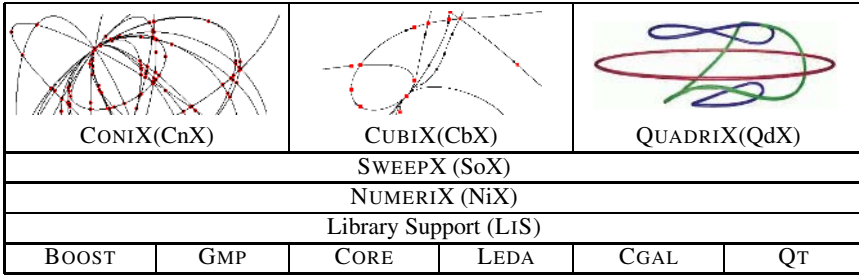


Fig. 1. Layered architecture of the EXACUS C++ libraries (with their name abbreviations)

The libraries MAPC [23] and ESOLID [8] deal with algebraic points and curves and with low-degree surfaces, respectively. Both libraries are not complete, e.g., require surfaces to be in general position.

Computer algebra methods, based on exact arithmetic, guarantee correctness of their results. A particularly powerful method related to our problems is Cylindrical Algebraic Decomposition invented by Collins [6] and subsequently implemented (with numerous refinements). Our approach to curve and curve pair analyses can be regarded as a form of cylindrical algebraic decomposition of the plane in which the lifting phase has been redesigned to take advantage of the specific geometric setting; in particular, to keep the degree of algebraic numbers low.

3 Overview and Library Structure

The design of the 94.000 lines of source code and documentation of the EXACUS C++ libraries follows the *generic programming paradigm* with C++ templates, as it is widely known from the *Standard Template Library*, STL [1], and successfully used in CGAL [12,5,19]. We use *concepts*² from STL (iterators, containers, functors) and CGAL (geometric traits class, functors) easing the use of the EXACUS libraries. C++ templates provide flexibility that is resolved at compile-time and hence has no runtime overhead. This is crucial for the efficiency and flexibility at the number type level (incl. machine arithmetic), however, is insignificant at higher levels of our design. In particular, a large part of our design can also be realized in other paradigms and languages.

EXACUS uses several external libraries: BOOST (interval arithmetic), GMP and CORE (number types), LEDA (number types, graphs, other data structures, and graphical user interface), CGAL (number types and arrangements), and Qt (graphical user interface). Generic programming allows us to avoid hard-coding dependencies. For example, we postpone the decision between alternative number type libraries to the final application code.

We organized the EXACUS libraries in a layered architecture, see Figure 1, with external libraries at the bottom and applications at the top. In between, we have li-

² A *concept* is a set of syntactical and semantical requirements on a template parameter, and a type is called a *model* of a *concept* if it fulfills these requirements and can thus be used as an argument for the template parameter [1].

brary support in LIS, number types, algebraic and numerical methods in NUMERIX, and the generic implementation of a sweep-line algorithm and a generic generalized polygon that supports regularized boolean operations on regions bounded by curved arcs in SWEEPX. Furthermore, SWEEPX contains *generic algebraic points and segments* (GAPS). It implements the generic and curve-type independent predicates and constructions for the sweep-line algorithm.

In this paper, we focus on the interface design between NUMERIX, SWEEPX, GAPS, and the applications. Descriptions of the application layer can be found in [3,10,4].

4 NUMERIX Library

The NUMERIX library comprises number types, algebraic constructions to build types from types, such as polynomials (over a number type), vectors, and matrices, and a tool box of algorithms solving linear systems (Gauss-Jordan elimination), computing determinants of matrices (Bareiss and division-free method of Berkowitz [26]), gcds, Sylvester and Bézout matrices for resultants and subresultants, isolating real roots of polynomials (Descartes algorithm), and manipulating algebraic numbers. We import the basic number types integer, rationals and (optionally) real expressions from LEDA [25], GMP [17] and CORE [21], or EXT [28]. We next discuss core aspects of NUMERIX.

Number Type Concepts and Traits Classes: We aim for the Real RAM model of computation. An effective realization must exploit the tradeoff between expressiveness and efficiency of different number type implementations. In EXACUS, we therefore provide a rich interface layer of number type concepts as shown in Figure 2. It allows us to write generic and flexible code with maximal reuse. All number types must provide the construction from small integers, in particular from 0 and 1. `IntegralDomain`, `UFDomain`, `Field`, and `EuclideanRing` correspond to the algebraic concepts with the same name. `FieldWithSqrt` are fields with a square root operator. The concept `IntegralDomainWithoutDiv` also corresponds to integral domains in the algebraic sense; the distinction results from the fact that some implementations of integral domains, e.g., `CGAL::MP_Float`, lack the (algebraically always well defined) integral division. A number type may be ordered or not; this is captured in the `RealComparable` concept, which is orthogonal to the other concepts. The fields \mathbb{Q} and $\mathbb{Z}/p\mathbb{Z}$ are ordered and not ordered respectively.

The properties of number types are collected in appropriate traits classes. Each concrete number type `NT` knows the (most refined) concept to which it belongs; it is encoded in `NT_traits<NT>::Algebra_type`. The usual arithmetic and comparison operators are required to be realized via C++ operator overloading for ease of use.

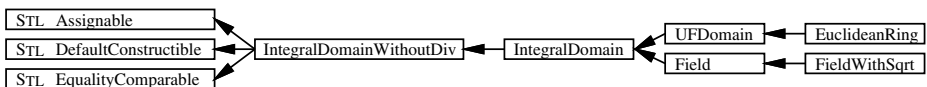


Fig. 2. The number type concepts in EXACUS. They refine three classical STL concepts.

The division operator is reserved for division in fields. All other unary (e.g., sign) and binary functions (e.g., integral division, gcd) must be models of the standard STL `AdaptableUnaryFunction` or `AdaptableBinaryFunction` concept and local to a traits class (e.g., `NT_traits<NT>::Integral_div`). This allows us to profit maximally from all parts in the STL and its programming style.

Design Rationale: Our interface extends the interface in CGAL [12,5,19,22] that only distinguishes `EuclideanRing`, `Field`, and `FieldWithSqrt`. The finer granularity is needed for the domain of curves and surfaces, in particular, the algebraic numbers and algorithms on polynomials. We keep the arithmetic and comparison operators for ease of use. Their semantic is sufficiently standard to presume their existence and compliance in existing number type libraries for C++. For the other functions we prefer the trouble-free and extendible traits class solution; this was suggested by the name-lookup and two-pass template compilation problems experienced in CGAL.

Polynomials: The class `Polynomial<NT>` realizes polynomials with coefficients of type `NT`. Depending on the capabilities of `NT`, the polynomial class adapts internally and picks the best implementation for certain functions (see below for an example). The number type `NT` must be at least of the `IntegralDomainWithoutDiv` concept. For all operations involving division, the `IntegralDomain` concept is required. Some functions require more, for example, the gcd-function requires `NT` to be of the `Field` or `UFDomain` concept. In general, the generic implementation of the polynomial class encapsulates the distinction between different variants of functions at an early level and allows the reuse of generic higher-level functions.

The `Algebra_type` of `Polynomial<NT>` is determined via template meta-programming by the `Algebra_type` of `NT`. It remains the same except in the case of `EuclideanRing`, which becomes a `UFDomain`, and both field concepts, which become an `EuclideanRing`. `NT` can itself be an instance of `Polynomial`, yielding a recursive form of multivariate polynomials. In our applications, we deal only with polynomials of small degree (so far at most 16) and a small number of variables (at most 3) and hence the recursive construction is appropriate. Some convenience functions hide the recursive construction in the bi- and trivariate case.

We use polynomial remainder sequences (PRS) to compute the gcd of two polynomials (uni- or multivariate). Template meta-programming is used to select the kind of PRS: Euclidean PRS over a field and Subresultant PRS (see [24] for exposition and history) over a UFD. An additional meta-programming wrapper attempts to make the coefficients fraction-free, so that gcd computation over the field of rational numbers is actually performed with integer coefficients and the Subresultant PRS (this is faster). Gcd computations are relatively expensive. Based on modular resultant computation, we provide a fast one-sided probabilistic test for coprimality and squarefreeness that yields a significant speedup for non-degenerate arrangement computations [18]. We offer several alternatives for computing the resultant of two polynomials: via the Subresultant PRS or as determinant of the Sylvester or Bézout matrix [15, ch. 12]. Evaluating a Bézout determinant with the method of Berkowitz [26] can be faster than the Subresultant PRS for bivariate polynomials of small degrees over the integers. The Bézout determinant can also express subresultants, see e.g. [16,20].

Algebraic Numbers: The `Algebraic_real` class represents a real root of a square-free polynomial. The representation consists of the defining polynomial and an isolating interval, which is an open interval containing exactly one root of the polynomial. Additionally, the polynomial is guaranteed to be non-zero at the endpoints of the interval. As an exception, the interval can collapse to a single rational value when we learn that the root has this exact rational value.

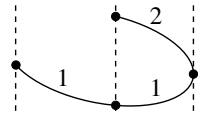
We use the Descartes Method [7,27] to find isolating intervals for all real roots of a polynomial. We have a choice of different implementations, of which Interval Descartes [9] and Sturm sequences are ongoing work. We cross link all real roots of a polynomial, such that, for example, if one number learns how to factorize the defining polynomial, all linked numbers benefit from that information and simplify their representation. We learn about such factorizations at different occasions in our algorithms, e.g., if we find a proper common factor in the various gcd computations.

Interval refinement is central to the Descartes Method, to the comparison of algebraic numbers, and to some algorithms in the application layer [10]. We expose the refinement step in the following interface: `refine()` bisects the isolating interval. `strong_refine(Field m)` refines the isolating interval until m is outside of the closed interval. `refine_to(Field lo, Field hi)` intersects the current interval with the isolating interval (lo, hi) . The global function `refine_zero_against` refines an isolating interval against all roots of another polynomial. The member function `x.rational_between(y)` returns a rational number between the two algebraic reals x and y . Here, it is desirable to pick a rational number of low-bit complexity.

5 SWEEPX Library

The SWEEPX library provides a generic sweep-line algorithm and generalized polygon class that supports regularized boolean operations on regions bounded by curved arcs. We based our implementation on the sweep-line algorithm for line-segments from LEDA [25], which handles all types of degeneracies in a simple unified event type. We extended the sweep-line approach with a new algorithm to reorder curve segments continuing through a common intersection point in linear time [3] and with a *geometric traits class* design for curve segments, which we describe in more detail here.

As input, we support full curves and, for conics, also arbitrary segments of curves, but both will be preprocessed into potentially smaller *sweepable segments* suitable for the algorithm. A *sweepable segment* is x -monotone, has a constant arc number in its interior (counting without multiplicities from bottom to top), and is free of one-curve events (explained in the next section) in its interior.



The *geometric traits class* defines the interface between the generic sweep-line algorithm and the actual geometric operations performed in terms of geometric types, predicates, and constructions, again realized as functors.³ The idea of geometric traits classes goes back to CGAL [12,5]. Our set of requirements is leaner than the geometric traits class in CGAL's arrangement and planar map classes [13], however, feedback from Emiris et al. [11] and our work is leading to improvements in CGAL's interface.

³ We omit here the access functions for the functors for brevity, see [12] for the details.

A geometric traits class following the `CurveSweepTraits_2` concept needs a type `Point_2`, representing end-points as well as intersection points, and a type `Segment_2`, representing curve segments. In STL terminology, both types have to be default-constructible and assignable, and are otherwise opaque types that we manipulate only through the following predicates, accessors, and constructions. Observe the compactness of the interface.

Predicates:

- `Compare_xy_2` and `Less_xy_2` lexicographically compare two points, the former with a three-valued return type and the latter as a predicate.
- `Is_degenerate_2` returns true if a segment consists only of one point.
- `Do_overlap_2` tells whether two segments have infinitely many points in common (i.e., intersect in a non-degenerate segment).
- `Compare_y_at_x_2` returns the vertical placement of a point relative to a segment.
- `Equal_y_at_x_2` determines if a point lies on a segment (equivalent to testing `Compare_y_at_x_2` for equality, but may have a more efficient implementation).
- `Multiplicity_of_intersection` computes the multiplicity of an intersection point between two segments. It is used in the linear-time reordering of segments and hence used only for non-singular intersections.
- `Compare_y_right_of_point` determines the ordering of two segments just after they both pass through a common point. It is used for the insertion of segments starting at an event point.

Accessors and Constructions:

- `Source_2` and `Target_2` return the source and target point of a curve segment.
- `Construct_segment_2` constructs a degenerate curve segment from a point.
- `New_endpoints_2` and `New_endpoints_opposite_2` replace the endpoints of a curve segment with new representations and return this new segment. The latter functor also reverses the orientation of the segment. They are used in the initialization phase of the sweep-line algorithm where equal end-points are identified and where the segments are oriented canonically from left to right [25].
- `Intersect_2` constructs all intersection points between two segments in lexicographical order.
- `Intersect_right_of_point_2` constructs the first intersection point between two segments right of a given point. It is used only for validity checking or if caching is disabled.

We also offer a geometric traits class for CGAL's planar map and arrangement classes, actually implemented as a thin adaptor for GAPS described next. Thus, we immediately get traits classes for CGAL for all curve types available in EXACUS. Our traits class supports the incremental construction and the sweep-line algorithm in CGAL. This solution is independent of LEDA.

6 Generic Algebraic Points and Segments (GAPS)

The generic point and segment types are essentially derived from the idea of a cylindrical algebraic decomposition of the plane and a number type for x -coordinates, while

y -coordinates are represented implicitly with arc numbers on supporting curves. Studying this more closely [10], one can see that all predicates and constructions from the previous section can be reduced to situations with one or two curves only.

The software structure is now as follows: The application libraries provide a curve analysis and a curve pair analysis, which depend on the actual curve type. Based on these analyses, the GAPS part of SWEEPX implements the generic algebraic points and segments with all predicates and constructions needed for the sweep-line algorithm, which is curve-type independent generic code. Small adaptor classes present the GAPS implementation in suitable geometric traits classes for the sweep-line implementation in SWEEPX or the CGAL arrangement.

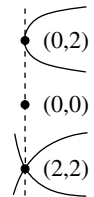
GAPS requires two preconditions; squarefreeness of the defining polynomial and coprimality of the defining polynomials in the curve pair analysis. We allow to defer the check to the first time at which any of the analysis functions is called. The application libraries may impose further restrictions on the choice of a coordinate system. If a violation is detected, an exception must be thrown that is caught outside SWEEPX. The caller can then remove the problem and restart afresh. CONIX does not restrict the choice of coordinate system. CUBIX and QUADRIX assume a generic coordinate system to simplify the analysis, see [10,4] for details.

We continue with curve and curve pair analysis, and how we handle unboundedness.

Curve Analysis: A planar curve is defined as zero set of a bivariate squarefree integer polynomial f . We look at the curve in the real affine plane per x -coordinate and see arcs at varying y -values on the fictitious vertical line $x = x_0$ at each x -coordinate x_0 . At a finite number of events, the number and relative position of arcs changes. Events are x -extreme points, singularities (such as self-intersections and isolated points) and vertical asymptotes (line components, poles).

The result of the curve analysis is a description of the curve's geometry at these events and over the open intervals of x -coordinates between them. This description is accessible through member functions in the AlgebraicCurve_2 concept. It consists of the number of events, their x -coordinates given as algebraic numbers, the number of arcs at events and between them, an inverse mapping from an x -coordinate to an event number, and it gives details for each event in objects of the Event1_info class.

The Event1_info class describes the topology of a curve over an event x -coordinate x_0 . The description answers four questions on the local topology of the curve over the corresponding x -coordinate: (1) How many real arcs (without multiplicities) intersect the (fictitious) vertical line $x = x_0$? (2) Does the curve have a vertical line component at x_0 ? (3) Is there a vertical asymptote at x_0 , and how many arcs approach it from the left and right? (4) For any intersection point of the curve with the vertical line $x = x_0$, how many real arcs on the left and right are incident to it? In the example figure, (0,0) stands for an isolated point, (0,2) for a left x -extreme point or cusp, and (2,2) for a singular point.



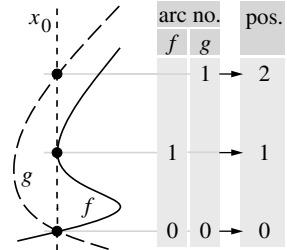
Curve Pair Analysis: We study ordered pairs of coprime planar algebraic curves f and g . Similar to the curve analysis, we look at the pair of curves in the real affine plane per x -coordinate and see arcs of both curves at varying y -values above each x -coordinate. At a finite number of events, the number and relative position of arcs changes: There

are intersections at which arcs of both curves run together, there are one-curve events, and there are events that are both.

The result of the curve pair analysis is a description of the curves' geometry at these events and over the open intervals of x -coordinates between them. This description is accessible through member functions in the `AlgebraicCurvePair_2` concept. It consists of various mappings of event numbers to indices of roots in the resultants $\text{res}(f, f_y)$, $\text{res}(g, g_y)$, and $\text{res}(f, g)$, to x -coordinates and reverse, and it gives details for each event in objects of the `Event2_slice` class.

An `Event2_slice` is a unified representation of the sequence of arcs over some x -coordinate x_0 , be there an event or not [10,4]. A slice of a pair (f, g) at x_0 describes the order of points of f and g lying on the fictitious vertical line $x = x_0$, sorted by y -coordinate. Points are counted without multiplicities.

Let the *arc number* of a point on f be its rank in the order amongst all points of f on $x = x_0$, and respectively for points on g . Let the *position number* of a point on f or g be its rank in the order amongst all points of $f \cup g$ on $x = x_0$. All ranks are counted, starting at 0, in increasing order of y -coordinates. An important function in the slice representation is the mapping of arc numbers, used in the point and sweepable segment representations, to position numbers, used to compare y -coordinates of points on different curves f and g . For the example in the figure, this mapping will tell us that the second point of g is the third amongst all points (w.r.t. y -coordinates), i.e., it maps arc number 1 to position 2.



Unbounded Curves and Segments: The implementation is not restricted to bounded curves and segments. We define symbolic “endpoints” at infinity, such that unbounded curves and segments, e.g., the hyperbola $xy - 1 = 0$ or any part of it, can be included in a uniform representation of sweepable segments. We use two techniques, compactification and symbolic perturbation: We add minus- and plus-infinity symbolically to the range of x -coordinate values. Then, we perturb x -coordinates to represent “endpoints” of curves that approach a pole and unbounded “endpoints” of vertical lines and rays. Let $\varepsilon > 0$ be an infinitesimal symbolic value. We perturb the x coordinate of an endpoint of a curve approaching a pole from the left by $-\varepsilon$, the lower endpoint of a vertical line by $-\varepsilon^2$, the upper endpoint of a vertical line by ε^2 , and an endpoint of a curve approaching a pole from the right by ε . We obtain the desired lexicographic order of event points.

We also extend the curve analyses with the convention that for x -coordinates of minus- or plus-infinity we obtain the number and relative position of arcs before the first or after the last event, respectively. Note that this is not “infinity” in the sense of projective geometry.

7 Evaluation and Conclusion

The libraries are extensively tested and benchmarked with data sets showing runtime behavior, robustness, and completeness. In particular for completeness we manufactured test data sets for all kinds of degeneracies [3,10,4]. A preliminary comparison [14] between Wein [30], Emiris et al. [11], and us showed that our implementation was the

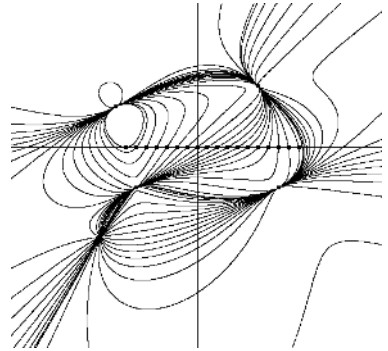
only robust and stable and almost always the fastest implementation at that time. In particular, the other implementations could not handle large random instances, degenerate instances of several ellipses intersecting in the same point or intersecting tangentially, and instances of ellipses with increasing bit-size of the coefficients. A new comparison is planned when the other implementations have stabilized.

The report [14] also contains a comparison of our sweep-line algorithm with the CGAL sweep-line algorithm used for the planar map with intersections. We briefly summarize the result in the table that shows running times in seconds of both implementations measured for three random instances of cubic curves of 30, 60, and 90 input curves respectively on a Pentium III at 1.2 GHz, Linux 2.4, g++ 3.3.3, with `-DNDEBUG`, LEDA 4.4.1, and CGAL 3.0.1.

Data set	Segs	Vertices	Halfedges	SoX	CGAL
random30	266	2933	11038	6.7	8.0
random60	454	11417	44440	27.7	34.6
random90	680	26579	104474	67.5	81.2

Profiling the executions on the “random30” instance exhibits a clear difference in the number of predicate calls. The dominant reason for this is certainly reordering by comparison (CGAL) as opposed to reordering by intersection multiplicities (SoX).

Additionally, we report the running time of a hand-constructed instance: a pencil of 18 curves intersecting in five distinct points, in four of them with multiplicity 2, and nowhere else. Here `SoX::sweep_curves()` can exhibit the full benefit of linear-time reordering; 1.7 seconds for SoX and 4.3 seconds for CGAL.



All experiments show an advantage for our implementation. We took care not to use examples that would penalize the CGAL implementation with the interface mapping to the CUBIX implementation (see [14]), but the implementations are too complex to allow a definitive judgment on the relative merits of the two implementations based on above numbers.

We conclude with a few remarks on efficiency: Arithmetic is the bottleneck, so we use always the simplest possible number type, i.e., integers where possible, then rationals, LEDA real or CORE, and algebraic numbers last. However, using LEDA reals naively, for example, to test for equality or in near equality situations, can be quite costly, which happened in the first CONIX implementation, and using algebraic numbers or a number type that extends rationals with one or two fixed roots instead can be beneficial. The released CONIX implementation has been improved considerably following the ideas developed for the CUBIX implementation. Furthermore, we use caching of the curve analysis and we compute all roots of a polynomial at once. All roots of the same polynomial are cross linked, and if one root happens to learn about a factorization of the polynomial, all other roots will also benefit from the simplified representation. We use modular arithmetic as a fast filter test to check for inequality.

So far we have almost exclusively worked with exact arithmetic and optimized its runtime. Floating-point filters have been only used insofar that they are integral part of some number types. More advanced filters are future work.

We use the sweep-line algorithm for computing arrangements, but since arithmetic is the obvious bottleneck, it might pay off to consider incremental construction with its lower algebraic degree in the predicates and the better asymptotic runtime.

CGAL plans a systematic support for non-linear geometry. We contribute to this effort with our design experience and implementations presented here.

Acknowledgements

We would like to acknowledge contributions by Michael Seel, Evghenia Stegantova, Dennis Weber, and discussions with Sylvain Pion.

References

1. M. H. Austern. *Generic Programming and the STL*. Addison-Wesley, 1998.
2. J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.*, C-28(9):643–647, September 1979.
3. E. Berberich, A. Eigenwillig, M. Hemmer, S. Hert, K. Mehlhorn, and E. Schömer. A computational basis for conic arcs and boolean operations on conic polygons. In *ESA 2002, LNCS 2461*, pages 174–186, 2002.
4. E. Berberich, M. Hemmer, L. Kettner, E. Schömer, and N. Wolpert. An exact, complete and efficient implementation for computing planar maps of quadric intersection curves. In *Proc. 21th Annu. Sympos. Comput. Geom.*, pages 99–106, 2005.
5. H. Brönnimann, L. Kettner, S. Schirra, and R. Veltkamp. Applications of the generic programming paradigm in the design of CGAL. In M. Jazayeri, R. Loos, and D. Musser, editors, *Generic Programming—Proceedings of a Dagstuhl Seminar*, LNCS 1766, pages 206–217. Springer-Verlag, 2000.
6. G. E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *Proc. 2nd GI Conf. on Automata Theory and Formal Languages*, volume 6, pages 134–183. LNCS, Springer, Berlin, 1975. Reprinted with corrections in: B. F. Caviness and J. R. Johnson (eds.), *Quantifier Elimination and Cylindrical Algebraic Decomposition*, 85–121. Springer, 1998.
7. G. E. Collins and A.-G. Akritas. Polynomial real root isolation using Descartes’ rule of sign. In *SYMSAC*, pages 272–275, 1976.
8. T. Culver, J. Keyser, M. Foskey, S. Krishnan, and D. Manocha. Esolid - a system for exact boundary evaluation. *Computer-Aided Design (Special Issue on Solid Modeling)*, 36, 2003.
9. A. Eigenwillig, L. Kettner, W. Krandick, K. Mehlhorn, S. Schmitt, and N. Wolpert. A Descartes algorithm for polynomials with bit-stream coefficients. In *Proc. 8th Int. Workshop on Computer Algebra in Scient. Comput. (CASC)*, LNCS. Springer, 2005. to appear.
10. A. Eigenwillig, L. Kettner, E. Schömer, and N. Wolpert. Complete, exact, and efficient computations with cubic curves. In *Proc. 20th Annu. Sympos. Comput. Geom.*, pages 409–418, 2004. accepted for *Computational Geometry: Theory and Applications*.
11. I. Z. Emiris, A. Kakargias, S. Pion, M. Teillaud, and E. P. Tsigaridas. Towards and open curved kernel. In *Proc. 20th Annu. Sympos. Comput. Geom.*, pages 438–446, 2004.
12. A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL, the computational geometry algorithms library. *Softw. – Pract. and Exp.*, 30(11):1167–1202, 2000.
13. E. Flato, D. Halperin, I. Hanniel, O. Nechushtan, and E. Ezra. The design and implementation of planar maps in CGAL. *ACM Journal of Experimental Algorithmics*, 5, 2000. Special Issue, selected papers of the Workshop on Algorithm Engineering (WAE).

14. E. Fogel, D. Halperin, R. Wein, S. Pion, M. Teillaud, I. Emiris, A. Kakargias, E. Tsigaridas, E. Berberich, A. Eigenwillig, M. Hemmer, L. Kettner, K. Mehlhorn, E. Schömer, and N. Wolpert. Preliminary empirical comparison of the performance of constructing arrangements of curved arcs. Technical Report ECG-TR-361200-01, Tel-Aviv University, INRIA Sophia-Antipolis, MPI Saarbrücken, 2004.
15. I. M. Gelfand, M. M. Kapranov, and A. V. Zelevinsky. *Discriminants, Resultants and Multi-dimensional Determinants*. Birkhäuser, Boston, 1994.
16. R. N. Goldman, T. W. Sederberg, and D. C. Anderson. Vector elimination: A technique for the implicitization, inversion, and intersection of planar parametric rational polynomial curves. *CAGD*, 1:327–356, 1984.
17. T. Granlund. *GNU MP, The GNU Multiple Precision Arithmetic Library, version 2.0.2*, 1996.
18. M. Hemmer, L. Kettner, and E. Schömer. Effects of a modular filter on geometric applications. Technical Report ECG-TR-363111-01, MPI Saarbrücken, 2004.
19. S. Hert, M. Hoffmann, L. Kettner, S. Pion, and M. Seel. An adaptable and extensible geometry kernel. In *Proc. 5th Workshop on Algorithm Engineering (WAE'01)*, LNCS 2141, pages 76–91, Aarhus, Denmark, August 2001. Springer-Verlag.
20. X. Hou and D. Wang. Subresultants with the Bézout matrix. In *Proc. Fourth Asian Symp. on Computer Math. (ASCM 2000)*, pages 19–28. World Scientific, Singapore New Jersey, 2000.
21. V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A core library for robust numeric and geometric computation. In *Proc. 15th Annu. Sympos. Comput. Geom.*, pages 351–359, 1999.
22. L. Kettner and S. Näher. Two computational geometry libraries: LEDA and CGAL. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Disc. and Comput. Geom.*, pages 1435–1463. CRC Press, second edition, 2004.
23. J. Keyser, T. Culver, D. Manocha, and Shankar Krishnan. MAPC: A library for efficient and exact manipulation of algebraic points and curves. In *Proc. 15th Annu. Sympos. Comput. Geom.*, pages 360–369, 1999.
24. R. Loos. Generalized polynomial remainder sequences. In B. Buchberger, G. E. Collins, and R. Loos, editors, *Computer Algebra: Symbolic and Algebraic Computation*, pages 115–137. Springer, 2nd edition, 1983.
25. K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, UK, 2000.
26. G. Rote. Division-free algorithms for the determinant and the pfaffian: algebraic and combinatorial approaches. In H. Alt, editor, *Computational Discrete Mathematics*, pages 119–135. Springer-Verlag, 2001. LNCS 2122.
27. F. Rouillier and P. Zimmermann. Efficient isolation of polynomial's real roots. *J. Comput. Applied Math.*, 162:33–50, 2004.
28. S. Schmitt. The diamond operator – implementation of exact real algebraic numbers. In *Proc. 8th Internat. Workshop on Computer Algebra in Scient. Comput. (CASC 2005)*, LNCS. Springer, 2005. to appear.
29. R. Seidel and N. Wolpert. On the exact computation of the topology of real algebraic curves. In *Proc. 21th Annual Symposium on Computational Geometry*, pages 107–115, 2005.
30. R. Wein. High level filtering for arrangements of conic arcs. In *ESA 2002, LNCS 2461*, pages 884–895, 2002.
31. N. Wolpert. Jacobi curves: Computing the exact topology of arrangements of non-singular algebraic curves. In *ESA 2003, LNCS 2832*, pages 532–543, 2003.

Min Sum Clustering with Penalties

Refael Hassin and Einat Or

Department of Statistics and Operations Research, Tel Aviv University,
Tel Aviv, 69978, Israel
{hassin, eior}@post.tau.ac.il

Abstract. Traditionally, clustering problems are investigated under the assumption that all objects must be clustered. A shortcoming of this formulation is that a few distant objects, called *outliers*, may exert a disproportionately strong influence over the solution. In this work we investigate the k -MIN-SUM clustering problem while addressing outliers in a meaningful way.

Given a complete graph $G = (V, E)$, a weight function $w : E \rightarrow \mathbb{N}_0$ on its edges, and $p : V \rightarrow \mathbb{N}_0$ a penalty function on its nodes, the PENALIZED k -MIN-SUM PROBLEM is the problem of finding a partition of V to $k + 1$ sets, $\{S_1, \dots, S_{k+1}\}$, minimizing $\sum_{i=1}^k w(S_i) + p(S_{k+1})$, where for $S \subseteq V$ $w(S) = \sum_{e=\{i,j\} \subset S} w_e$, and $p(S) = \sum_{i \in S} p_i$.

We offer an efficient 2-approximation to the penalized 1-min-sum problem using a primal-dual algorithm. We prove that the penalized 1-min-sum problem is NP-hard even if w is a metric and present a randomized approximation scheme for it. For the metric penalized k -min-sum problem we offer a 2-approximation.

1 Introduction

Traditionally clustering problems are investigated under the assumption that all objects must be clustered. A significant shortcoming of this formulation is that a few very distant objects, called *outliers*, may exert a disproportionately strong influence over the solution. In this work we investigate the k -MIN-SUM clustering problem while addressing outliers in a meaningful way.

Given a complete graph $G = (V, E)$, and a weight function $w : E \rightarrow \mathbb{N}_0$, on its edges, let $w(S) = \sum_{e=\{i,j\} \subset S} w_e$. The k -MIN-SUM problem is the problem of finding a partition of V to k sets, $\{S_1, \dots, S_k\}$, as to minimize $\sum_{i=1}^k w(S_i)$. This problem is NP-hard even for $k = 2$ [8], and has no known constant approximation ratio for $k > 2$ unless P=NP (k -coloring can be reduced to k -min-sum. However if w is a metric, then there is a randomized PTAS for any fixed k [4]. In the following we generalize the k -MIN-SUM problem and allow outliers that are not clustered. Let $p : V \rightarrow \mathbb{N}_0$ denote the penalty function on the nodes of G . For $S \subseteq V$ let $p(S) = \sum_{i \in S} p_i$. The PENALIZED k -MIN-SUM CLUSTERING PROBLEM (we denote this problem by k PMS, and use PMS to denote 1PMS.) is the problem of finding k disjoint subsets $S_1, \dots, S_k \subseteq V$ minimizing $\sum_{i=1}^k w(S_i) + \sum_{v \in V \setminus \cup_{i=1}^k S_i} p_v$. The formulation of the k PMS makes no assumptions regarding the amount of outliers.

2 Related Work

PMS is related to the MINIMUM l -DISPERSION PROBLEM (l MDP), which is the problem of finding a subset $V' \subset V$ of cardinality l that minimizes $w(V')$. While in PMS the relation between the penalties and the weight on the edges determines the size of the cluster, in (l MDP) the size of the cluster is predetermined. PMS with uniform penalties is the Lagrangian relaxation of l MDP. The algorithm in Section 5.4 could be adapted to give a 2-approximation for metric l MDP (using $k = 1$). The PTAS we present in Section 5 for metric PMS is correct for metric l MDP if $l \in O(n)$, while for $l \in o(n)$ the problem of finding a PTAS remains open since a metric sample cannot be taken from the cluster. The MINIMUM l -DISPERSION PROBLEM, has a $\max\{n^{\epsilon - \frac{1}{3}}, \frac{k}{2n}\}$ approximation rate by Feige, Kortsarz and Peleg [5], and the metric case of the maximization version of l MDP has 2-approximation [10].

A 2-approximation for PMS follows directly from the approximation framework presented in [9]. A modeling of outliers for the location problems k -MEDIAN, and UNCAPACITATED FACILITY LOCATION is presented in [1,12].

Randomized PTAS for metric MAX-CUT by Fernandez de la Vega and Kenyon [2], metric MIN-BISECTION by Fernandez de la Vega, Karpinski and Kenyon [3], metric 2-MIN-SUM by Indyk [11,4], and metric k -MIN-SUM for fixed k by Fernandez de la Vega et al [4], were presented in recent years. In [4] the approximation algorithm for 2-MIN-SUM uses the PTAS of [2] for MAX-CUT when the clusters are not well separated, i.e the weight of the cut is not much greater than the weight of the clusters. In [3] the MIN-BISECTION is addressed. In this case the two sides of the bisection are not well separated, since the cut is minimized, and the problem of good estimation of the distance of a node to each side of the bisection arises. A natural approach to the problem is to use a sample of each side of the bisection as the basis of the estimation. It is proved that a sampling method referred to as *metric sampling* gives good enough estimation. A metric sample is taken from each side of the bisection. In addition to good estimation the *hybrid partitioning* method is used in the algorithm presented in [3].

3 Our Contribution

We prove that PMS is NP -hard even if w is a metric. We offer an efficient 2-approximation to PMS using a primal-dual algorithm. A randomized approximation scheme for the metric PMS where the ratio between the minimal and the maximal penalty is bounded, is presented, based on methods used to approximate MIN-BISECTION and 2-MIN-SUM [3,4]. While the approach in [4] is a PTAS for metric PMS when the cluster includes most of the nodes, it gives poor results if the cluster is smaller. The approach in [3] is the basis for a PTAS for metric PMS where the cluster and the set of non-clustered points are both large, but it gives poor approximation if one of the parts is small. Therefore we present a combination of the two approaches. For the metric k PMS we offer a 2-approximation by generalizing [7].

4 Efficient 2-Approximation for PMS

The linear programming relaxation of PMS, *LP – primal*, is:

$$\begin{aligned} \text{Minimize } & \sum_{e \in E} w_e x_e + \sum_{i \in V} p_i y_i \\ & y_i + y_j + x_e \geq 1 & \forall e = \{i, j\} \in E, \\ & x_e \geq 0 & \forall e = \{i, j\} \in E, \\ & y_i \geq 0 & \forall i \in V. \end{aligned}$$

LP-primal has half integral basic solutions [9], and a 2-approximation algorithm is presented in [9] for a family of problems with the same type of constraints with time complexity $O(mn \log(\frac{n^2}{m}))$ where $m = |E|$, which in a case of a full graph is $O(n^3)$. We present a 2-approximation algorithm, denoted by *PD*, with time complexity $O(n^2)$.

Let $\delta(i) = \{e \in E \mid e = \{i, j\} \in E, j \in V\}$. The dual of the relaxation *LP – dual* is:

$$\begin{aligned} \text{Maximize } & \sum_{e \in E} R_e \\ & R_e \leq w_e & \forall e \in E, \\ & \sum_{e \in \delta(i)} R_e \leq p_i & \forall i \in V, \\ & R_e \geq 0 & \forall e \in E. \end{aligned}$$

A *maximal solution* to *LP*-dual is a feasible solution that satisfies the following property: An increase in any variable $R_e, e \in E$, results in a non feasible solution. Algorithm *PD* given in Figure 4 has time complexity $O(n^2)$ since an arbitrary maximal solution to *LP*-dual can be found by removing nodes from V until all nodes contribute to the weight of the cluster a value smaller than their penalty.

PD

input

1. A complete graph $G = (V, E)$.
2. A function $w : E \rightarrow \mathbb{N}$ [the length of the edge $\{i, j\}$]
3. A function $p : V \rightarrow \mathbb{N}$ [the penalty for not covering i]

output

A cluster C .

begin

Find a maximal solution to the dual problem.

$C := V \setminus \{i \in V \mid \sum_{e \in \delta(i)} \hat{R}_e = p_i\}$.

return C

end *PD*

Fig. 1. *PD*

Claim. Algorithm *PD* returns a 2-approximation to PMS.

Proof. Denote the value of the approximation by $apx = \sum_{e=\{i,j\} \in C} w_e + \sum_{i \in V \setminus C} p_i$, the value of the dual relaxation by $dual = \sum_{e \in E} R_e$, and the

optimal solution by *opt*. We now show that $apx \leq 2 \sum_{e \in E} R_e$. Consider an edge $e = \{i, j\} \in E$. If i and j are in $V \setminus C$, then $\sum_{e \in \delta(i)} R_e = p_i$, and $\sum_{e \in \delta(j)} R_e = p_j$, and hence R_e is charged in *apx* at most twice, once in p_i and once in p_j . If $i \in V \setminus C$ and $j \in C$ then R_e is charged in *apx* only once in p_i . If $i \in C$ and $j \in C$ then $R_e = w_e$ is charged in *apx* only once. We get: $apx \leq 2 \sum_{e \in E} R_e = 2dual \leq 2opt$. \square

5 Metric PMS

5.1 Hardness Proof

Claim. PMS is NP-hard even if w is a metric and $p_v = p$ for every $v \in V$.

The proof to claim 5.1 is by reduction from the l -clique problem.

5.2 Randomized PTAS for Uniform Penalties

In the following we assume the penalties are uniform, i.e. $p_v = p$ for every $v \in V$. We use the following notation: *opt* - the value of the optimal solution. *apx* - the value of the approximation. C^* - the cluster in the optimal solution and $P^* = V \setminus C^*$. For $A, B \subset V$: $w(A, B) = \sum_{a \in A} \sum_{b \in B} w(a, b)$, $w(u, B) = w(\{u\}, B)$, $\bar{w}(A, B) = \frac{w(A, B)}{|A||B|}$, $w(A) = \frac{1}{2}w(A, A)$, $\bar{w}(A) = \frac{w(A, A)}{|A|^2}$. $d_u = w(u, V)$, and $D_C = w(C, V)$.

Definition 1. [3] *A metric sample of U is a random sample $\{u_1, \dots, u_t\}$ of U with replacement, where each u_i is obtained by picking a point $u \in U$ with probability $\frac{d_u}{D_U}$.*

For $|C^*| \leq \frac{1}{\epsilon^2}$ the problem can be solved by exhaustive search in polynomial time, hence in the following we assume $|C^*| > \frac{1}{\epsilon^2}$.

An intuitive approach to the problem would be to take a random sample of C^* , use it to estimate the distance of each vertex from C^* , and then form a cluster from the vertices closest to the sample. This approach fails, in general, because the distance between the points entered to the cluster was not estimated and it is part of the weight of the cluster.

The algorithm presented below is based on two approaches to approximation schemes for partitioning problems. For the case where $|C^*| \leq \epsilon n$ we choose $C = \emptyset$. Since the penalty is uniform and $|P^*| \geq (1 - \epsilon)n$, this is a $(1 + 2\epsilon)$ -approximation.

For the case where $|P^*| \leq \epsilon n$ we use a method presented in [11,4] for the METRIC 2-MIN SUM PROBLEM. In general, the method in [4] is to find a representing vertex of the cluster, use it to estimate the distance between every vertex and the cluster, and add the close vertices to the cluster. This approach works because the number of vertices misplaced in the cluster is bounded by $|P^*| \leq \epsilon n$. This method is used in [4] for METRIC 2-MIN SUM PROBLEM for the case where one of the clusters is much greater than the other and the clusters

are well separated, that is, the maximum cut is much greater than the weight of the clusters. In *MSCP*, C^* and P^* are not necessarily well separated, but as will be proved bellow, an algorithm based on this method is a *PTAS* when $|P^*| \leq \epsilon n$. We denote this part of our algorithm by *UC*.

For the case where $|C^*| > \epsilon n$ and $|P^*| > \epsilon n$ we encounter the following difficulties: A large random sample of C^* is not enough to estimate, with sufficient accuracy, the distance $w(v, C^*)$. This problem was addressed and solved in [3] by taking a metric sample which enables to estimate $w(v, C^*)$ accurately. But good estimation of $w(v, C^*)$ is not enough. Consider the following instance of *MSCP*: $V = A \cup B$ where $|A| = |B| = n$, all distances in A and between A and B are 1, the distances between points of B are 2, and $p = n - 1$. The optimal solutions are $C^* = A$, and $C^* = (A \setminus \{v\}) \cup \{u\}$ for every $v \in A$ and $u \in B$. Note that for every sample T of A , $w(v, T) = w(u, T)$ where $v \in A$ and $u \in B$. Adding to the cluster points closer to the sample may lead to adding only points of B , resulting in a poor approximation. The distance **between** the points added to the sample should also be considered. We consider the distances between the added points using the hybrid partition method presented by [6] and used in [3]. The use in [3] is for the creation of a cut, whereas we create a cluster and hence the analysis is different.

Our algorithm for the case where $C^* > \epsilon n$ and $P^* > \epsilon n$ begins by taking a metric sample T of size $O(\frac{\ln(n)}{\epsilon^4})$ from V , and by exhaustive search find $T^* = C^* \cap T$ and use it to estimate $w(v, C^*)$ for every $v \in V$. Let \hat{e}_v denote the estimate of $w(v, C^*)$, and let C denote the cluster returned by the algorithm. We consider only the vertices with $\hat{e}_v \leq (1 + \epsilon)p$ as candidates for C . We partition these vertices into the following two sets, $C_{-\epsilon} = \{v \in V \mid \hat{e}_v \leq p(1 - \epsilon)\}$ and $C_{\pm\epsilon} = \{v \in V \mid p(1 - \epsilon) \leq \hat{e}_v \leq p(1 + \epsilon)\}$. We assume $C_{-\epsilon} \subset C^*$ and hence add it to C . We then use the hybrid partition method on the set $C_{\pm\epsilon}$, meaning that we randomly partition $C_{\pm\epsilon}$ to $r = \frac{1}{\epsilon}$ sets of equal size V_1, \dots, V_r . Assume we know $|V_j \cap C^*|$ for $j = 1, \dots, r$ (these are found by exhaustive search). The algorithm begins with $C = T^* \cup C_{-\epsilon}$, and then goes over $V_j, j = 1, \dots, r$ and adds to C the set C_j of the l_j vertices with smallest values of $\bar{c}(v) = \sum_{k < j} w(v, C_k) + \frac{r - (j - 1)}{r} (\hat{e}_v - w(v, C_{-\epsilon}))$ from V_j . This step considers the distances **between** part of the vertices added to C and ensures good approximation. We denote this part of the algorithm by *BC*. Our algorithm calls the two algorithms *UC* and *BC* presented below, and returns the best solution.

Let C denote the cluster returned by the algorithm. Let $P = V \setminus C$. We will analyze the following three cases separately:

- Case 1: $|C^*| < \epsilon n$.
- Case 2: $|P^*| < \epsilon n$.
- Case 3: $|C^*| \geq \epsilon n$ and $|P^*| \geq \epsilon n$.

In our analysis we assume that ϵ is sufficiently small (for example, $\epsilon < \frac{1}{8}$).

Theorem 1. *If $|C^*| < \epsilon n$ then $apx \leq (1 + 2\epsilon)opt$.*

Proof. In this case $opt \geq (n - |C^*|)p \geq n(1 - \epsilon)p$, and $apx \leq apx_{UC} \leq np$, which yields $\frac{apx}{opt} \leq \frac{1}{1 - \epsilon} \leq 1 + 2\epsilon$ for $\epsilon \leq \frac{1}{2}$. □

```

UC
begin
  C := ∅, apx_UC := np, l = |C*|. [l is found by exhaustive search.]
  for every v ∈ V. [v is considered the vertex defining the cluster]
    C := l nodes of V with the smallest values of l · w(u, v).
    apx := w(C) + (n - l)p.
    if apx < apx_UC
      then
        apx_UC := apx, UC_min := C.
      end if
    end for
  return UC_min, apx_UC
end UC

```

Fig. 2. UC

For Case 2, $|P^*| < \epsilon n$, we use the following lemma proved in [4] for the case where $w(C) + w(P) \leq \epsilon^2 w(V)$. Here we do not make this assumption. Note that $\bar{w}(C^*) = \frac{2w(C^*)}{|C^*|^2}$ and $\bar{w}(C^*)|C^*| = \frac{2w(C^*)}{|C^*|}$ is the average value of $w(v, C^*)$ for $v \in C^*$ and hence there is a vertex $z \in C^*$ for which $w(z, C^*) \leq \frac{2w(C^*)}{|C^*|}$.

Lemma 1. *Assume $|P^*| < \epsilon n$. Let $z \in C^*$ such that $w(z, C^*) \leq \frac{2w(C^*)}{|C^*|}$. Let C consist of the $|C^*|$ nodes in V with the smallest values of $w(v, z)$. Then:*

1. $w(C \setminus C^*, C^*) - w(C^* \setminus C, C^*) \leq \frac{4\epsilon}{1-\epsilon} w(C^*)$.
2. $w(C^* \setminus C) \leq \frac{\epsilon}{1-\epsilon} w(C^*)$.
3. $w(C \setminus C^*) \leq \frac{3\epsilon}{1-\epsilon} \left(\frac{4\epsilon}{1-\epsilon} + 3 \right) w(C^*)$.

We omit the proof of the lemma, but it relies on the following outcome of the triangle inequality, which is used throughout this extended abstract, for any sets $X, Y, Z \subset V$,

$$|Z|w(X, Y) \leq |X|w(Y, Z) + |Y|w(X, Z). \tag{1}$$

Theorem 2. *If $|P^*| < \epsilon n$ then $apx \leq (1 + 32\epsilon)opt$.*

Proof. We may assume that the vertex $z \in C$ that defined the cluster satisfies $z \in C^*$, $w(z, C^*) \leq \frac{2w(C^*)}{|C^*|}$ and that $|C| = |C^*|$. If this is not the case then the bound may only improve. Under these assumptions we may use the bounds presented by Lemma 1 to bound the approximation ratio. Let $X = C^* \setminus C$ and $Y = C \setminus C^*$, then

$$\begin{aligned}
 apx - opt &= [w(Y, C^* \cap C) - w(X, C^* \cap C)] + [w(Y) - w(X)] \\
 &= [w(Y, C^*) - w(Y, X)] - [w(X, C^*) - w(X, X)] + [w(Y) - w(X)] \\
 &\leq w(Y, C^*) - w(X, C^*) + w(X, X) + w(Y) - w(Y, X) - w(X) \\
 &\leq \frac{4\epsilon}{1-\epsilon} w(C^*) + \frac{\epsilon}{1-\epsilon} w(C^*) + \frac{3\epsilon}{1-\epsilon} \left(\frac{4\epsilon}{1-\epsilon} + 3 \right) w(C^*) - w(Y, X) \\
 &\leq (1 + 32\epsilon)w(C^*) \leq (1 + 32\epsilon)opt.
 \end{aligned}$$

BC

begin

$$\hat{D}_{C^*} := \{(1 + \epsilon)^j \mid (1 + \epsilon)^j \leq D_{C^*} < (1 + \epsilon)^{j+1}\}.$$

[we do not know \hat{D}_{C^*} but we find it by exhaustive search.]

$$r := \frac{1}{\epsilon}. \text{ [w.l.o.g } r \text{ is an integer.]}$$

Take a metric sample T of V , $|T| = \frac{8 \ln(4n)}{\epsilon^4}$.

$l = |C^*|$. [l is found by exhaustive search.]

$T^* := C^* \cap T$. [T^* is found by exhaustive search.]

$$\forall v \in V, \hat{e}_v := \frac{\hat{D}_{C^*}}{|T^*|} \sum_{u \in T^*} \frac{w(v, u)}{d_u}.$$

$$C_{-\epsilon} := \{v \in V \mid \hat{e}_v \leq p(1 - \epsilon)\}.$$

if $|C_{-\epsilon}| \geq l$

then Add to BC_{min} l vertices of $C_{-\epsilon}$ with the smallest value of \hat{e}_v .

return BC_{min} and apx_BC .

end if

$$C_0 := T^* \cup C_{-\epsilon}, P_0 := T \setminus T^*.$$

$$C_{\pm\epsilon} := \{v \in V \mid p(1 - \epsilon) \leq \hat{e}_v \leq p(1 + \epsilon)\}.$$

Partition $C_{\pm\epsilon}$ randomly into r sets V_1, \dots, V_r of equal size (as possible).

Let $l_j := |V_j \cap C^*|$, $j = 1, \dots, r$. [l_1, \dots, l_r are found by exhaustive search.]

for $j = 1, \dots, r$

$$\forall v \in V_j, \bar{c}(v) := \sum_{k < j} w(v, C_k) + \frac{r - (j - 1)}{r} (\hat{e}_v - w(v, C_{-\epsilon})).$$

$C_j := l_j$ vertices $v \in V_j$ with smallest values of $\bar{c}(v)$.

end for

$$BC_{min} = \cup_j C_j, apx_BC = w(BC_{min}) + (n - l)p.$$

return BC_{min}, apx_BC

end BC

Fig. 3. BC

The last inequality holds for $\epsilon < \frac{1}{8}$. □

For Case 3, $|C^*| \geq \epsilon n$ and $|P^*| \geq \epsilon n$, we use the following lemma:

Lemma 2. [3] Let t be given and $U \subset V$. Let $T = \{u_1, \dots, u_t\}$ be a metric sample of U of size t . Consider a fixed vertex $v \in V$,

$$\Pr[|w(v, U) - e_v| \leq \epsilon w(v, U)] \geq 1 - 2e^{-t\epsilon^2/8}$$

and moreover,

$$E[|w(v, U) - e_v|] \leq \frac{2}{\sqrt{t}} w(v, U),$$

$$\text{where } e_v = \frac{D_U}{t} \sum_{u \in T} \frac{w(v, u)}{d_u}.$$

Remark 1. For simplicity we assume in the following that $D_{C^*} = \hat{D}_{C^*}$, implying $e_v = \hat{e}_v$ for every $v \in V$. Since $\frac{D_{C^*}}{\hat{D}_{C^*}} < 1 + \epsilon$, the real value of the solution is at most $1 + \epsilon$ times the value of the solution under this assumption.

Lemma 3. *Let T be a metric sample of V where $|T| \geq \frac{1}{\epsilon^4}$, and let $C \subset V$ where $|C| \geq \epsilon n$. Then $T \cap C$ is a metric sample of C , and $Pr[|T \cap C| \geq \epsilon^2 |T|] \geq 1 - \epsilon$.*

Proof. Clearly $C \cap T$ is a metric sample of C . It is sufficient to prove the bound on $|T \cap C|$ for the boundary case $|C| = \epsilon n$ and $|P| = |V \setminus C| = (1 - \epsilon)n$. By (1) with $X = Y = P$ and $Z = C$ $\bar{w}(P) \leq 2\bar{w}(C, P)$. Therefore,

$$\begin{aligned} D_P &= (1 - \epsilon)n [\epsilon n \bar{w}(C, P) + ((1 - \epsilon)n - 1)\bar{w}(P)] \\ &\leq (1 - \epsilon)n^2 [\epsilon + 2(1 - \epsilon)] \bar{w}(C, P) \\ &= [\epsilon + 2(1 - \epsilon)] K. \end{aligned}$$

Also, $D_C \geq (1 - \epsilon)\epsilon n^2 \bar{w}(C, P) = \epsilon K$, and by the metric sample definition,

$$Pr[u_i \in C / u_i \in T] \geq \frac{D_C}{D_C + D_P} \geq \frac{\epsilon}{\epsilon + \epsilon + 2(1 - \epsilon)} = \frac{\epsilon}{2}.$$

The number of vertices from C in T stochastically dominates the binomial random variable $X \sim B(|T|, \frac{\epsilon}{2})$ and by the Central Limit Theorem, for $|T| \geq \frac{1}{\epsilon^4}$ and $\epsilon \leq 0.2$, $Pr[X \geq 2\epsilon E[X]] \geq 1 - \epsilon$. □

Let $C_{+\epsilon} := \{v \in V \mid e_v \leq p(1 + \epsilon)\}$,

Lemma 4. *Let $|T| = \frac{8 \ln(4n)}{\epsilon^4}$.*

1. $Pr \left[\left(|T \cap C^*| \geq \frac{8 \ln(4n)}{\epsilon^2} \right) \wedge (w(v, C^*) \leq 2p \ \forall v \in C_{+\epsilon}) \right] \geq \frac{3}{4}(1 - \epsilon)$.
2. $E[|P^* \cap C_{-\epsilon}|] \leq 1$ and $E[w(C, P^* \cap C_{-\epsilon})] \leq 4\epsilon opt$.
3. $E[|C^* \setminus C_{+\epsilon}|] \leq 1$ and $E[w(C^* \setminus C_{+\epsilon}, C^*)] \leq \epsilon opt$.

In the following we assume that $C_{-\epsilon} \subseteq C^*$ and that $C^* \subseteq C_{+\epsilon}$. It follows from the third part of Lemma 4 that with probability $\frac{3}{4}(1 - \epsilon)$ the expected weight of the errors due to this assumption are $O(\epsilon opt)$.

The following lemma is based on the deterministic analysis in [3]. For $j = 1, \dots, r$, let $C_j^* = C^* \cap V_j$, and let I_j denote the following *left part* of the hybrid partitioning:

$$I_j = \left(\bigcup_{k \leq j} C_k \right) \cup \left(\bigcup_{k > j} C_k^* \right).$$

Under the assumptions $C_{-\epsilon} \subset C^*$ and $C^* \in C_{+\epsilon}$, $I_0 = C^*$ and $I_r = C$. For $j = 1, \dots, r$, consider the points that are classified differently in I_j and I_{j-1} . Let $X_j := C_j^* \setminus C_j = \{x_1, \dots, x_m\}$ and $Y_j := C_j \setminus C_j^* = \{y_1, \dots, y_m\}$.

Lemma 5.

$$\begin{aligned} E[w(I_j) - w(I_{j-1})] &\leq + \sum_{u \in X_j \cup Y_j} E \left[\left| w(u, \bigcup_{k \geq j-1} C_k^*) - \frac{r-j+1}{r} w(u, C^* \cap C_{\pm\epsilon}) \right| \right. \\ &\quad \left. + \frac{r-j+1}{r} |w(u, C^*) - e_u| \right] + E[w(Y_j, X_j)]. \end{aligned}$$

Proof.

$$\begin{aligned}
 w(I_j) - w(I_{j-1}) &= w(Y_j, I_{j-1} \setminus X_j) + w(Y_j) - [w(X_j, I_{j-1} \setminus X_j) + w(X_j)] \\
 &= w(Y_j, I_{j-1}) - w(X_j, I_{j-1}) + w(Y_j) + w(X_j) - w(Y_j, X_j) \\
 &\leq \sum_{i=1}^{|Y_j|} [w(y_i, I_{j-1}) - w(x_i, I_{j-1})] + w(Y_j, X_j) \\
 &\leq \sum_{i=1}^{|Y_j|} [w(y_i, I_{j-1}) - \bar{c}(y_i) + \bar{c}(x_i) - w(x_i, I_{j-1})] + w(Y_j, X_j) \\
 &\leq \sum_{u \in X_j \cup Y_j} |w(u, I_{j-1}) - \bar{c}(u)| + w(Y_j, X_j) \tag{2}
 \end{aligned}$$

The first inequality is due to (1) with $X = Y = Y_j$, $Z = X_j$ and $|X_j| = |Y_j|$, giving $w(Y_j) \leq w(X_j, Y_j)$. Similarly, $w(X_j) \leq w(X_j, Y_j)$. The second inequality holds since $\bar{c}(y_i) \leq \bar{c}(x_i)$.

Under the assumption $\hat{e}_v = e_v$, $\bar{c}(u) = \sum_{k < j} w(u, C_k) + \frac{r-(j-1)}{r}(e_u - w(u, C_{-\epsilon}))$,

$$\begin{aligned}
 |w(u, I_{j-1}) - \bar{c}(u)| &= \left| w(u, \bigcup_{k \geq j-1} C_k^*) - \frac{r-j+1}{r} [e_u - w(u, C_{-\epsilon})] \right| \\
 &= \left| w(u, \bigcup_{k \geq j-1} C_k^*) - \frac{r-j+1}{r} [e_u + w(u, C^* \cap C_{\pm\epsilon}) - w(u, C^*)] \right| \\
 &\leq \left| w(u, \bigcup_{k \geq j-1} C_k^*) - \frac{r-j+1}{r} w(u, C^* \cap C_{\pm\epsilon}) \right| \\
 &\quad + \frac{r-j+1}{r} |w(u, C^*) - e_u|. \tag{3}
 \end{aligned}$$

Substituting (3) into (2) and drawing expectation on both sides completes the proof of the lemma. \square

Theorem 3. *If $|C^*| \geq \epsilon n$ and $|P^*| \geq \epsilon n$, then with probability of at least $1 - \epsilon$, $apx \leq (1 + 52\epsilon)opt$.*

Remark 2. *It is sufficient to show that $apx \leq (1 + 52\epsilon)opt$ with any constant probability, and obtaining an $1 - \epsilon$ probability by running the algorithm a polynomial number of times and choosing the best solution.*

Proof. In the following we assume for every $v \in C_{+\epsilon}$

$$w(v, C^*) \leq 2p, \tag{4}$$

and $|T \cap C^*| \geq \frac{8 \ln(4n)}{\epsilon^2}$. By the first part of Lemma 4, these assumptions hold with probability $\frac{3}{4}(1 - \epsilon)$.

Fix $u \in X_j \cup Y_j$ and let $Z_u = w(u, \bigcup_{k \geq j-1} C_k^*) = \sum_{k \geq j-1} w(u, C_k^*)$. Since $C_{\pm\epsilon}$ is partitioned randomly into V_1, \dots, V_r , $E[Z_u] = \frac{r-j+1}{r} w(u, C^* \cap C_{\pm\epsilon})$ and $Z_u = \sum_{s \in C^* \cap C_{\pm\epsilon}} w(u, s) A_s$, where $\{A_s\}$ i.i.d.r.v's with $Pr[A_s = 1] = \frac{r-j+1}{r}$ and zero with the complementary probability.

We use (1) with $X = \{u\}$, $Y = \{s\}$ and $Z = C^*$, (4), and the fact that $w(s, C^*) \leq p$ for $s \in C^*$, to obtain for every $u \in C_{\pm\epsilon}$,

$$w(u, s) \leq \frac{w(u, C^*) + w(s, C^*)}{|C^*|} \leq \frac{3p}{|C^*|}. \tag{5}$$

We use Hoeffding’s inequality for $Z_u = \sum_{s \in C^* \cap C_{\pm\epsilon}} w(u, s)A_s := \sum_{s \in C^* \cap C_{\pm\epsilon}} Q_s$, where Q_s , for every $s \in C^* \cap C_{\pm\epsilon}$, is nonnegative and bounded by $b_s = \frac{3p}{|C^*|}$, to obtain

$$E[|Z_u - E[Z_u]|] \leq \sqrt{\frac{\pi \sum_{s \in C^* \cap C_{\pm\epsilon}} 9p^2}{2|C^*|^2}} \leq \frac{4p}{|C^*|^{\frac{1}{2}}},$$

and therefore, for every $u \in X_j \cup Y_j$,

$$\begin{aligned} E\left[\left| w(u, \bigcup_{k \geq j-1} C_k^*) - \frac{r-j+1}{r} w(u, C^* \cap C_{\pm\epsilon}) \right| \right] &= E[|Z_u - E[Z_u]|] \\ &\leq \frac{4p}{|C^*|^{\frac{1}{2}}}. \end{aligned} \tag{6}$$

By the second part of Lemma 2, 4 and the assumption $|T \cap C^*| \geq \frac{8 \ln(4n)}{\epsilon^2}$, for every $u \in X_j \cup Y_j$,

$$E[|w(u, C^*) - e_u|] \leq \frac{2w(u, C^*)}{\sqrt{|T \cap C^*|}} \leq \frac{\epsilon w(u, C^*)}{\sqrt{2 \ln(4n)}} \leq \frac{\epsilon p}{\sqrt{\ln(4n)}}. \tag{7}$$

Substituting (6) and (7) in (3), we get for every $u \in X_j \cup Y_j$

$$\begin{aligned} |w(u, I_{j-1}) - \bar{c}(u)| &\leq \frac{4p}{|C^*|^{\frac{1}{2}}} + \frac{r-j+1}{r} \left(\frac{\epsilon p}{\sqrt{\ln(4n)}} \right) \\ &\leq \frac{4p}{|C^*|^{\frac{1}{2}}} + \frac{\epsilon p}{\sqrt{\ln(4n)}}. \end{aligned} \tag{8}$$

Since V_j are determined randomly, in expectation $l_1 = \dots = l_r = \epsilon |C^* \cap C_{\pm\epsilon}|$. Therefore $E[|Y_j|] = E[|X_j|] \leq \epsilon \min\{|C^* \cap C_{\pm\epsilon}|, |P^*|\}$. Substituting (8) into Lemma 5,

$$\begin{aligned} E[w(I_j) - w(I_{j-1})] &\leq E \left[w(Y_j, X_j) + \sum_{u \in X_j \cup Y_j} \left(\frac{4p}{|C^*|^{\frac{1}{2}}} + \frac{\epsilon p}{\sqrt{\ln(4n)}} \right) \right] \\ &= E(w(Y_j, X_j)) + 2E|Y_j| \left[\frac{4p}{|C^*|^{\frac{1}{2}}} + \frac{\epsilon p}{\sqrt{\ln(4n)}} \right] \\ &\leq E(w(Y_j, X_j)) + 10\epsilon^2 |P^*| p. \end{aligned} \tag{9}$$

The second inequality holds since $|C^*| \geq \frac{1}{\epsilon^2}$ and $|P^*| \geq \frac{1}{\epsilon^2}$ (else the optimal solution can be found by exhaustive search), and $E[|Y_j|] \leq \epsilon |P^*|$. Let $x \in X_j$ and $y \in Y_j$.

Summing (5) over all $x \in X_j$ and $y \in Y_j$, and since $E[|Y_j|] = E[|X_j|] \leq \epsilon \min\{|C^* \cap C_{\pm\epsilon}|, |P^*|\}$ we get

$$E[w(X_j, Y_j)] \leq \frac{E[|Y_j|]E[|X_j|]3p}{|C^*|} \leq 3\epsilon^2|P^*|p \leq 3\epsilon^2 \text{opt}. \quad (10)$$

Substituting (10) into (9) and noting that $\text{opt} \geq |P^*|p$,

$$E[w(I_j) - w(I_{j-1})] \leq 13\epsilon^2 \text{opt}. \quad (11)$$

Summing (11) over $j = 1, \dots, r = \frac{1}{\epsilon}$ gives,

$$E[w(C) - w(C^*)] = E[w(I_r) - w(I_0)] \leq 13\epsilon \text{opt}. \quad (12)$$

Using Markov's inequality for a constant $K \geq 52$ completes the proof of the theorem: With probability $\frac{3}{4}(1 - \epsilon)$,

$$Pr[w(C) - w(C^*) \leq K\epsilon \text{opt}] \geq 1 - \frac{E[w(C) - w(C^*)]}{K\epsilon \text{opt}} \geq 1 - \frac{13}{K} \geq \frac{3}{4}. \quad \square$$

Corollary 1. *With probability of at least $1 - \epsilon$ the algorithm returns an $(1 + 52\epsilon)$ approximation to MSCP in $O(n^{\frac{1}{\epsilon} + 2} \ln(n))$ time.*

5.3 Non Uniform Penalties

The algorithm can be generalized to the case where the ratio between the maximal penalty and the minimal penalty is bounded, by exhaustively searching for the number of nodes from each cost, in an optimal cluster. An approximation ratio smaller than 2 for the case of general penalties remains open.

5.4 2-Approximation for Metric k PMS

In this section we generalize [7], which offers a 2-approximation for the metric K-MIN-SUM problem for a fixed k . First we define THE MINIMUM STAR PARTITION PROBLEM (SPP). Given a graph $G = (V, E)$ (SPP) requires to find $k + 1$ distinct nodes $\{v_i\}_{i=1}^{k+1} \in V$ and a partition of V into $K + 1$ disjoint sets $\{S_1, \dots, S_{k+1}\}$ such that $\sum_{u=1}^K l_i w(v_i, S_i) + \sum_{v \in S_{k+1}} w(z, v)$ is minimized. SPP can be solved in polynomial time by solving a transportation problem presented in [7]. We suggest the following algorithm:

- Create a new graph $G' = (V', E')$ by adding a node z to V . $V' = V \cup \{z\}$, $E' = E \cup \{(v, z) \mid v \in V\}$, and $w(v, z) = p_v$ for every $v \in V$.
- Denote by S^* the value of the optimal solution to the SPP problem on G' and let $\{S_1^*, \dots, S_{k+1}^*\}$ be its optimal partitioning.
- Find the sets $\{S_1^*, \dots, S_{k+1}^*\}$ using the method presented in [7], and return the the optimal partition $\{S_1^*, \dots, S_{k+1}^*\}$.

Let $apx = \sum_{u=1}^k w(S_u^*) + \sum_{v \in S_{k+1}^*} p_v$. Let O_1, \dots, O_{k+1} denote an optimal partition for the k PMS problem, and denote its value by opt .

Claim. $apx \leq 2opt$.

Proof. Let $w(x_i, O_i) = \min_{t \in O_i} w(t, O_i)$. Then, $2opt \geq \sum_{u=1}^k \sum_{t \in O_i} l_i w(t, x_i) + \sum_{j \in O_{k+1}} p_j \geq \sum_{u=1}^K l_i w(v_i, S_i^*) + \sum_{v \in S_{K+1}^*} w(z, v) = S^*$. By the triangle inequality $S^* \geq apx$ and we conclude $2opt \geq apx$. \square

References

1. M. Charikar, S. Khuller, D.M. Mount, and G. Narasimhan, "Algorithms for facility location problems with outliers", *SODA* (2001), 642-651.
2. W. Fernandez de la Vega, and C. Kenyon, "A randomized approximation scheme for metric MAX-CUT", *J. Comput. Science* 63 (2001), 531-541.
3. W. Fernandez de la Vega, M. Karpinski and C. Kenyon, "Approximation schemes for metric bisection and partitioning", *SODA* (2004).
4. W. Fernandez de la Vega, M. Karpinski, C. Kenyon and Y. Rabani, "Approximation schemes for clustering problems", *Proc. 35th ACM STOC* (2003).
5. U. Feige, G. Kortsarz and D. Peleg, "The dense k-subgraph problem", *Algorithmica*, (2001), 410-421.
6. O. Goldreich, S. Goldwasser and D. Ron, "Property testing and its connection to learning and approximation", *Proc. 37th IEEE FOCS* (1996), 339-348.
7. Guttman-Beck, N. and R. Hassin, "Approximation algorithms for min-sum p-clustering", *Discrete Applied Mathematics* **89** (1998), 125-142.
8. M.R. Garey and D.S. Johnson "Computers and Intractability", *Freeman* (1979).
9. Hochbaum, D.S., "Solving integer programs over monotone inequalities in three variables: a framework for half integrality and good approximation", *European Journal of Operational Research* **140** (2002), 291-321.
10. R. Hassin, S. Rubinstein and A. Tamir, "Approximation algorithm for maximum dispersion", *Operations research letters* **21**, (1997), 133-137.
11. Indyk, P., "A sublinear time approximation scheme for clustering in metric spaces", *40th Symposium on Foundations of Computer Science*, 1999, 154-159.
12. G. Xu, J. Xu, "An LP rounding algorithm for approximating uncapacitated facility location problem with penalties [rapid communication]", *Information Processing Letters*, **94**, 3 (2005), 119-123.

Improved Approximation Algorithms for Metric Max TSP*

Zhi-Zhong Chen** and Takayuki Nagoya

Dept. of Math. Sci., Tokyo Denki Univ., Hatoyama, Saitama 350-0394, Japan
{chen, nagoya}@r.dendai.ac.jp

Abstract. We present two polynomial-time approximation algorithms for the metric case of the maximum traveling salesman problem. One of them is for directed graphs and its approximation ratio is $\frac{27}{35}$. The other is for undirected graphs and its approximation ratio is $\frac{7}{8} - o(1)$. Both algorithms improve on the previous bests.

1 Introduction

The *maximum traveling salesman problem* (MaxTSP) is to compute a maximum-weight Hamiltonian circuit (called a *tour*) in a given complete edge-weighted (undirected or directed) graph. Usually, MaxTSP is divided into the *symmetric* and the *asymmetric* cases. In the symmetric case, the input graph is undirected; we denote this case by SymMaxTSP. In the asymmetric case, the input graph is directed; we denote this case by AsymMaxTSP. Note that SymMaxTSP can be trivially reduced to AsymMaxTSP.

A natural constraint one can put on AsymMaxTSP and SymMaxTSP is the *triangle inequality* which requires that for every set of three vertices u_1 , u_2 , and u_3 in the input graph G , $w(u_1, u_2) \leq w(u_1, u_3) + w(u_3, u_2)$, where $w(u_i, u_j)$ is the weight of the edge from u_i to u_j in G . If we put this constraint on AsymMaxTSP, we obtain a problem called *metric* AsymMaxTSP. Similarly, if we put this constraint on SymMaxTSP, we obtain a problem called *metric* SymMaxTSP.

Both metric SymMaxTSP and metric AsymMaxTSP are Max-SNP-hard [1] and there have been a number of approximation algorithms known for them [5,3,4]. In 1985, Kostochka and Serdyukov [5] gave an $O(n^3)$ -time approximation algorithm for metric SymMaxTSP that achieves an approximation ratio of $\frac{5}{6}$. Their algorithm is very simple and elegant. Tempted by improving the ratio $\frac{5}{6}$, Hassin and Rubinfeld [3] gave a randomized $O(n^3)$ -time approximation algorithm for metric SymMaxTSP whose *expected* approximation ratio is $\frac{7}{8} - o(1)$. This randomized algorithm was recently (partially) derandomized by Chen *et al.* [2]; their result is a (deterministic) $O(n^3)$ -time approximation algorithm for metric SymMaxTSP whose approximation ratio is $\frac{17}{20} - o(1)$. In this paper, we completely derandomize the randomized algorithm, i.e., we obtain

* The full version can be found at <http://rnc.r.dendai.ac.jp/~chen/papers/metric.pdf>.

** Supported in part by the Grant-in-Aid for Scientific Research of the Ministry of Education of Japan.

a (deterministic) $O(n^3)$ -time approximation algorithm for metric SymMaxTSP whose approximation ratio is $\frac{7}{8} - o(1)$. Our algorithm also has the advantage of being easy to parallelize. Our derandomization is based on the idea of Chen *et al.* [2] and newly discovered properties of a folklore partition of the edges of a $2n$ -vertex complete undirected graph into $2n - 1$ perfect matchings. These properties may be useful elsewhere. In particular, one of the properties says that if $G = (V, E)$ is a $2n$ -vertex complete undirected graph and M is a perfect matching of G , then we can partition $E - M$ into $2n - 2$ perfect matchings M_1, \dots, M_{2n-2} among which there are at most $k^2 - k$ perfect matchings M_i such that the graph $(V, M \cup M_i)$ has a cycle of length at most $2k$ for every natural number k . This property is interesting because Hassin and Rubinfeld [3] prove that if G and M are as before and M' is a random perfect matching of G , then with probability $1 - o(1)$ the multigraph $(V, M \cup M')$ has no cycle of length at most \sqrt{n} . Our result shows that instead of sampling from the set of all perfect matchings of G , it suffices to sample from M_1, \dots, M_{2n-2} . This enables us to completely derandomize their algorithm.

As for metric AsymMaxTSP, Kostochka and Serdyukov [5] gave an $O(n^3)$ -time approximation algorithm that achieves an approximation ratio of $\frac{3}{4}$. Their result remained the best in two decades until Kaplan *et al.* [4] gave a polynomial-time approximation algorithm whose approximation ratio is $\frac{10}{13}$. The key in their algorithm is a polynomial-time algorithm for computing two cycle covers \mathcal{C}_1 and \mathcal{C}_2 in the input graph G such that \mathcal{C}_1 and \mathcal{C}_2 do not share a 2-cycle and the sum of their weights is at least twice the optimal weight of a tour of G . They then observe that the multigraph formed by the edges in 2-cycles in \mathcal{C}_1 and \mathcal{C}_2 can be split into two subtours of G . In this paper, we show that the multigraph formed by the edges in 2-cycles in \mathcal{C}_1 and \mathcal{C}_2 *together with* a constant fraction of the edges in non-2-cycles in \mathcal{C}_1 and \mathcal{C}_2 can be split into two subtours of G . This enables us to improve Kaplan *et al.*'s algorithm to a polynomial-time approximation algorithm whose approximation ratio is $\frac{27}{35}$.

2 Basic Definitions

Throughout this paper, a *graph* means a simple undirected or directed graph (i.e., it has neither multiple edges nor self-loops), while a multigraph may have multiple edges but no self-loops.

Let G be a multigraph. We denote the vertex set of G by $V(G)$, and denote the edge set of G by $E(G)$. For a subset F of $E(G)$, $G - F$ denotes the graph obtained from G by deleting the edges in F . Two edges of G are *adjacent* if they share an endpoint.

Suppose G is undirected. The *degree* of a vertex v in G is the number of edges incident to v in G . A *cycle* in G is a connected subgraph of G in which each vertex is of degree 2. A *cycle cover* of G is a subgraph H of G with $V(H) = V(G)$ in which each vertex is of degree 2. A *matching* of G is a (possibly empty) set of pairwise nonadjacent edges of G . A *perfect matching* of G is a matching M of G such that each vertex of G is an endpoint of an edge in M .

Suppose G is directed. The *indegree* of a vertex v in G is the number of edges entering v in G , and the *outdegree* of v in G is the number of edges leaving v in G . A *cycle* in G is a connected subgraph of G in which each vertex has indegree 1 and outdegree 1. A *cycle cover* of G is a subgraph H of G with $V(H) = V(G)$ in which each vertex has indegree 1 and outdegree 1. A *2-path-coloring* of G is a partition of $E(G)$ into two subsets E_1 and E_2 such that both graphs $(V(G), E_1)$ and $(V(G), E_2)$ are collections of vertex-disjoint paths. G is *2-path-colorable* if it has a 2-path-coloring.

Suppose G is undirected or directed. A *path* in G is either a single vertex of G or a subgraph of G that can be transformed to a cycle by adding a single (new) edge. The *length* of a cycle or path C is the number of edges in C . A *k-cycle* is a cycle of length k . A 3^+ -*cycle* is a cycle of length at least 3. A *tour* (also called a *Hamiltonian cycle*) of G is a cycle C of G with $V(C) = V(G)$. A *subtour* of G is a subgraph H of G which is a collection of vertex-disjoint paths.

A *closed chain* is a directed graph that can be obtained from an undirected k -cycle C with $k \geq 3$ by replacing each edge $\{u, v\}$ of C with the two directed edges (u, v) and (v, u) . Similarly, an *open chain* is a directed graph that can be obtained from an undirected path P by replacing each edge $\{u, v\}$ of P with the two directed edges (u, v) and (v, u) . An open chain is *trivial* if it is a single vertex. A *chain* is a closed or open chain. A *partial chain* is a subgraph of a chain.

For a graph G and a weighting function w mapping each edge e of G to a nonnegative real number $w(e)$, the *weight* of a subset F of $E(G)$ is $w(F) = \sum_{e \in F} w(e)$, and the *weight* of a subgraph H of G is $w(H) = w(E(H))$.

3 New Algorithm for Metric AsymMaxTSP

Throughout this section, fix an instance (G, w) of metric AsymMaxTSP, where G is a complete directed graph and w is a function mapping each edge e of G to a nonnegative real number $w(e)$.

Let OPT be the weight of a maximum-weight tour in G . Our goal is to compute a tour in G whose weight is large compared to OPT . We first review Kaplan *et al.*'s algorithm and define several notations on the way.

3.1 Kaplan et al.'s Algorithm

The key in their algorithm is the following:

Theorem 1. [4] *We can compute two cycle covers $\mathcal{C}_1, \mathcal{C}_2$ in G in polynomial time that satisfy the following two conditions:*

1. \mathcal{C}_1 and \mathcal{C}_2 do not share a 2-cycle. In other words, if C is a 2-cycle in \mathcal{C}_1 (respectively, \mathcal{C}_2), then \mathcal{C}_2 (respectively, \mathcal{C}_1) does not contain at least one edge of C .
2. $w(\mathcal{C}_1) + w(\mathcal{C}_2) \geq 2 \cdot OPT$.

Let G_2 be the subgraph of G such that $V(G_2) = V(G)$ and $E(G_2)$ consists of all edges in 2-cycles in \mathcal{C}_1 and/or \mathcal{C}_2 . Then, G_2 is a collection of vertex-disjoint chains. For each closed chain C in G_2 , we can compute two edge-disjoint tours T_1 and T_2 (each of which is of length at least 3), modify \mathcal{C}_1 by substituting T_1 for the 2-cycles shared by C and \mathcal{C}_1 , modify \mathcal{C}_2 by substituting T_2 for the 2-cycles shared by C and \mathcal{C}_2 , and further delete C from G_2 . After this modification of \mathcal{C}_1 and \mathcal{C}_2 , the two conditions in Theorem 1 still hold. So, we can assume that there is no closed chain in G_2 .

For each $i \in \{1, 2\}$, let $W_{i,2}$ denote the total weight of 2-cycles in \mathcal{C}_i , and let $W_{i,3} = w(\mathcal{C}_i) - W_{i,2}$. For convenience, let $W_2 = \frac{1}{2}(W_{1,2} + W_{2,2})$ and $W_3 = \frac{1}{2}(W_{1,3} + W_{2,3})$. Then, by Condition 2 in Theorem 1, we have $W_2 + W_3 \geq OPT$. Moreover, using an idea in [5], Kaplan *et al.* observed the following:

Lemma 1. [4] *We can use \mathcal{C}_1 and \mathcal{C}_2 to compute a tour T of G with $w(T) \geq \frac{3}{4}W_2 + \frac{5}{6}W_3$ in polynomial time.*

Since each nontrivial open chain has a 2-path-coloring, we can use G_2 to compute a tour T' of G with $w(T') \geq W_2$ in polynomial time. Combining this observation, Lemma 1, and the fact that $W_2 + W_3 \geq OPT$, the heavier one between T and T' is of weight at least $\frac{10}{13}OPT$.

3.2 Details of the New Algorithm

The idea behind our new algorithm is to improve the second tour T' in Kaplan *et al.*'s algorithm so that it has weight at least $W_2 + \frac{1}{9}W_3$. The tactics is to add some edges of 3^+ -cycles in \mathcal{C}_i with $W_{i,3} = \max\{W_{1,3}, W_{2,3}\}$ to G_2 so that G_2 remains 2-path-colorable. Without loss of generality, we may assume that $W_{1,3} \geq W_{2,3}$. Then, our goal is to add some edges of 3^+ -cycles in \mathcal{C}_1 to G_2 so that G_2 remains 2-path-colorable.

We say that an open chain P in G_2 *spoils* an edge (u, v) of a 3^+ -cycle in \mathcal{C}_1 if u and v are the two endpoints of P . Obviously, adding a spoiled edge to G_2 destroys the 2-path-colorability of G_2 . Fortunately, there is no 3^+ -cycle in \mathcal{C}_1 in which two consecutive edges are both spoiled. So, let $\mathcal{C}_1, \dots, \mathcal{C}_\ell$ be the 3^+ -cycles in \mathcal{C}_1 ; we modify each C_j ($1 \leq j \leq \ell$) as follows (see Figure 1):

- For every two consecutive edges (u, v) and (v, x) of C_j such that (u, v) is spoiled, replace (u, v) by the two edges (u, x) and (x, v) . (*Comment:* We call (u, x) a *bypass edge* of C_j , call the 2-cycle between v and x a *dangling 2-cycle* of C_j , and call v the *articulation vertex* of the dangling 2-cycle. We also say that the bypass edge (u, x) and the dangling 2-cycle between v and x *correspond* to each other.)

We call the above modification of C_j the *bypass operation* on C_j . Note that applying the bypass operation on C_j does not decrease the weight of C_j because of the triangle inequality. Moreover, the edges of C_j not contained in dangling 2-cycles of C_j form a cycle. We call it the *primary cycle* of C_j . Note that C_j may have neither bypass edges nor dangling 2-cycles (this happens when C_j has no spoiled edges).

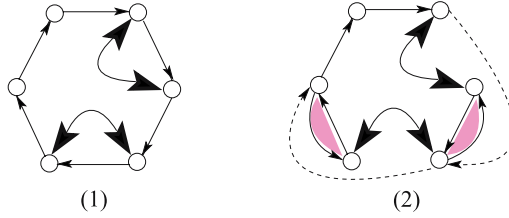


Fig. 1. (1) A 3^+ -cycle C_j (formed by the one-way edges) in C_1 and the open chains (each shown by a two-way edge) each of which has a parallel edge in C_j . (2) The modified C_j (formed by the one-way edges), where bypass edges are dashed and dangling 2-cycles are painted.

Let H be the union of the modified C_1, \dots, C_ℓ , i.e., let H be the directed graph with $V(H) = \bigcup_{1 \leq j \leq \ell} V(C_j)$ and $E(H) = \bigcup_{1 \leq j \leq \ell} E(C_j)$. We next show that $E(H)$ can be partitioned into three subsets each of which can be added to G_2 without destroying its 2-path-colorability. Before proceeding to the details of the partitioning, we need several definitions and lemmas.

Two edges (u_1, u_2) and (v_1, v_2) of H form a *critical pair* if u_1 and v_2 are the endpoints of some open chain in G_2 and u_2 and v_1 are the endpoints of another open chain in G_2 (see Figure 2). Note that adding both (u_1, u_2) and (v_1, v_2) to G_2 destroys its 2-path-colorability. An edge of H is *critical* if it together with another edge of H forms a critical pair. Note that for each critical edge e of H , there is a unique edge e' in H such that e and e' form a critical pair. We call e' the *rival* of e . An edge of H is *safe* if it is not critical. A *bypass edge* of H is a bypass edge of a C_j with $1 \leq j \leq \ell$. Similarly, a *dangling 2-cycle* of H is a dangling 2-cycle of a C_j with $1 \leq j \leq \ell$. A *dangling edge* of H is an edge in a dangling 2-cycle of H .

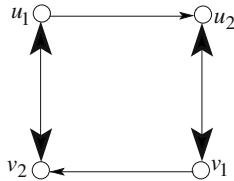


Fig. 2. A critical pair formed by edges (u_1, u_2) and (v_1, v_2)

Lemma 2. *No bypass edge of H is critical.*

Lemma 3. *Fix a j with $1 \leq j \leq \ell$. Suppose that an edge e of C_j is a critical dangling edge of H . Let C be the dangling 2-cycle of C_j containing e . Let e' be the rival of e . Then, the following statements hold:*

1. e' is also an edge of C_j .
2. If e' is also a dangling edge of H , then the primary cycle of C_j consists of the two bypass edges corresponding to C and C' , where C' is the dangling 2-cycle of C_j containing e' .

3. If e' is not a dangling edge of H , then e' is the edge in the primary cycle of C_j whose head is the tail of the bypass edge corresponding to C .

Lemma 4. Fix a j with $1 \leq j \leq \ell$ such that the primary cycle C of C_j contains no bypass edge. Let u_1, \dots, u_k be a cyclic ordering of the vertices in C . Then, the following hold:

1. Suppose that there is a chain P in G_2 whose endpoints appear in C but not consecutively (i.e., its endpoints are not connected by an edge of C). Then, at least one edge of C is safe.
2. Suppose that every edge of C is critical. Then, there is a unique $C_{j'}$ with $j' \in \{1, \dots, \ell\} - \{j\}$ such that (1) the primary cycle C' of $C_{j'}$ has exactly k vertices and (2) the vertices of C' have a cyclic ordering v_1, \dots, v_k such that for every $1 \leq i \leq k$, u_i and v_{k-i+1} are the endpoints of some chain in G_2 . (See Figure 4.)

Now we are ready to describe how to partition $E(H)$ into three subsets each of which can be added to G_2 without destroying its 2-path-colorability. We use the three colors 0, 1, and 2 to represent the three subsets, and want to assign each edge of $E(H)$ a color in $\{0, 1, 2\}$ so that the following conditions are satisfied:

- (C1) For every critical edge e of H , e and its rival receive different colors.
- (C2) For every dangling 2-cycle C of H , the two edges in C receive the same color.
- (C3) If two adjacent edges of H receive the same color, then they form a 2-cycle of H .

To compute a coloring of the edges of H satisfying the above three conditions, we process C_1, \dots, C_ℓ in an arbitrary order. While processing C_j ($1 \leq j \leq \ell$), we color the edges of C_j by distinguishing four cases as follows (where C denotes the primary cycle of C_j):

Case 1: C is a 2-cycle. Then, C contains either one or two bypass edges. In the former (respectively, latter) case, we color the edges of C_j as shown in Figure 3(2) (respectively, Figure 3(1)). Note that the colored edges satisfy Conditions (C1) through (C3) above.

Case 2: Every Edge of C is Critical. Then, by Lemma 2, C contains no bypass edge. Let j' be the integer in $\{1, \dots, \ell\} - \{j\}$ such that $C_{j'}$ satisfies the two conditions (1) and (2) in Statement 2 in Lemma 4. Then, by Lemma 3 and Statement 2 in Lemma 4, neither C_j nor $C_{j'}$ has a bypass edge or a dangling 2-cycle. So, the primary cycle of C_j (respectively, $C_{j'}$) is C_j (respectively, $C_{j'}$) itself. We color the edges of C_j and $C_{j'}$ simultaneously as follows (see Figure 4). First, we choose one edge e of C_j , color e with 2, and color the rival of e with 0. Note that the uncolored edges of C_j form a path Q . Starting at one end of Q , we then color the edges of Q alternately with colors 0 and 1. Finally, for each uncolored edge e' of $C_{j'}$, we color it with the color $h \in \{1, 2\}$ such that the rival of e' has been colored with $h - 1$. Note that the colored edges satisfy Conditions (C1) through (C3) above.

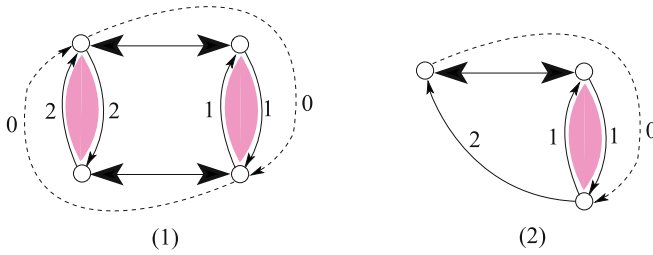


Fig. 3. Coloring C_j when its primary cycle is a 2-cycle

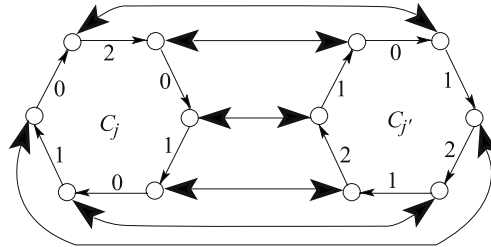


Fig. 4. Coloring C_j and $C_{j'}$ when all their edges are critical

Case 3: Neither Case 1 nor Case 2 Occurs and No Edge of C_j Is a Critical Dangling Edge of H . Then, by Lemma 2 and Statement 1 in Lemma 4, C contains at least one safe edge. Let e_1, \dots, e_k be the edges of C , and assume that they appear in C cyclically in this order. Without loss of generality, we may assume that e_1 is a safe edge. We color e_1 with 0, and then color the edges e_2, \dots, e_k in this order as follows. Suppose that we have just colored e_i with a color $h_i \in \{0, 1, 2\}$ and we want to color e_{i+1} next, where $1 \leq i \leq k - 1$. If e_{i+1} is a critical edge and its rival has been colored with $(h_i + 1) \bmod 3$, then we color e_{i+1} with $(h_i + 2) \bmod 3$; otherwise, we color e_{i+1} with $(h_i + 1) \bmod 3$. If e_k is colored 0 at the end, then we change the color of e_1 from 0 to the color in $\{1, 2\}$ that is not the color of e_2 . Now, we can further color each dangling 2-cycle C' of C_j with the color in $\{0, 1, 2\}$ that has not been used to color the two edges of C incident to the articulation vertex of C' . Note that the colored edges satisfy Conditions (C1) through (C3) above.

Case 4: Neither Case 1 Nor Case 2 Occurs and Some Edge of C_j Is a Critical Dangling Edge of H . For each dangling edge e of H with $e \in E(C_j)$, we define the partner of e to be the edge e' of C leaving the articulation vertex u of the dangling 2-cycle containing e , and define the mate of e to be the bypass edge e'' of C_j entering u (see Figure 6). We say that an edge e of C_j is bad if e is a critical dangling edge of H and its partner is the rival of another critical dangling edge of H . If C_j has a bad edge e , then Statement 3 in Lemma 3 ensures that C_j is as shown in Figure 5 and can be colored as shown there without violating Conditions (C1) through (C3) above.

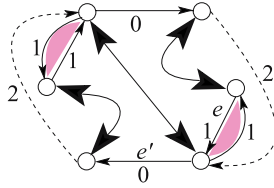


Fig. 5. C_j (formed by the one-way edges) and its coloring when it has a bad edge e

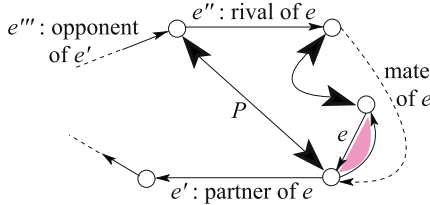


Fig. 6. The rival, the mate, and the partner of a critical dangling edge e of H together with the opponent of the partner of e

So, suppose that C_j has no bad edge. We need one more definition (see Figure 6). Consider a critical dangling edge e of H with $e \in E(C_j)$. Let e' and e'' be the partner and the rival of e , respectively. Let e''' be the edge of C entering the tail of e'' . Let P be the open chain in G_2 whose endpoints are the tails of e' and e'' . We call e''' the *opponent* of e' . Note that $e' \neq e'''$ because the endpoints of P are the tail of e' and the head of e''' . Moreover, if e' is a critical edge of H , then the rival of e' has to be e''' because e is not bad and P exists. In other words, whenever an edge of C has both its rival and its opponent, they must be the same. Similarly, if e''' is a critical edge of H , then its rival has to be e' . Obviously, neither e' nor e''' can be the rival or the mate of a critical dangling edge of H (because C_j has no bad edge).

Now, let e_1, \dots, e_q be the edges of C none of which is the rival or the mate of a critical dangling edge of C_j . We may assume that e_1, \dots, e_q appear in C cyclically in this order. Without loss of generality, we may further assume that e_1 is the partner of a critical dangling edge of H . Then, we color e_1 with 0, and further color e_2, \dots, e_q in this order as follows. Suppose that we have just colored e_i with a color $h_i \in \{0, 1, 2\}$ and we want to color e_{i+1} next, where $1 \leq i \leq q - 1$. If e_{i+1} is a critical edge of H and its rival or opponent has been colored with $(h_i + 1) \bmod 3$, then we color e_{i+1} with $(h_i + 2) \bmod 3$; otherwise, we color e_{i+1} with $(h_i + 1) \bmod 3$. Note that the colored edges satisfy Conditions (C1), (C2), (C3) above, because the head of e_q is not the tail of e_1 .

We next show how to color the rival and the mate of each critical dangling edge of C_j . For each critical dangling edge e of C_j , since its partner e' and the opponent of e' have been colored, we can color the rival of e with the color of e' and color the mate of e with a color in $\{0, 1, 2\}$ that is not the color of e' . Note that the colored edges satisfy Conditions (C1) through (C3) above, because e' and its opponent have different colors.

Finally, for each dangling 2-cycle D of C_j , we color the two edges of D with the color in $\{0, 1, 2\}$ that has not been used to color an edge incident to the articulation vertex of D . Note that the colored edges satisfy Conditions (C1) through (C3) above, because the rival of each critical dangling edge e of H has the same color as the partner of e does. This completes the coloring of C_j (and hence H).

We next want to show how to use the coloring to find a large-weight tour in G . For each $i \in \{0, 1, 2\}$, let E_i be the edges of H with color i . Without loss of generality, we may assume that $w(E_0) \geq \max\{w(E_1), w(E_2)\}$. Then, $w(E_0) \geq \frac{1}{3}W_{1,3}$ (see the beginning of this subsection for $W_{1,3}$). Consider the undirected graph $U = (V(G), F_1 \cup F_2)$, where F_1 consists of all edges $\{v_1, v_2\}$ such that (v_1, v_2) or (v_2, v_1) is an edge in E_0 , and F_2 consists of all edges $\{v_3, v_4\}$ such that v_3 and v_4 are the endpoints of an open chain in G_2 . We further assign a weight to each edge of F_1 as follows. We first initialize the weight of each edge of F_1 to be 0. For each edge $(v_1, v_2) \in E_0$, we then add the weight of edge (v_1, v_2) to the weight of edge $\{v_1, v_2\}$. Note that for each $i \in \{1, 2\}$, each connected component of the undirected graph $(V(G), F_i)$ is a single vertex or a single edge because of Condition (C3) above. So, each connected component of U is a path or a cycle. Moreover, each cycle of U contains at least three edges of F_1 because of Condition (C1) above. For each cycle D of U , we mark exactly one edge $\{v_1, v_2\} \in F_1$ in D whose weight is the smallest among all edges $\{v_1, v_2\} \in F_1$ in D . Let E_3 be the set of all edges $(v_1, v_2) \in E_0$ such that $\{v_1, v_2\}$ is marked. Then, $w(E_3) \leq \frac{1}{3}w(E_0)$. Consider the directed graph G'_2 obtained from G_2 by adding the edges of $E_0 - E_3$. Obviously, $w(G'_2) \geq (W_{1,2} + W_{2,2}) + \frac{1}{9}W_{1,3}$. Moreover, G'_2 is a collection of partial chains and hence is 2-path-colorable. So, we can partition the edges of G'_2 into two subsets E'_1 and E'_2 such that both graphs $(V(G), E'_1)$ and $(V(G), E'_2)$ are subtours of G . The heavier one among the two subtours can be completed to a tour of G of weight at least $\frac{1}{2}(W_{1,2} + W_{2,2}) + \frac{1}{18}W_{1,3} \geq W_2 + \frac{1}{9}W_3$. Combining this with Lemma 1, we now have:

Theorem 2. *There is a polynomial-time approximation algorithm for AsymMaxTSP achieving an approximation ratio of $\frac{27}{35}$.*

4 New Algorithm for Metric SymMaxTSP

Throughout this section, fix an instance (G, w) of metric SymMaxTSP, where G is a complete undirected graph with n vertices and w is a function mapping each edge e of G to a nonnegative real number $w(e)$. Because of the triangle inequality, the following fact holds (see [2] for a proof):

Fact 1. *Suppose that P_1, \dots, P_t are vertex-disjoint paths in G each containing at least one edge. For each $1 \leq i \leq t$, let u_i and v_i be the endpoints of P_i . Then, we can use some edges of G to connect P_1, \dots, P_t into a single cycle C in linear time such that $w(C) \geq \sum_{i=1}^t w(P_i) + \frac{1}{2} \sum_{i=1}^t w(\{u_i, v_i\})$.*

Like Hassin and Rubinfeld's algorithm (H&R2-algorithm) for the problem, our algorithm computes two tours T_1 and T_2 of G and outputs the one with

the larger weight. The first two steps of our algorithm are the same as those of H&R2-algorithm:

1. Compute a maximum-weight cycle cover \mathcal{C} . Let C_1, \dots, C_r be the cycles in G .
2. Compute a maximum-weight matching M in G .

Lemma 5. [2] *In linear time, we can compute two disjoint subsets A_1 and A_2 of $\bigcup_{1 \leq i \leq r} E(C_i) - M$ satisfying the following conditions:*

- (a) *For each $j \in \{1, 2\}$, each connected component of the graph $(V(G), M \cup A_j)$ is a path of length at least 1.*
- (b) *For each $j \in \{1, 2\}$ and each $i \in \{1, \dots, r\}$, $|A_j \cap E(C_i)| = 1$.*

For a technical reason, we will allow our algorithm to use only 1 random bit (so we can easily derandomize it, although we omit the details). The third through the seventh steps of our algorithm are as follows:

3. Compute two disjoint subsets A_1 and A_2 of $\bigcup_{1 \leq i \leq r} E(C_i) - M$ satisfying the two conditions in Lemma 5.
4. Choose A from A_1 and A_2 uniformly at random.
5. Obtain a collection of vertex-disjoint paths each of length at least 1 by deleting the edges in A from \mathcal{C} ; and then connect these paths into a single (Hamiltonian) cycle T_1 as described in Fact 1.
6. Let $S = \{v \in V(G) \mid \text{the degree of } v \text{ in the graph } (V, M \cup A) \text{ is } 1\}$ and $F = \{\{u, v\} \in E(G) \mid \{u, v\} \subseteq S\}$. Let H be the complete graph (S, F) . Let $\ell = \frac{1}{2}|S|$. (Comment: $|S|$ is even, because of Condition (a) in Lemma 5.)
7. Let M' be the set of all edges $\{u, v\} \in F$ such that some connected component of the graph $(V, M \cup A)$ contains both u and v . (Comment: M' is a perfect matching of H because of Condition (a) in Lemma 5.)

Lemma 6. [2] *Let $\alpha = w(A_1 \cup A_2)/w(\mathcal{C})$. For a random variable X , let $\mathcal{E}[X]$ denote its expected value. Then, $\mathcal{E}[w(F)] \geq \frac{1}{4}(1 - \alpha)(2\ell - 1)w(\mathcal{C})$.*

The next lemma shows that there cannot exist matchings of large weight in an edge-weighted graph where the weights satisfy the triangle inequality:

Lemma 7. *For every perfect matching N of H , $w(N) \leq w(F)/\ell$.*

The following is our main lemma whose proof is omitted for lack of space:

Lemma 8. *We can partition $F - M'$ into $2\ell - 2$ perfect matchings $M_1, \dots, M_{2\ell-2}$ of H in linear time satisfying the following condition:*

- *For every natural number q , there are at most $q^2 - q$ matchings M_i with $1 \leq i \leq 2\ell - 2$ such that the graph $(S, M' \cup M_i)$ has a cycle of length at most $2q$.*

Now, the eighth through the thirteenth steps of our algorithm are as follows:

8. Partition $F - M'$ into $2\ell - 2$ perfect matchings $M_1, \dots, M_{2\ell-2}$ of H in linear time satisfying the condition in Lemma 8.
9. Let $q = \lceil \sqrt[3]{\ell} \rceil$. Find a matching M_i with $1 \leq i \leq 2\ell - 2$ satisfying the following two conditions:
 - (a) The graph $(S, M' \cup M_i)$ has no cycle of length at most $2q$.
 - (b) $w(M_i) \geq w(M_j)$ for all matchings M_j with $1 \leq j \leq 2\ell - 2$ such that the graph $(S, M' \cup M_j)$ has no cycle of length at most $2q$.
10. Construct the graph $G'_i = (V(G), M \cup A \cup M_i)$. (Comment: $M_i \cap (M \cup A) = \emptyset$ and each connected component of G'_i is either a path, or a cycle of length $2q + 1$ or more.)
11. For each cycle D in G'_i , mark exactly one edge $e \in M_i \cap E(D)$ such that $w(e) \leq w(e')$ for all $e' \in M_i \cap E(D)$.
12. Obtain a collection of vertex-disjoint paths each of length at least 1 by deleting the marked edges from G'_i ; and then connect these paths into a single (Hamiltonian) cycle T_2 as described in Fact 1.
13. If $w(T_1) \geq w(T_2)$, output T_1 ; otherwise, output T_2 .

Theorem 3. *There is an $O(n^3)$ -time approximation algorithm for metric SymMaxTSP achieving an approximation ratio of $\frac{7}{8} - O(1/\sqrt[3]{n})$.*

Proof. Let OPT be the maximum weight of a tour in G . It suffices to prove that $\max\{\mathcal{E}[w(T_1)], \mathcal{E}[w(T_2)]\} \geq (\frac{7}{8} - O(1/\sqrt[3]{n}))OPT$. By Fact 1, $\mathcal{E}[w(T_1)] \geq (1 - \frac{1}{2}\alpha + \frac{1}{4}\alpha)w(\mathcal{C}) \geq (1 - \frac{1}{4}\alpha)OPT$.

We claim that $|S| \geq \frac{1}{3}n$. To see this, consider the graphs $G_M = (V(G), M)$ and $G_A = (V(G), M \cup A)$. Because the length of each cycle in \mathcal{C} is at least 3, $|A| \leq \frac{1}{3}n$ by Condition (b) in Lemma 5. Moreover, since M is a matching of G , the degree of each vertex in G_M is 0 or 1. Furthermore, G_A is obtained by adding the edges of A to G_M . Since adding one edge of A to G_M increases the degrees of at most two vertices, there exist at least $n - 2|A| \geq \frac{1}{3}n$ vertices of degree 0 or 1 in G_A . So, by Condition (a) in Lemma 5, there are at least $\frac{1}{3}n$ vertices of degree 1 in G_A . This establishes that $|S| \geq \frac{1}{3}n$. Hence, $\ell \geq \frac{1}{6}n$.

Now, let x be the number of matchings M_j with $1 \leq j \leq 2\ell - 2$ such that the graph $(S, M' \cup M_i)$ has a cycle of length at most $2q$. Then, by Lemmas 7 and 8, the weight of the matching M_i found in Step 9 is at least $(1 - \frac{x+1}{\ell}) \cdot w(F) \cdot \frac{1}{2\ell-2-x}$. So, $w(M_i) \geq \frac{1}{\ell} \cdot (1 - \frac{\ell-1}{2\ell-2-q^2+q}) \cdot w(F)$ because $x \leq q^2 - q$. Let N_i be the set of edges of M_i marked in Step 11. Then, $w(M_i - N_i) \geq \frac{q}{q+1} \cdot \frac{\ell-q^2+q-1}{\ell(2\ell-2-q^2+q)} \cdot w(F)$. Hence, by Lemma 6 and the inequality $\ell \geq \frac{1}{6}n$, we have $\mathcal{E}[w(M_i - N_i)] \geq \frac{1}{4}(1 - \alpha)(1 - O(1/\sqrt[3]{n}))w(\mathcal{C})$.

Obviously, $\mathcal{E}[w(T_2)] \geq \mathcal{E}[w(M \cup A)] + \mathcal{E}[w(M_i - N_i)] \geq (\frac{1}{2} - \frac{1}{2n})OPT + \frac{1}{2}\alpha w(\mathcal{C}) + \mathcal{E}[w(M_i - N_i)]$. Hence, by the last inequality in the previous paragraph, $\mathcal{E}[w(T_2)] \geq (\frac{3}{4} + \frac{1}{4}\alpha - O(1/\sqrt[3]{n}))OPT$. Combining this with the inequality $\mathcal{E}[w(T_1)] \geq (1 - \frac{1}{4}\alpha)OPT$, we finally have $\mathcal{E}[\max\{w(T_1), w(T_2)\}] \geq (\frac{7}{8} - O(1/\sqrt[3]{n}))OPT$.

The running time of the algorithm is dominated by the $O(n^3)$ time needed for computing a maximum-weight cycle cover and a maximum-weight matching.

References

1. A. I. Barvinok, D. S. Johnson, G. J. Woeginger, and R. Woodroffe. Finding Maximum Length Tours under Polyhedral Norms. *Proceedings of the Sixth International Conference on Integer Programming and Combinatorial Optimization (IPCO)*, Lecture Notes in Computer Science, **1412** (1998) 195–201.
2. Z.-Z. Chen, Y. Okamoto, and L. Wang. Improved Deterministic Approximation Algorithms for Max TSP. To appear in *Information Processing Letters*.
3. R. Hassin and S. Rubinfeld. A $7/8$ -Approximation Approximations for Metric Max TSP. *Information Processing Letters*, **81** (2002) 247–251.
4. H. Kaplan, M. Lewenstein, N. Shafrir, and M. Sviridenko. Approximation Algorithms for Asymmetric TSP by Decomposing Directed Regular Multigraphs. *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, pp. 56–75, 2003.
5. A. V. Kostochka and A. I. Serdyukov. Polynomial Algorithms with the Estimates $\frac{3}{4}$ and $\frac{5}{6}$ for the Traveling Salesman Problem of Maximum (in Russian). *Upravlyaemye Sistemy*, **26** (1985) 55–59.

Unbalanced Graph Cuts

Ara Hayrapetyan^{1,*}, David Kempe^{2,**}, Martin Pál^{3,***}, and Zoya Svitkina^{1,†}

¹Dept. of Computer Science, Cornell University
{ara, zoya}@cs.cornell.edu

²Dept. of Computer Science, University of Southern California
dkempe@usc.edu

³DIMACS Center, Rutgers University
mpal@dimacs.rutgers.edu

Abstract. We introduce the MINIMUM-SIZE BOUNDED-CAPACITY CUT (MINSBCC) problem, in which we are given a graph with an identified source and seek to find a cut minimizing the number of nodes on the source side, subject to the constraint that its capacity not exceed a prescribed bound B . Besides being of interest in the study of graph cuts, this problem arises in many practical settings, such as in epidemiology, disaster control, military containment, as well as finding dense subgraphs and communities in graphs.

In general, the MINSBCC problem is NP-complete. We present an efficient $(\frac{1}{\lambda}, \frac{1}{1-\lambda})$ -bicriteria approximation algorithm for any $0 < \lambda < 1$; that is, the algorithm finds a cut of capacity at most $\frac{1}{\lambda}B$, leaving at most $\frac{1}{1-\lambda}$ times more vertices on the source side than the optimal solution with capacity B . In fact, the algorithm's solution *either* violates the budget constraint, *or* exceeds the optimal number of source-side nodes, but not both. For graphs of bounded treewidth, we show that the problem with unit weight nodes can be solved optimally in polynomial time, and when the nodes have weights, approximated arbitrarily well by a PTAS.

1 Introduction

Graph cuts are among the most well-studied objects in theoretical computer science. In the most pristine form of the problem, two given vertices s and t have to be separated by removing an edge set of minimum capacity. By a fundamental result of Ford and Fulkerson [16], such an edge set can be found in polynomial time. Since then, many problems have been shown to reduce to graph cut problems, sometimes quite surprisingly (e.g. [19]). One way to view the Min-Cut Problem is to think of “protecting” the sink node t from the presumably harmful node s by way of removing edges: the capacity of the cut then corresponds to the cost of edge removals. This interpretation in turn suggests a very natural variant of the graph cut problem: given a node s and a bound B on the total edge removal cost, try to “protect” as many nodes from s as possible, while cutting

* Supported in part by NSF grant CCR-0325453.

** Work done while supported by an NSF graduate fellowship.

*** Supported by NSF grant EIA 02-05116, and ONR grant N00014-98-1-0589.

† Supported in part by NSF grant CCR-0325453 and ONR grant N00014-98-1-0589.

at most a total edge capacity of B . In other words, find an s - t cut of capacity at most B , minimizing the size of the s -side of the cut. This is the MINIMUM-SIZE BOUNDED-CAPACITY CUT (MINSBCC) problem that we study.

Naturally, the MINSBCC problem has direct applications in the areas of disaster, military, or crime containment. In all of these cases, a limited amount of resources can be used to monitor or block the edges by which the disaster could spread, or people could escape. At the same time, the area to which the disaster is confined should be as small as possible. For instance, in the firefighter's problem [6], a fixed small number of firefighters must confine a fire to within a small area, trying to minimize the value of the property and lives inside.

Perhaps even more importantly, the MINSBCC problem arises naturally in the control of epidemic outbreaks. While traditional models of epidemics [4] have ignored the network structure in order to model epidemic diseases via differential equations, recent work by Eubank et al. [7,9], using highly realistic large-scale simulations, has shown that the graph structure of the social contacts has a significant impact on the spread of the epidemic, and crucially, on the type of actions that most effectively contain the epidemic. If we assume that patient 0, the first infected member of the network, is known, then the problem of choosing which individuals to vaccinate in order to confine the epidemic to a small set of people is exactly the node cut version of the MINSBCC problem.

Besides the obvious connections to the containment of damage or epidemics, the MINSBCC problem can also be used for finding small dense subgraphs and communities in graphs. Discovering communities in graphs has received much attention recently, in the context of analyzing social networks and the World Wide Web [14,20]. It involves examining the link structure of the underlying graph so as to extract a small set of nodes sharing a common property, usually expressed by high internal connectivity, sometimes in combination with small expansion. We show how to reduce the community finding problem to MINSBCC.

Our Results. Formally, we define the MINSBCC problem as follows. Given an (undirected or directed) graph $G = (V, E)$ with edge capacities c_e , source and sink nodes s and t , as well as a total capacity bound (also called the *budget*) B , we wish to find an s - t cut (S, \bar{S}) , $s \in S$ of capacity no more than B , which leaves as few nodes on the source side as possible. We will also consider a generalization in which the nodes are assigned weights w_v , and the objective is to minimize the total node weight $\sum_{v \in S} w_v$, subject to the budget constraint.¹

We show in Sections 2 and 4.2 that MINSBCC is NP-hard on general graphs with uniform node weights, and on trees with non-uniform node weights. We therefore develop two $(\frac{1}{\lambda}, \frac{1}{1-\lambda})$ -bicriteria approximation algorithms for MINSBCC, where $0 < \lambda < 1$. These algorithms, in polynomial time, find a cut (S, \bar{S}) of capacity at most $\frac{1}{\lambda}B$, such that the size of S is at most $\frac{1}{1-\lambda}$ times that of S^* , where (S^*, \bar{S}^*) is the optimal cut of capacity at most B . The first algorithm obtains this guarantee by a simple rounding of a linear programming relaxation

¹ Some of our motivating examples and applications do not specify a sink; this can be resolved by adding an isolated sink to the graph.

of MINSBCC. The second one bypasses solving the linear program by running a single parametric maximum flow computation and is thus very efficient [17]. It also has a better guarantee: it outputs either a $(\frac{1}{\lambda}, 1)$ -approximation or a $(1, \frac{1}{1-\lambda})$ -approximation, thus violating at most one of the constraints by the corresponding factor. The analysis of this algorithm is based on the same linear programming formulation of MINSBCC and its Lagrangian relaxation.

We then investigate the MINSBCC problem for graphs of bounded treewidth in Section 3. We give a polynomial-time algorithm based on dynamic programming to solve MINSBCC optimally for graphs of bounded treewidth with unit node weights. We then extend the algorithm to a PTAS for general node weights.

Section 4 discusses the reductions from node cut and dense subgraph problems to MINSBCC. We conclude with directions for future work in Section 5.

Related Work. Minimum cuts have a long history of study and form part of the bread-and-butter of everyday work in algorithms [1]. While minimum cuts can be computed in polynomial time, additional constraints on the size of the cut or on the relationship between its capacity and size (such as its density) usually make the problem NP-hard.

Much recent attention has been given to the computation of *sparse cuts*, partly due to their application in divide-and-conquer algorithms [24]. The seminal work of Leighton and Rao [22] gave the first $O(\log n)$ approximation algorithm for sparsest and balanced cut problems using region growing techniques. This work was later extended by Garg, Vazirani, and Yannakakis [18]. In a recent breakthrough result, the approximation factor for these problems was improved to $O(\sqrt{\log n})$ by Arora, Rao, and Vazirani [2].

A problem similar to MINSBCC is studied by Feige et al. [11,12]: given a number k , find an s - t cut (S, \bar{S}) with $|S| = k$ of minimum capacity. They obtain an $O(\log^2 n)$ approximation algorithm in the general case [11], and improve the approximation guarantees when k is small [12].

MINSBCC has a natural maximization version MAXSBCC, where the goal is to maximize the size of the s -side of the cut instead of minimizing it, while still obeying the capacity constraint. This problem was recently introduced by Svitkina and Tardos [25]. Based on the work of Feige and Krauthgamer [11], Svitkina and Tardos give an $(O(\log^2 n), 1)$ -bicriteria approximation which is used as a black box to obtain an approximation algorithm for the min-max multiway cut problem, in which one seeks a multicut minimizing the number of edges leaving any one component. The techniques in [25] readily extend to an $(O(\log^2 n), 1)$ -bicriteria approximation for the MINSBCC problem.

Recently, and independently of our work, Eubank, et al [8] also studied the MINSBCC problem and gave a weaker $(1 + 2\lambda, 1 + \frac{2}{\lambda})$ -bicriteria approximation.

2 Bicriteria Approximation Algorithms

We first establish the NP-completeness of MINSBCC.

Proposition 1. *The MINSBCC problem with arbitrary edge capacities and node weights is NP-complete even when restricted to trees.*

Proof. We give a reduction from KNAPSACK. Let the KNAPSACK instance consist of items $1, \dots, n$ with sizes s_1, \dots, s_n and values a_1, \dots, a_n , and let the total Knapsack size be B . We create a source s , a sink t , and a node v_i for each item i . The node weight of v_i is a_i , and it is connected to the source by an edge of capacity s_i . The sink t has weight 0, and is connected to v_1 by an edge of capacity 0. The budget for the MINSBCC problem is B .

The capacity of any s - t cut is exactly the total size of the items on the t -side, and minimizing the total node weight on the s -side is equivalent to maximizing the total value of items corresponding to nodes on the t -side. ■

Now, we turn our attention to our main approximation results, which are the two $(\frac{1}{\lambda}, \frac{1}{1-\lambda})$ -bicriteria approximation algorithms for MINSBCC on general graphs. For the remainder of the section, we will use $\delta(S)$ to denote the capacity of the cut (S, \bar{S}) in G . We use S^* to denote the minimum-size set of nodes such that $\delta(S^*) \leq B$, i.e. $(S^*, V \setminus S^*)$ is the optimum cut of capacity at most B .

The analysis of both of our algorithms is based on the following linear programming (LP) relaxation of the natural integer program for MINSBCC. We use a variable x_v for every vertex $v \in V$ to denote which side of the cut it is on, and a variable y_e for every edge e to denote whether or not the edge is cut.

$$\begin{aligned}
 &\text{Minimize } \sum_{v \in V} x_v \\
 &\text{subject to } x_s = 1 \\
 &\quad x_t = 0 \\
 &\quad y_e \geq x_u - x_v \quad \text{for all } e = (u, v) \in E \\
 &\quad \sum_{e \in E} y_e \cdot c_e \leq B \\
 &\quad x_v, y_e \geq 0
 \end{aligned} \tag{1}$$

2.1 Randomized Rounding-Based Algorithm

Our first algorithm is based on randomized rounding of the solution to (1).

Algorithm 1. Randomized LP-rounding algorithm with parameter λ

- 1: Let (x^*, y^*) be the optimal solution to LP (1).
 - 2: Choose $\ell \in [1 - \lambda, 1]$ uniformly at random.
 - 3: Let $S = \{v \mid x_v^* \geq \ell\}$, and output S .
-

Theorem 1. *The Randomized Rounding algorithm (Algorithm 1) outputs a set S of size at most $\frac{1}{1-\lambda}$ times the LP objective value. The expected capacity of the cut (S, \bar{S}) is at most $\frac{1}{\lambda}B$.*

Proof. To prove the first statement of the theorem, observe that for each $v \in S$, $x_v^* \geq \ell \geq 1 - \lambda$. Therefore $\sum_{v \in V} x_v^* \geq \sum_{v \in S} x_v^* \geq (1 - \lambda)|S|$.

For the second statement, observe that ℓ is selected uniformly at random from an interval of size λ . Furthermore, an edge $e = (u, v)$ will be cut only if ℓ lies between x_u^* and x_v^* . The probability of this happening is thus at most

$\frac{|x_u^* - x_v^*|}{\lambda} \leq \frac{y_e^*}{\lambda}$. Summing over all edges yields that the expected total capacity of the cut is at most $\sum_e \frac{c_e y_e^*}{\lambda} \leq \frac{1}{\lambda} B$. Notice, that the above algorithm can be derandomized by trying all values $l = x_v^*$, since there are at most $|V|$ of those. ■

2.2 A Parametric Flow-Based Algorithm

Next, we show how to avoid solving the LP, and instead compute the cuts directly via a parametric max-flow computation. This analysis will also show that in fact, at most one of the two criteria is approximated, while the other is preserved.

Algorithm Description: The algorithm searches for candidate solutions among the parametrized minimum cuts in the graph G^α , which is obtained from G by adding an edge of capacity α from every vertex v to the sink t (introducing parallel edges if necessary). Here, α is a parameter ranging over non-negative values. Observe that the capacity of a cut (S, \bar{S}) in the graph G^α is $\alpha|S| + \delta(S)$, so the minimum s - t cut in G^α minimizes $\alpha|S| + \delta(S)$.

Initially, as $\alpha = 0$, the min-cut of G^α is the min-cut of G . As α increases, the source side of the min-cut of G^α will contain fewer and fewer nodes, until eventually it contains the single node $\{s\}$. All these cuts for the different values of α can be found efficiently using a single run of the push relabel algorithm. Moreover, the source sides of these cuts form a nested family $S_0 \supset S_1 \supset \dots \supset S_k$ of sets [17]. (S_0 is the minimum s - t cut in the original graph, and $S_k = \{s\}$). Our solution will be one of these cuts S_j .

We first observe that $\delta(S_i) < \delta(S_j)$ if $i < j$; for if it were not, then S_j would be a superior cut to S_i for all values of α . If $\delta(S_k) \leq B$, then, of course, $\{s\}$ is the optimal solution. On the other hand, if $\delta(S_0) > B$, then no solution exists. In all other cases, choose i such that $\delta(S_i) \leq B \leq \delta(S_{i+1})$. If $\delta(S_{i+1}) \leq \frac{1}{\lambda} B$, then output S_{i+1} ; otherwise, output S_i .

Theorem 2. *The above algorithm produces either (1) a cut S^- such that $\delta(S^-) \leq B$ and $|S^-| \leq \frac{1}{1-\lambda}|S^*|$, or (2) a cut S^+ such that $\delta(S^+) \leq \frac{1}{\lambda} B$ and $|S^+| \leq |S^*|$.*

Proof. For the index i chosen by the algorithm, we let $S^- = S_i$ and $S^+ = S_{i+1}$. Hence, $\delta(S^-) \leq B \leq \delta(S^+)$.

First, observe that $|S^+| \leq |S^*|$, or else the parametric cut procedure would have returned S^* instead of S^+ . If S^+ also satisfies $\delta(S^+) \leq \frac{1}{\lambda} B$, then we are done. In the case that $\delta(S^+) > \frac{1}{\lambda} B$, we will prove that $|S^-| \leq \frac{1}{1-\lambda}|S^*|$.

Because S^+ and S^- are neighbors in our sequence of parametric cuts, there is a value of α , call it α^* , for which both are minimum cuts of G^{α^*} . Applying the Lagrangian Relaxation technique, we remove the constraint $\sum_e y_e c_e \leq B$ from LP (1) and put it into the objective function using the constant α^* .

$$\begin{aligned}
 &\text{Minimize } \alpha^* \cdot \sum_{v \in V} x_v + \sum_{e \in E} y_e \cdot c_e \\
 &\text{subject to } x_s = 1 \\
 &\quad x_t = 0 \\
 &\quad y_e \geq x_u - x_v \quad \text{for all } e = (u, v) \in E \\
 &\quad x_v, y_e \geq 0
 \end{aligned} \tag{2}$$

Lemma 1. *LP (2) has an integer optimal solution.*

Proof. Recall that in G^{α^*} we added edges of capacity α^* from every node to the sink. Extend any solution of LP (2) to these edges by setting $y_e = x_v - x_t = x_v$ for the newly added edge e connecting v to t . We claim that after this extension, the objective function of LP (2) is equivalent to $\sum_{e \in G^{\alpha^*}} y_e c'_e$, where c'_e is the edge capacity in the graph G^{α^*} . Indeed, this claim follows from observing that the first part of the objective of LP (2) is identical to the contribution that the newly added edges of G^{α^*} are making towards $\sum_{e \in G^{\alpha^*}} y_e c'_e$.

Consider a fractional optimal solution (\hat{x}, \hat{y}) to LP (2) with objective function value $L^* = \sum_{e \in G^{\alpha^*}} \hat{y}_e c'_e$. As this is an optimal solution, we can assume without loss of generality that $y_e = \max(0, x_u - x_v)$ for all edges $e = (u, v)$. So if we define $w_x = \sum_{u,v: x_u \geq x \geq x_v} c'_{uv}$, then $L^* = \int_0^1 w_x dx$.

Also, for any $x \in (0, 1)$, we can obtain an integral solution to LP (2) whose objective function value is w_x by rounding \hat{x}_v to 0 if it is no more than x , and to 1 otherwise (and setting $y_{uv} = \max(0, x_u - x_v)$). Since this process yields feasible solutions, we know that $w_x \geq L^*$ for all x . On the other hand, L^* is a weighted average (integral) of w_x 's, and hence in fact $w_x = L^*$ for all x , and any of the rounded solutions is an integral optimal solution to LP (2). ■

Notice that feasible integral solutions to LP (2) correspond to s - t cuts in G^{α^*} . Therefore, by Lemma 1, the optimal solutions to LP (2) are the minimum s - t cuts in G^{α^*} . In particular, S^+ and S^- are two such cuts. From S^+ and S^- , we naturally obtain solutions to LP (2), by setting $x_v^+ = 1$ for $v \in S^+$ and $x_v^+ = 0$ otherwise, with $y_e^+ = 1$ if e is cut by (S^+, \bar{S}^+) , and 0 otherwise (similarly for S^-). By definition of α^* , both (x^+, y^+) and (x^-, y^-) are then optimal solutions to LP (2). Thus, their linear combination $(x^*, y^*) = \ell \cdot (x^+, y^+) + (1 - \ell) \cdot (x^-, y^-)$ is also an optimal feasible solution. Choose ℓ such that

$$\ell \cdot \sum_{e \in E} y_e^+ c_e + (1 - \ell) \cdot \sum_{e \in E} y_e^- c_e = B. \tag{3}$$

Such an ℓ exists because our choice of S^- and S^+ ensured that $\delta(S^-) \leq B \leq \delta(S^+)$. For this choice of ℓ , the fractional solution (x^*, y^*) , in addition to being optimal for the Lagrangian relaxation, also satisfies the constraint $\sum_e y_e^* c_e \leq B$ of LP (1) with equality, implying that it is optimal for LP (1) as well. Crudely bounding the second term in Equation (3) by 0, we obtain that $\delta(S^+) = \sum_{e \in E} y_e^+ c_e \leq \frac{B}{\ell}$.

As we assumed that $\delta(S^+) > \frac{B}{\lambda}$, we conclude that $\ell < \lambda$. Because (x^*, y^*) is an optimal solution to LP (1), it provides the lower bound $\sum_v x_v^* \leq |S^*|$, and the fact that $x_v^* \geq (1 - \ell)x_v^-$ now implies that $|S^-| = \sum_{v \in V} x_v^- \leq \frac{|S^*|}{1 - \ell} \leq \frac{1}{1 - \lambda} \cdot |S^*|$.

Hence, in this case, S^- meets the capacity constraint, and exceeds the optimal size by at most a factor of $\frac{1}{1 - \lambda}$. ■

Both of the above algorithms can be extended with simple modifications to allow for node weights in addition to edge capacities.

3 Bounded Treewidth

As we saw in Section 2, the MINSBCC problem is NP-complete even on trees when both node weights and edge capacities are allowed. However, if all nodes have unit weights, then the problem can be solved in polynomial time for graphs of bounded treewidth, via a dynamic programming algorithm. In order to present the intuition behind our algorithm, we first describe it for trees, and then extend it to graphs of bounded treewidth (see [19] for a review of tree decompositions).

3.1 An Algorithm for Trees

We root the tree at the source node s and direct all edges away from s . When all edges have capacity 1, then clearly, only edges incident with s should be cut. They must include the edge on the unique s - t path, and in addition, the edges to the roots of the largest subtrees. Choosing these B edges gives the smallest possible size for the s -side of the cut.

For the case of general edge capacities, consider the tree T_v rooted at a node v , together with the edge e_v into v . We define the quantity a_v^k to be the smallest total capacity of edges in T_v that must be cut if at most k nodes of T_v are to be included in the source side of the cut. Notice that $a_v^0 = c_{e_v}$. Also, as the sink must always be excluded, we have $a_t^k = c_{e_t}$ for all k .

For a leaf v , we have $a_v^0 = c_{e_v}$, and $a_v^k = 0$ for $k > 0$. For an internal node v with children v_1, \dots, v_d , we can either cut the edge e_v into v , or otherwise include v and solve the problem recursively for the children of v , hence

$$a_v^k = \min(c_{e_v}, \min_{k_1 \geq 0, \dots, k_d \geq 0: \sum k_i = k-1} \sum_i a_{v_i}^{k_i}) \quad \text{for } k > 0, v \neq t.$$

Note that the optimal partition into k_i 's can be found in polynomial time by a nested dynamic programming subroutine that uses optimal partitions of each k into $k_1 \dots k_j$ in order to calculate the optimal partition into $k_1 \dots k_{j+1}$.

Once we have computed a_s^k at the source s for all values of k , we simply pick the smallest k^* such that $a_s^{k^*} \leq B$.

3.2 An Algorithm for Graphs with Bounded Treewidth

Recall [19] that a graph $G = (V, E)$ has treewidth θ if there exists a tree T , and subsets $V_w \subseteq V$ of nodes associated with each vertex w of T , such that:

1. Every node $v \in V$ is contained in some subset V_w .
2. For every edge $e = (u, v) \in E$, some set V_w contains both u and v .
3. If \hat{w} lies on the path between w and w' in T , then $V_w \cap V_{w'} \subseteq V_{\hat{w}}$.
4. $|V_w| \leq \theta + 1$ for all vertices w of the tree T .

The pair $(T, \{V_w\})$ is called a *tree decomposition* of G , and the sets V_w will be called *pieces*. It can be shown that for any two neighboring vertices w and w' of the tree T , the deletion of $V_w \cap V_{w'}$ from G disconnects G into two components, just as the deletion of the edge (w, w') would disconnect T into two components.

We assume that we are given a tree decomposition $(T, \{V_w\})$ for G with treewidth θ [5]. To make sure that each edge of the original graph is accounted for exactly once by the algorithm, we partition the set E by mapping each edge in it to one of the nodes in the decomposition tree. In other words, we associate with each node $w \in T$ a set $E_w \subseteq E \cap (V_w \times V_w)$ of edges both of whose endpoints lie in V_w , such that each edge appears in exactly one set E_w ; if an edge lies entirely in V_w for several nodes w , we assign it arbitrarily to one of them. We will identify some node r of the tree T with $s \in V_r$ as being the *root*, and consider the edges of T as directed away from r .

Let $W \subseteq T$ be the set of nodes in the subtree rooted at some node w , $E_W = \bigcup_{u \in W} E_u$, and $V_W = \bigcup_{u \in W} V_u$. Also, let $U, U' \subseteq V_w$ be arbitrary disjoint sets of nodes. We define $a_w^k(U, U')$ to be the minimum capacity of edges from E_W that must be cut by any set $S \subseteq V_W$ such that $S \supseteq U$, $S \cap U' = \emptyset$, the sink t is not included in S (i.e., $t \notin S$), and $|S \setminus U| \leq k$. But for the extension regarding the sets U and U' , this is exactly the same quantity we were considering in the case of trees. Also, notice that the minimum size of any cut of capacity at most B is the smallest k for which $a_r^{k-1}(\{s\}, \emptyset) \leq B$.

Our goal is to derive a recurrence relation for $a_w^k(U, U')$. At any stage, we will be taking a minimum over all subsets that meet the constraints imposed by the size k and the sets U, U' . We therefore write $\mathcal{S}_w^k(U, U') = \{S \mid U \subseteq S \subseteq V_w, S \cap U' = \emptyset, t \notin S, |S| \leq k\}$. The size of $\mathcal{S}_w^k(U, U')$ is $O(2^\theta)$. The cost incurred by cutting edges assigned to w is denoted by $\beta_w(S) = \sum_{e \in E_w \cap e(S, V_w \setminus S)} c_e$, and can be computed efficiently.

If w is a leaf node, then we can include up to k additional nodes, so long as the constraints imposed by the sets U and U' are not violated. Hence,

$$a_w^k(U, U') = \min_{S \in \mathcal{S}_w^k(U, U')} \beta_w(S).$$

For a non-leaf node w , let w_1, \dots, w_d denote its children. We can include an arbitrary subset of nodes, so long as we add no more than k nodes, and do not violate the constraints imposed by the sets U and U' . The remaining additional nodes can then be divided among the children of w in any way desired. Once we have decided to include (or exclude) a node $v \in V_w$, this decision must be respected by all children, i.e., we obtain different sets as constraints for the children. Notice that any node v contained in the pieces at two descendants of w must also be in the piece at w itself by property 3 of a tree decomposition. Also, by the same property, any node v from V_w that is not in V_{w_i} (for some child w_i of w) will not be in the piece at any descendant of w_i , and hence the information about v being forbidden or forced to be included is irrelevant in the subtree rooted at w_i . We hence have the following recurrence:

$$a_w^k(U, U') = \min_{S \in \mathcal{S}_w^k(U, U')} \min_{\{k_i\}: \sum k_i = k - |S \setminus U|} (\beta_w(S) + \sum_{i=1}^d a_{w_i}^{k_i}(S \cap V_{w_i}, (V_w \setminus S) \cap V_{w_i})).$$

As before, for any fixed set S , the minimum over all combinations of k_i values can be found by a nested dynamic program.

By induction over the tree, we can prove that this recursive definition actually coincides with the initial definition of $a_w^k(U, U')$, and hence that the algorithm is correct. The computation of $a_w^k(U, U')$ takes time $O(d \cdot k \cdot 2^\theta) = O(n^2 \cdot 2^\theta)$. For each node, we need to compute $O(n^2 \cdot 4^\theta)$ values, so the total running time is $O(n^4 \cdot 8^\theta)$, and the space requirement is $O(n^2 \cdot 4^\theta)$. To summarize, we have proved the following theorem:

Theorem 3. *For graphs of treewidth bounded by θ , there is an algorithm that finds, in polynomial time $O(8^\theta n^4)$, an optimal MINSBCC.*

3.3 A PTAS for the Node-Weighted Version

We conclude by showing how to extend the above algorithm to a polynomial-time approximation scheme (PTAS) for MINSBCC with arbitrary node weights.

Suppose we want a $(1 + 2\epsilon)$ guarantee. Let S^* denote the optimal solution and OPT denote its value. We first guess W such that $OPT \leq W \leq 2 \cdot OPT$ (test all powers of 2). Next, we remove all *heavy* nodes, i.e. those whose weight is more than W . We then rescale the remaining node weights w_v to $w'_v := \lceil \frac{w_v n}{\epsilon W} \rceil$. Notice that the largest node weight is now at most $\frac{n}{\epsilon}$. Hence, we can run the dynamic programming algorithm on the rescaled graph in polynomial time.

We now bound the cost of the obtained solution, which we call S . The scaled weight of the solution S^* is at most $\sum_{v \in S^*} \lceil \frac{w_v n}{\epsilon W} \rceil \leq \frac{n}{\epsilon W} OPT + n$ (since $|S^*| \leq n$). Since S^* is a feasible solution for the rescaled problem, the solution S found by the algorithm has (rescaled) weight no more than that of S^* . Thus, the original weight of S is at most $(OPT + \epsilon W)$. Considering that $W \leq 2 \cdot OPT$, we obtain the desired guarantee, namely that the cost of S is at most $(1 + 2\epsilon)OPT$.

4 Applications

4.1 Epidemiology and Node Cuts

Some important applications, such as vaccination, are phrased much more naturally in terms of node cuts than edge cuts. Here, each node has a *weight* w_v , the cost of including it on the s -side of the cut, and a *capacity* c_v , the cost of removing (cutting) it from the graph. The goal is to find a set $R \subseteq V$, not containing s , of capacity $c(R)$ not exceeding a budget B , such that after removing R , the connected component S containing s has minimum total weight $w(S)$.

This problem can be reduced to (node-weighted) MINSBCC in the standard way. First, if the original graph G is undirected, we bidirect each edge. Now, each vertex v is split into two vertices v_{in} and v_{out} ; all edges into v now enter v_{in} , while all edges out of v now leave v_{out} . We add a directed edge from v_{in} to v_{out} of capacity c_v . Each originally present edge, i.e., each edge into v_{in} or out of v_{out} , is given infinite capacity. Finally, v_{in} is given node weight 0, and v_{out} is given node weight w_v . Call the resulting graph G' .

Now, one can verify that (1) no edge cut in G' ever cuts any originally present edges, (2) the capacity of an edge cut in G' is equal to the node capacity of a

node cut in G , and (3) the total node weight on the s -side of an edge cut in G' is exactly the total node weight in the s component of the corresponding node cut in G . Hence an approximation algorithm for MINSBCC carries to node-cuts.

4.2 Graph Communities

Identifying “communities” has been an important and much studied problem for social or biological networks, and more recently, the web graph [14,15]. Different mathematical formalizations for the notion of a community have been proposed, but they usually share the property that a community is a node set with high edge density within the set, and comparatively small expansion.

It is well known [21] that the *densest subgraph*, i.e., the set S maximizing $\frac{c(S)}{|S|} := \frac{c(S,S)}{|S|}$ can be found in polynomial time via a reduction to MIN-CUT. On the other hand, if the size of the set S is prescribed to be at most k , then the problem is the well-studied *densest k -subgraph problem* [3,10,13], which is known to be NP-complete, with the best known approximation ratio of $O(n^{1/3-\epsilon})$ [10]. We consider the converse of the densest k -subgraph problem, in which the density of the subgraph is given, and the size has to be minimized.

The definition of a graph community as the densest subgraph has the disadvantage that it lacks specificity. For example, adding a high-degree node tends to increase the density of a subgraph, but intuitively such a node should not belong to the community. The notion of a community that we consider avoids this difficulty by requiring that a certain fraction of a community’s edges lie inside of it. Formally, let an α -community be a set of nodes S with $\frac{c(S)}{d(S)} \geq \alpha$, where $d(S)$ is the sum of degrees of nodes in S . This definition is a relaxation of one introduced by Flake et al. [14] and is used in [23]. We are interested in finding such communities of smallest size.

The problem of finding the smallest α -community and the problem of finding the smallest subgraph of a given density have a common generalization, which is obtained by defining a node weight w_v which is equal to node degree for the former problem and to 1 for the latter. We show how to reduce this general size minimization problem to MINSBCC in an approximation-preserving way. In particular, by applying this reduction to the densest k -subgraph problem, we show that MINSBCC is NP-hard even for the case of unit node weights.

Given a graph $G = (V, E)$ with edge capacities c_e , node weights w_v , and a specified node $s \in V$, we consider the problem of finding the smallest (in terms of the number of nodes) set S containing s with $\frac{c(S)}{w(S)} \geq \alpha$. (The version where s is not specified can be reduced to this one by trying all nodes s .) We modify G to obtain a graph G' as follows. Add a sink t , connect each vertex v to the source s with an edge of capacity $d(v) := \sum_u c(v,u)$, and to the sink with an edge of capacity $2\alpha w_v$. The capacity for all edges $e \in E$ stays unchanged.

Theorem 4. *A set $S \subseteq V$ with $s \in S$ has $\frac{c(S)}{w(S)} \geq \alpha$ if and only if $(S, \bar{S} \cup \{t\})$ is an s - t cut of capacity at most $2c(V) = 2 \sum_{e \in E} c_e$ in G' .*

Notice that this implies that any approximation guarantees on the size of S carry over from the MINSBCC problem to the problem of finding communities. Also notice that by making all node weights and edge capacities 1, and setting $\alpha = \frac{k-1}{2}$, a set S of size at most k satisfies $\frac{c(S)}{w(S)} \geq \alpha$ if and only if S is a k -clique. Hence, the MinSBCC problem is NP-hard even with unit node weights. However, the approximation hardness of CLIQUE does not carry over, as the reduction requires the size k to be known.

Proof. The required condition can be rewritten as $c(S) - \alpha w(S) \geq 0$. As

$$2(c(S) - \alpha w(S)) = 2c(V) - (c(S, \bar{S}) + \sum_{v \in \bar{S}} d(v) + 2\alpha w(S)),$$

we find that S is an α -community iff $c(S, \bar{S}) + \sum_{v \in \bar{S}} d(v) + 2\alpha w(S) \leq 2c(V)$. The quantity on the left is the capacity of the cut $(S, \bar{S} \cup \{t\})$, proving the theorem. ■

5 Conclusion

In this paper, we present a new graph-theoretic problem called the minimum-size bounded-capacity cut problem, in which we seek to find unbalanced cuts of bounded capacity. Much attention has already been devoted to balanced and sparse cuts [24,22,18,2]; we believe that unbalanced cut problems will pose an interesting new direction of research and will enhance our understanding of graph cuts. In addition, as we have shown in this paper, unbalanced cut problems have applications in disaster and epidemics control as well as in computing small dense subgraphs and communities in graphs. Together with the problems discussed in [11,12,25], the MINSBCC problem should be considered part of a more general framework of finding *unbalanced* cuts in graphs.

This paper raises many interesting questions for future research. The main open question is how well the MINSBCC problem can be approximated in a single-criterion sense. At this time, we are not aware of any non-trivial upper or lower bounds for its approximability. The work of [11,25] implies a $(\log^2 n, 1)$ approximation — however, it approximates the capacity instead of the size, and thus cannot be used for dense subgraphs or communities. Moreover, obtaining better approximation algorithms will require using techniques different from those in this paper, since our linear program has a large integrality gap.

Further open directions involve more realistic models of the spread of diseases or disasters. The implicit assumption in our node cut approach is that each social contact will always result in an infection. If edges have infection probabilities, for instance based on the frequency or types of interaction, then the model becomes significantly more complex. We leave a more detailed analysis for future work.

Acknowledgments. We would like to thank Tanya Berger-Wolf, Venkat Guruswami, Jon Kleinberg, and Éva Tardos for useful discussions.

References

1. R. Ahuja, T. Magnanti, and J. Orlin. *Network Flows*. Prentice Hall, 1993.
2. S. Arora, S. Rao, and U. Vazirani. Expander flows, geometric embeddings and graph partitioning. In *STOC*, 2004.
3. Y. Asahiro, K. Iwama, H. Tamaki, and T. Tokuyama. Greedily finding a dense subgraph. *Journal of Algorithms*, 34, 2000.
4. N. Bailey. *The Mathematical Theory of Infectious Diseases and its Applications*. Hafner Press, 1975.
5. H. L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. *SIAM J. on Computing*, 25:1305–1317, 1996.
6. M. Develin and S. G. Hartke. Fire containment in grids of dimension three and higher, 2004. Submitted.
7. S. Eubank, H. Guclu, V.S.A. Kumar, M.V. Marathe, A. Srinivasan, Z. Toroczkai, and N. Wang. Modelling disease outbreaks in realistic urban social networks. *Nature*, 429:180–184, 2004.
8. S. Eubank, V. S. A. Kumar, M. V. Marathe, A. Srinivasan, and N. Wang. Structure of social contact networks and their impact on epidemics. *AMS-DIMACS Special Volume on Epidemiology*.
9. S. Eubank, V.S.A. Kumar, M.V. Marathe, A. Srinivasan, and N. Wang. Structural and algorithmic aspects of massive social networks. In *SODA*, 2004.
10. U. Feige, G. Kortsarz, and D. Peleg. The dense k -subgraph problem. In *STOC*, 1993.
11. U. Feige and R. Krauthgamer. A polylogarithmic approximation of the minimum bisection. *SIAM J. on Computing*, 31:1090–1118, 2002.
12. U. Feige, R. Krauthgamer, and K. Nissim. On cutting a few vertices from a graph. *Discrete Applied Mathematics*, 127:643–649, 2003.
13. U. Feige and M. Seltser. On the densest k -subgraph problem. Technical report, The Weizmann Institute, Rehovot, 1997.
14. G. Flake, S. Lawrence, C. L. Giles, and F. Coetzee. Self-organization of the web and identification of communities. *IEEE Computer*, 35, 2002.
15. G. Flake, R. Tarjan, and K. Tsioutsoulis. Graph clustering techniques based on minimum cut trees. Technical Report 2002-06, NEC, Princeton, 2002.
16. L. Ford and D. Fulkerson. Maximal flow through a network. *Can. J. Math*, 1956.
17. G. Gallo, M. D. Grigoriadis, and R. E. Tarjan. A fast parametric maximum flow algorithm and applications. *SIAM J. on Computing*, 18:30–55, 1989.
18. N. Garg, V. V. Vazirani, and M. Yannakakis. Approximate max-flow min-(multi)cut theorems and their applications. *SIAM J. on Computing*, 1996.
19. J. Kleinberg and E. Tardos. *Algorithm Design*. Addison-Wesley, 2005.
20. R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Trawling the web for emerging cyber-communities. In *WWW*, 1999.
21. E. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehard and Winston, 1976.
22. F.T. Leighton and S. Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *Journal of the ACM*, 46, 1999.
23. F. Radicchi, C. Castellano, F. Cecconi, V. Loreto, and D. Parisi. Defining and identifying communities in networks. *Proc. Natl. Acad. Sci. USA*, 2004.
24. D. Shmoys. Cut problems and their application to divide-and-conquer. In D. Hochbaum, editor, *Approximation Algorithms for NP-hard problems*, pages 192–235. PWD Publishing, 1995.
25. Z. Svitkina and E. Tardos. Min-max multiway cut. In *APPROX*, 2004.

Low Degree Connectivity in Ad-Hoc Networks

Luděk Kučera*

Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic

ludek@kam.ms.mff.cuni.cz

<http://kam.mff.cuni.cz/~ludek>

Abstract. The aim of the paper is to investigate the average case behavior of certain algorithms that are designed for connecting mobile agents in the two- or three-dimensional space. The general model is the following: let X be a set of points in the d -dimensional Euclidean space E_d , $d \geq 2$; r be a function that associates each element of $x \in X$ with a positive real number $r(x)$. A graph $G(X, r)$ is an oriented graph with the vertex set X , in which (x, y) is an edge if and only if $\rho(x, y) \leq r(x)$, where $\rho(x, y)$ denotes the Euclidean distance in the space E_d . Given a set X , the goal is to find a function r so that the graph $G(X, r)$ is strongly connected (note that the graph $G(X, r)$ need not be symmetric). Given a random set of points, the function r computed by the algorithm of the present paper is such that, for any constant δ , the average value of $r(x)^\delta$ (the average transmitter power) is almost surely constant.

1 Introduction

A motivation of a problem described in the abstract is obvious: elements of X can be mobile agents or sensors in the plane or 3D space that have transmitters and receivers; we assume uniform sensitivity of receivers and possibility of varying the power of transmitters that determines the reach of a transmitter, denoted by the function r . Due to space limitation, only most important references are presented in the present abstract, and the reader is directed e.g. to [7] for a detailed history of the problem.

It is known that the power necessary to induce the intensity of the electromagnetic field equal to the sensitivity of receivers in the distance r is proportional to r^δ for certain δ . One measure of resources used to create a connected network is the sum of powers of all transmitters of agents in X , i.e. $P(X, r) = \sum_{x \in X} r^d(x)$, and we would try to create a strongly connected graph $G(X, r)$ using a function r minimizing $P(X, r)$. It was proved [1] that the problem to decide whether for a given x and P there is a function r such that $P(X, r) \leq P$ is NP-hard for any $d \geq 2$, and, unless P=NP, there is no polynomial time approximation scheme for $d \geq 3$ (the existence of PTAS for $d = 2$ is still open).

* Work supported by European Commission - Fet Open project DELIS IST-001907 "Dynamically Evolving Large Scale Information Systems" and by the Czech Ministry of Education, Youth and Sports Project MSM0021620838.

It was also proved in [1] that if $d = 2$, $\Omega(n^\delta)$, where δ is the smallest distance of two different elements of X , is the lower bound for the problem, and there is a polynomial time algorithm that constructs a function r such that $P(X, r) = O(\Delta^2)$, where Δ is the largest distance among two elements. As a consequence, if points are within a square of side L , then there is r such that $P(X, r) = O(L^2)$, i.e. it is at most proportional to the area of the square. Similar bounds could be expected for larger dimension d . In the present paper we are going to investigate the case when X is a collection of N points distributed uniformly and independently at random in the d -dimensional cube $[0, L]^d$. In such a case it has been proved for $d = 2$ in [1] that $\Omega(L^2)$ is a lower bound for $P(X, r)$, which means that the expected behavior of the algorithm presented in [1] is asymptotically tight. However, the algorithm requires global knowledge of a configuration and information that often is not available, e.g. the coordinates of points.

In view of the nature of the main application, simple local methods are required to solve the problem. The simplest model is a *threshold* method: in the case r is constant, i.e. $r(x)$ is the same for all nodes x . There is the smallest r_0 such that the corresponding graph is connected. Unfortunately, the total power of such a network is too high, when compared to the bound derived in [1], because it is obviously

$$r_0 \geq \max_{u \in X} \min_{v \in X, v \neq u} \rho(u, v),$$

and the mean of the max-min expression on the right side, the largest nearest-neighbor distance, for a uniformly and independently distributed random set X is $\Theta(L \sqrt[d]{\log N / N})$, which means that $P(X, \mathcal{T}_{r_0}(X))$ is likely to be $\Theta(L^d \log N)$, much larger than the $O(L^d)$ bound that follows from [1].

The explication is quite simple and follows from the expected evolution of connectivity of $\mathcal{T}_r(X)$ as r increases. There are two threshold values of r . If r is smaller than the first threshold, components of the graph are likely to be small. For r between the thresholds the graph is likely to have one giant component (more than one half of nodes), but also a large number of smaller components. Finally, above the second threshold the graph is likely to be connected. The giant component threshold corresponds to $O(L^d)$ power, while the connectivity threshold corresponds to $O(L^d \log N)$ power. This means that the connectivity threshold is too large for most nodes - increasing the power of nodes in the giant component (interconnected using low power) is not necessary and only creates undesirable interference among transmitters and complicates wavelength assignment.

A similar effect can be observed for many other random situations. The Erdős-Rényi random graph Gn, p is almost surely highly disconnected for $p \ll 1/n$, has one giant component for $1 < p < \log n/n$, and it is likely to be connected for $p > \log n/n$. The gap between the giant component threshold and the connectivity threshold is also quite large.

It is well known that random D -regular graphs of Erdős-Rényi type are likely to be connected for any $D \geq 3$. Let us therefore consider the following model, called a neighborhood model, which is a geometrical analogy of a regular graph: given a random set X of points of the d -dimensional unit cube and a number

D (which might depend on the size of X), $\widehat{\mathcal{N}}_D(X)$ is a collection of all oriented pairs (x, y) of elements of X such that y is among the D nearest neighbors of x . This graph is obviously equal to $R(X, r)$, where $r(x)$ is the distance of x and its D -th nearest neighbor. Since the relation is asymmetric in general, we also define two symmetric variants: $(x, y) \in \mathcal{N}_D(X)$ if either (x, y) and/or (y, x) is in $\widehat{\mathcal{N}}_D(X)$, and $(x, y) \in \widetilde{\mathcal{N}}_D(X)$ if both (x, y) and (y, x) are in $\widehat{\mathcal{N}}_D(X)$. Obviously $\widetilde{\mathcal{N}}_D(X) \subseteq \widehat{\mathcal{N}}_D(X) \subseteq \mathcal{N}_D(X)$.

Unfortunately, it was proved in [7] that if D is smaller than $0,074 \log n$, then even $(X, \mathcal{N}_D(X))$ is likely to be disconnected, which means that the solution obtained by the algorithm is asymptotically as bad as for the threshold model.

The aim of the paper is investigate another algorithm, which is a modification of the neighborhood model and makes use of the fact that the giant component is likely to appear for small degrees of vertices of a random geometric graph. The algorithm works in rounds until a connected graph is obtained, and the i -th round is as follows:

In the i -th round, every vertex x , which is not yet in the giant component, increases the power of its transmitter to reach the i -th nearest point y of the set X , and asks y to increase power, if necessary, to be able to call back.

The idea of the algorithm is that a constant degree is sufficient to obtain a giant component; moreover, the number of nodes that are outside the giant component decreases exponentially with their degree, and therefore only a very small proportion of vertices increases their degree (the number of vertices within the reach of their transmitter) to larger values, and therefore the average transmitter power is constant.

There is one technical problem connected with the algorithm: a node x outside the giant component asks its i -th neighbor to adjust the power to be able to reach x , and the neighbor might happen to be in the giant component. In this way even vertices of the giant component increase degrees. However, we will show that the number of vertices that are not in the interior of the giant component (not sufficiently far from outside vertices) is quite small, and therefore this feature of the algorithm does not asymptotically increase the power to create a connected network.

The paper does not address the question of communication protocols necessary to implement the algorithm. Let us mention e.g. that a synchronized action of all agents is assumed in the analysis, and even though it is conjectured that an asynchronous version has essentially the same properties, this and many other problems remain open. On the other hand it is clear that the algorithm is well-adapted for distributed protocols using local information only.

2 The Algorithm

Given $x \in X$, denote by $ngbh(x, i)$ the i -th nearest neighbor of x (and we put $ngbh(x, 0) = x$); moreover, if $\in X$, then $n(X, r, x)$ will denote the size of the strong component of x in $G(X, r)$:

As mentioned above, the following algorithm is investigated in the paper

```

begin;
 $d \leftarrow 0$ ;
for all  $x \in X$  do  $r(x) = 0$ ;
while  $G(X, r)$  is not strongly connected do begin
     $d \leftarrow d + 1$ ;
    for all  $x \in X$  do compute  $n(X, r, x)$ ;
    for all  $x \in X$  do begin
         $y = \text{ngbh}(x, d)$ ;
        if  $x$  and  $y$  are in different components and  $n(X, r, x) \leq n(X, r, y)$  then begin
             $r(x) \leftarrow \rho(x, \text{ngbh}(x, d))$ ;
            if  $r(y) < r(x)$  then  $r(y) \leftarrow r(x)$ ;
        end;
    end;
end;
end.

```

The increase of $r(x)$ in the first line of the body of the **if** x and $y \dots$ statement will be called an *active* increase, while the increase of $r(y)$ is *passive*. Note that if x is in the giant component, then either y is also in the giant component or the component of y is smaller than that of x , and therefore $r(x)$ no more increases actively.

The main goal of the paper is to prove the next theorem

Theorem 1. *Given a set of N points distributed uniformly and independently at random in the cube $[0, L]^d$, where L is a fixed positive real number, then $\sum_{x \in X} r^d(x) = O(L^d)$, where r is the function computed by the above algorithm.*

3 Outline of Analysis

While the analysis of the algorithm will be given for the Euclidean space of a general dimension d , we will first explain the structure of the proof for $d = 2$ to make it easier to understand our method. A situation to be analyzed consists of N points randomly distributed in a uniform and independent way in the square $[0, L] \times [0, L]$ (the cube $[0, L]^d$ in general - the use of semi-closed intervals is purely formal and avoids intersection of subsquares used for a tessalation of the cube).

It is possible to prove that, with large probability, during the whole computation of the algorithm the degree of any vertex remains bounded by $c \log N$ and the power of any vertex is at most $c' L^d \log N / N$ for certain constants c, c' . Therefore it is sufficient to prove the $O(L^d)$ bound to the sum of powers of vertices from a certain set A that involves all but $O(N / \log N)$ vertices of X , because the sum of the power of exceptional vertices outside of A would be $O(L^d)$.

We will use a technique called de-Poissonization that will be described later, see also e.g. [5].

A great technical advantage of a Poisson process is that random events in disjoint regions of the square are independent. As already mentioned, we will

use a Poisson process of such intensity, that the expected number of generated points is N . Given an integer Q , the square is divided into $K \times K$ disjoint blocks, where K is chosen in such a way that the expected number of points generated in one block is approximately Q . The set of blocks corresponds to a square lattice $\mathcal{K} = \{0, \dots, K/1\} \times \{0, \dots, K/1\}$. Each block is in turn divided into 7×7 square cells of the same size, see Fig.1. Consequently, the expected number of points generated in each cell is approximately $Q7^{-2}$. We will say that a cell is *open* if there is at least one point generated by a Poisson process in a cell, but the number of such points is at most $2Q7^{-2}$ (i.e. about twice the expectation). A block is called *open* if all its cells are open.

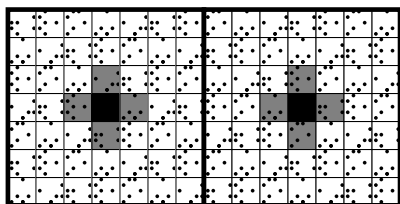


Fig. 1. Two blocks and their cells

The structure of blocks translates a Poisson process in the square area of the Euclidean space into a Bernoulli process in the lattice \mathcal{K} , which independently opens sites of the lattice with certain probability. The lattice percolation theory will be used to prove that high number of open sites of the lattice means that a very large connected cluster of open sites is likely to exist.

Let X be a set generated by a Poisson process, $D = 2Q + 1$. Define a symmetric relation \bar{R} on X as follows: given $x, y \in X$ $(x, y) \in \bar{R}$ iff y is among D nearest neighbors (with respect to the Euclidean distance) of x and in the same time x is among the D nearest neighbors of y . Note that if x belongs to an open block B , the D nearest neighbors of x must involve at least one point of X outside of B , because an open block could contain at most $2Q$ points of X . Moreover, if x is in the central cell of B (black in Fig. 1), the distance from x to any other point of the central cell or a cell that has a common face with the central cell (gray in Fig. 1) is smaller than the distance from x to any point outside of B . Therefore all points of the central cell and neighboring cells must be among the D nearest neighbors of a point of the central cell of an open block.

It follows that points in the central cell of an open block form a clique of the graph (X, \bar{R}) . Moreover, if two open blocks are neighbors, see Fig. 2, than the same argument could be applied to all 6 cells that are between the central cells of the blocks (gray in Fig. 2) to prove that the union of points in these cells form a connected subset of X in the graph (X, \bar{R}) , because each such cell is in the middle of a collection of 7×7 open cells. Finally, let us suppose that all 8 blocks that are around an open block B (4 blocks sharing a face with B and 4 blocks meeting B by corners) are open as well. Such a block B and the corresponding lattice site will be called **-open*. It is clear that in such a case *all* cells of B are

surrounded by 7×7 open cells, and therefore all points of X in the block B form a connected component of (X, \bar{R}) (cliques within the cells and all possible connections among points of two neighboring cells).

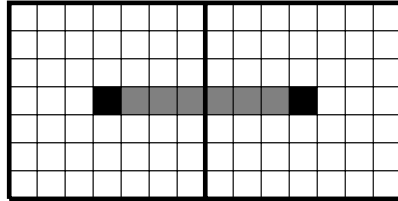


Fig. 2. Connection of the center cells of two neighboring blocks

We will adapt known results of the percolation theory to prove that if Q increases, the number of lattice sites, which are not $*$ -open sites of the largest cluster of open sites of the lattice, is very small. This does not imply directly a similar result about the size of the largest component of the graph (X, \bar{R}) . First, even if the number of closed blocks is small, such blocks could contain in general an arbitrary number of points of X , because no upper bound is imposed on a block that is not open. Therefore we will prove directly that the number of cells that contain more than $s \geq 2Q7^{-2}$ Poisson points decreases exponentially with s and therefore the number of points in such cells is not too important, because $\sum_s sc^{-s}$ is convergent for any constant $0 < c < 1$.

Moreover, any open lattice site corresponds to at least a clique of points of the central cell of the corresponding block, and if two open lattice sites are connected in the lattice, than the corresponding central cliques are connected via paths through cells between the central cells of the blocks. Hence connectivity in the lattice translates into connectivity in the geometric graph, but the component corresponding to a connected cluster of open sites of the lattice need not contain all points of the corresponding blocks. However, all points of blocks that correspond to $*$ -open sites of a cluster *do* belong to a component of the geometric graph, and therefore a connected cluster with a large number of $*$ -open sites does correspond to a large component of the graph (X, \bar{R}) .

4 Lattices

Given positive integers K and d , we denote by $\{0, \dots, K - 1\}^d$ the set of all d -tuples (i_1, \dots, i_d) , where $i_1, \dots, i_d \in \{0, \dots, K - 1\}$. If $i = (i_1, \dots, i_d)$ and $j = (j_1, \dots, j_d)$ are two d -tuples of integers (not necessarily from $\{0, \dots, K - 1\}^d$), then we write $i - j = (i_1 - j_1, \dots, i_d - j_d)$, $\|i\|_1 = |i_1| + \dots + |i_d|$ and $\|i\|_\infty = \max(i_1, \dots, i_d)$.

A *Bernoulli process* with probability p in a finite set A is a random subset B of the set A such that the probability that a given $a \in A$ belongs to B is $1 - p$ and these probabilities are independent for different elements of A . Elements of

B are called *open*, elements of $A - B$ are called *closed*. Note that p is used to denote the probability that an element is closed.

Given two sites $i, j \in \{0, \dots, K/1\}^d$, we say that they are *neighbors* (**-neighbors*, resp.), if $\|i - j\|_1 \leq 1$ ($\|i - j\|_\infty \leq 1$, resp.) Given a set $C \subseteq \{0, \dots, K - 1\}^d$, we say that C is connected if for each two $i, j \in C$ there is a sequence $i^{(0)}, \dots, i^{(k)}$ of elements of C such that $i^{(m-1)}$ and $i^{(m)}$ are neighbors for $m = 1, \dots, k$. A **-interior* of C is the set of all sites $i \in C$ such that all **-neighbors* of i are elements of C as well. A *cluster* (or a component) of a set $B \subseteq \{0, \dots, K/1\}^d$ is any connected subset $C \subseteq B$ such that c' is not connected for any $C \subset C' \subset B$.

The following theorem is a generalization of a percolation bound proved by [2] (see also [5], Theorem 9.8):

Theorem 2. *There exist $C > 0, c > 0$ and $p_0 > 0$ such that for each positive integers K, d , and a real $p, \log^{-1} K < p < p_0$, if sites of the lattice $\{0, \dots, K - 1\}^d$ are independently set to be open with probability $1 - p$, then the probability that the **-interior* of the largest open cluster has at least $K^d(1 - Cp)$ elements is at least $1 - e^{-c(K/\log K)^{d-1}}$.*

The meaning of the theorem is that, with large probability, the number of “bad” sites that are either closed or open, but in smaller clusters or outside of the **-interior* of the largest cluster is at most C times the number of closed sites, and C does not depend on other parameters (others than the dimension d).

Outline of Proof: If p is the probability that a site of the lattice is closed, the expected number of closed sites is $K^d p$ and using e.g. the Chernoff bound, it is possible to prove that it is very unlikely that there are more than $2K^d p$ closed sites. Each closed site can be a **-neighbor* of at most $3^d - 1$ open sites, and therefore all but at most $2 \cdot 3^{d+1} K^d p$ sites of the largest open cluster are in the **-interior*. It is therefore sufficient to prove the bound for the largest cluster and then to increase C by $2 \cdot 3^{d+1}$ to get the bound for the **-interior*.

The main difference of the above theorem and the result of [2] is that the latter shows that for a fixed (i.e. not dependent on K) $\varepsilon > 0$ there is a fixed but unspecified $p_0 > 0$ such that if $p < p_0$, then the size of the largest open cluster is at least $K^d(1 - \varepsilon)$ with very large probability (of the same order as in the theorem above), while we assume that p can be chosen as $\min(p_0, \varepsilon/C)$ for some constant C no matter if ε is fixed or dependent on K . The proof of the theorem is practically identical with the proof of the result of [2], as presented in [5], it is only necessary to check details of the bounds to the distribution of the size of the closed **-connected* cluster containing the origin in the infinite d -dimensional lattice with sites closed independently with probability p (and slightly modify the probability bounds). Let Y be the size of such a cluster. p should be chosen so that the two following propositions are satisfied (see [5], page 186):

Prob($Y = n$) $\leq 2^{-n}$, and
 $E(Y^{d/(d-1)}) < \delta_1 \varepsilon / 2$, where $\delta_1 = ((1 - (2/3)^{1/d})(2d))^{d/(d-1)}$.

We will show that this is satisfied if

$$p \leq p_0 = \min\left(\frac{\delta_1 \varepsilon}{2\gamma\Gamma}, \frac{1}{2\Gamma}\right), \quad \text{where} \quad \gamma = \sum_{i=0}^{\infty} (i+1)^{d/(d-1)}, \quad \Gamma = 2^{3^d}.$$

The proof of both propositions is based on Peierls argument [4] (see also [5], Lemma 9.3), that we cite in a restricted form: The number of n -element $*$ -connected subsets of the d -dimensional infinite lattice is at most $2^{3^d n} = \Gamma^n$.

Given a fixed n -element subset of the infinite lattice, and assuming that the probability that a site is closed is p , and these probabilities are independent, the probability that all its elements are closed is p^n . Together with the Peierls bound, the probability of the existence of a closed n -element set containing the origin (i.e. $\mathbf{Prob}(Y = n)$) is at most $(\Gamma p)^n \leq 2^{-n}$. Moreover

$$\begin{aligned} E(Y^{d/(d-1)}) &= \sum_{n=0}^{\infty} n^{d/(d-1)} \mathbf{Prob}(Y = n) = \sum_{n=1}^{\infty} n^{d/(d-1)} (\Gamma p)^n \leq \\ &\leq \Gamma p \sum_{i=0}^{\infty} (i+1)^{d/(d-1)} 2^{-i} = \gamma \Gamma p \leq \frac{\delta_1 \varepsilon}{2}. \end{aligned}$$

It is therefore sufficient to choose C in the theorem equal to $\delta_1/(2\gamma\Gamma)$.

The restriction $p \log K \geq 1$ is used to obtain sufficiently strong probability bound that follows from the proof given in [2]. ♣

5 Poisson Process and De-Poissonization

The technique of de-Poissonization is a very useful tool when investigating properties of a random set of N points in a rectangular or otherwise shaped subset of the Euclidean space. While very general (but also very complicated) theorems could be found e.g. in [5], it can be shown that it is quite easy to prove a result that is sufficient for purposes of the present paper. Essentially a uniform continuous Poisson process is executed with λ chosen in such a way that N is the most likely size of the set generated by the process, and the process is independently repeated $c\sqrt{N}$ times for a sufficiently large constant c , which guarantees that, with large probability, at least once is generated a set of N points. Hence, if the output of the Poisson process has certain property \mathcal{P} with probability $1 - p$, where $p = o(\sqrt{N})$, then the probability that *all* repetitions of the process have the property \mathcal{P} is at least $1 - pc\sqrt{N} = 1 - o(1)$, which gives a lower bound for the probability that a random set of N points has the property \mathcal{P} . Details will be given in a full paper.

6 The Analysis of the Algorithm

Suppose that Q is a positive real number. Choose an odd integer J such that $\sqrt{3+d} < (J-1)/2$. If $d = 2, 345$, it is sufficient to choose $(JK)^d$ subcubes of

side size ℓ , called *cells*; the cell $\mathcal{C}(i_1, \dots, i_d)$ where $i_1, \dots, i_d \in \{0, \dots, JK - 1\}$, is the collection of all points $x = (x_1, \dots, x_d)$ such that $x_k - \ell_{i_k} \in [0, \ell)$ for each $k = 1, \dots, d$. A *lattice cell* is any cell of the form $\mathcal{C}(J_{j_1} + (J - 1)/2, \dots, J_{j_d} + (J - 1)/2)$, where j_1, \dots, j_d are non-negative integers smaller than K . A distance of two cells $\mathcal{B}(i_1, \dots, i_d)$ and $\mathcal{B}(i'_1, \dots, i'_d)$ is $\sum_{k=1}^d |i_k - i'_k|$ and their $*$ -distance is $\max(|i_1 - i'_1|, \dots, |i_d - i'_d|)$.

A *block* around a cell \mathcal{C} is a union of all cells in $*$ -distance at most $(J - 1)/2$ from \mathcal{C} . \mathcal{C} is called the central cell of the block around \mathcal{C} . A *lattice block* is any block around a lattice cell. Each block around a lattice cell is clearly a hypercube formed by J^d cells, the cell \mathcal{C} being in the middle of the block. A block around a general cell, which is close to the boundary of the hypercube might contain less cells. Note that different lattice blocks are disjoint and that the hypercube $[0, L)^d$ is the union of all lattice blocks. A block around a lattice cell $\mathcal{C}(J_{j_1} + (J - 1)/2, \dots, J_{j_d} + (J - 1)/2)$ will be denoted by $\mathcal{B}(j_1, \dots, j_d)$. This notation gives a one-to-one correspondence of lattice blocks and sites of the lattice $\{0, \dots, K - 1\}^d$.

We will say that two different lattice blocks are *neighbors* ($*$ *neighbors*, resp.) if the distance ($*$ -distance, resp.) of their central cells is J . Note that a block \mathcal{B} has generally (and at most) $2d$ neighbors (that share a face with \mathcal{B}) and 2^d $*$ -neighbors (that have at least a common corner with \mathcal{B}).

Given a finite set X of points of the hypercube $[0, L)^d$, we say that a cell \mathcal{C} is open (with respect to X) if \mathcal{C} contains at least one and at most $2QJ^{-d}$ elements of X . A block is open if all its cells are open. In this way a set X also determines open sites of the lattice $\{0, \dots, K - 1\}^d$. Note that a Poisson process in the hypercube is transformed in this way to a Bernoulli process in the lattice.

Let \hat{R} be the following relation on the set X : $(x, y) \in \hat{R}$ iff y is among the D nearest neighbors of x from the set X , and define \bar{R} by $(x, y) \in \bar{R}$ iff $(x, y), (y, x) \in \hat{R}$.

Let \mathcal{C} be a cell, \mathcal{B} be a block around \mathcal{C} . Suppose that \mathcal{B} is open. Let x be an element of \mathcal{C} . Since \mathcal{B} is open, it contains at most $D - 1$ elements, and at least one point z among the D nearest neighbors of x is outside of \mathcal{B} . The distance between x and such a point z is at least $\ell(J - 1)/2$ and therefore all points of X that are at distance at most $\ell\sqrt{3 + d}$ from x are among the D nearest neighbors of x . Such points involve all points of X that belong to the cell \mathcal{C} and to any cell which is in distance 1 from \mathcal{C} . In other words, if the block around \mathcal{C} is open, points of X in \mathcal{C} form a clique of the graph (X, \bar{R}) and if $x \in \mathcal{C}$ and y is a point of X in a cell in distance 1 from \mathcal{C} , then $(x, y) \in \bar{R}$. In Fig. 1, $(x, y) \in \bar{R}$ whenever x is in a black cell and y is either in a black cell or a grey cell of the same block.

Now let us suppose that \mathcal{B}' and \mathcal{B}'' are two open lattice blocks that are neighbors. This means that their central cell $\mathcal{C}(i'_1, \dots, i'_d)$ and $\mathcal{C}(i''_1, \dots, i''_d)$ are such that there is k , $|i'_k - i''_k| = J$ and $i'_r = i''_r$ for $r \neq k$. Consider the central cells of the blocks and any cell that is between them, i.e. a cell $\mathcal{C}(i_1, \dots, i_d)$ such that $i_r = i'_r$ for $r \neq k$ and i_k is between the numbers i'_k and i''_k . It is clear that a block around any such cell is open, because it is contained in the union of the blocks \mathcal{B}' and \mathcal{B}'' . Therefore points of X in any such cell forms a clique of \bar{R} and

are \bar{R} -connected with any point in a neighboring cell of this collection, which means that points of X in those cells form a connected subset of (X, \bar{R}) .

Finally, let us suppose that \mathcal{B} is an open lattice block and all blocks that are its $*$ -neighbors are open as well. In this case a block around any cell of \mathcal{B} is open, because it is included in the union of the block \mathcal{B} and its $*$ -neighbors. This means that all elements of X that belong to the block \mathcal{B} form a connected subset, because each such point is \bar{R} -connected to any point of X in its and neighboring cells.

As a consequence, given an open connected cluster in the lattice $\{0, \dots, K - 1\}^d$, there is a component of the graph (X, \bar{R}) , which contains all points of X that are either in the central cell of a lattice block corresponding to a site of the cluster and/or in any cell of a lattice block corresponding to a site of the $*$ -interior of the cluster.

Let us now estimate the probability that a cell is open with respect to a Poisson process of intensity $\lambda = NL^{-d}$ (which guarantees that the expected size of the generated set is N). Note that the probability that a block is not open is equal to the sum of the probabilities that its cells are not open, i.e. at most J^d times more than the bound given by the following theorem.

Theorem 3. *The probability that a cell is not open with respect to a Poisson process of intensity $\lambda = NL^{-d}$ is at most $e^q + 2e(e/4)^q$, where $q = QJ^{-d}$.*

Proof. It is

$$\lambda \ell^d = N \frac{\ell^d}{L^d} = N \frac{L^d}{J^d K^d L^d} = \frac{NQ}{J^d N} = QJ^{-d} = q.$$

The probability that a cell receives m points is

$$p_m = \frac{(\lambda \ell^d)^m}{m!} e^{-\lambda \ell^d} = \frac{q^m}{m!} e^{-q}.$$

The probability that no point falls into the cell is $p_0 = e^{-q}$. If $m \geq 2q$, $2p_{m+1} \leq p_m$ and therefore $p_m + p_{m+1} + \dots \leq 2p_m$. Finally if $m = \lceil 2q \rceil$, then $m! \geq (m/e)^m$ and

$$p_m \leq \frac{q^m}{m!} e^{-q} \leq \left(\frac{qe}{m}\right)^m \leq \left(\frac{e}{2}\right)^m e^{-q} = e^{m-q} 2^{-m} \leq e^{2q+1-q} 2^{-2q} = e \left(\frac{e}{4}\right)^q.$$

Note that $q = (D - 1)/2$. Given a constant k , there is $c > 0$ such that if $D \geq c \log N$, then $e^{-q} + 2e(e/4)^q = O(N^{-(1+k)})$ and therefore the expected number of closed cells is $O(N^{-k})$. Using Markov inequality, the probability that there is a closed cell is $O(N^{-k})$ as well. In such a case the relation \bar{R} is connected with high probability, and the same is true for the result of binomial process (with slightly smaller probability), using the de-Poissonization theorem.

Theorem 4 makes it also possible to estimate the size of the largest component of the geometric random graph $(X, \mathcal{N}_D(X))$ for a random set X of N points of the cube $[0, L]^d$.

Theorem 4. *There exist constants $c > 1$ and $q > 0$ such that, with probability $1 - o(1)$, if $c^D < q \log N$, then the size of the largest component of $(X, \mathcal{N}_D(X))$, where X is a set of N points distributed uniformly and independently at random in the cube $[0, L]^d$, is at least $N(1 - O(\kappa^D))$, where $\kappa = \sqrt{e/4}$.*

Outline of Proof: In view of Theorem 3, it is sufficient to prove the theorem with the probability bound $1 - o(N^{-1/2})$ for the case when X is the result of a Poisson process in $[0, L]^d$ with intensity $\lambda = NL^{-d}$. In this proof we will not exactly compute probabilities of events, but in all cases such probabilities could be bounded by Chernoff bound that is strong enough for purposes of the proof.

Choose Q so that $2J^dQ + 1 = D$, denote $q = QJ^{-d}$. Note that $q = (D - 1)/2$. Consider a partition into K^d blocks as defined in the beginning of this section. The probability that a block is not open is $O(\kappa^D)$ in view of Theorem 4. If $\kappa^D > \log^{-1} K$, it follows from Theorem 2 that at most $K^d C \kappa^D$ lattice sites are open sites that are not in the $*$ -interior of the largest cluster, and the corresponding open blocks contain at most $2J^d Q K^D C \kappa^D = 2J^d N C \kappa^D$ elements of X . As mentioned above, all remaining points of open blocks belong to one large connected component of the graph. The expected number of closed blocks is at most $K^d \kappa^D \geq N^d / \log K \geq N^D / \log N$, and therefore the Chernoff bound shows that, with very large probability, the number of closed blocks is less than $2K^d \kappa^D$. The number of open cells in closed blocks is therefore bounded by $2J^d K^d \kappa^D$ and the number of nodes of the graph in such open cells is therefore at most $4QK^d \kappa^D = 4N\kappa^D$. The number of nodes in closed cells cannot be bounded by a direct use of the same method, as there is no upper bound to the number of graph nodes in such a cell.

However, the probability p_k that a given cell contains $k \geq 2QJ^{-d}$ elements of X is $q^k e^{-q}/k!$, the expected number of such cells is $(KJ)^d p_k$, and either $k(KJ)^d p_k$ is very small or the Chernoff bound shows that the actual number of such cells is very likely to be at most $2(KJ)^d p_k$ and therefore there are at most $2k(KJ)^d p_k$ nodes of the graph in such cells. If $k \geq 2q$, then

$$\frac{(k + 1)p_{k+1}}{kp_k} = \frac{k + 1}{k} \frac{q}{k + 1} \leq \frac{4}{3} = \frac{2}{3},$$

and therefore the number of nodes of the graph in all closed cells is very likely to be at most

$$\sum_{k > 2q} 2k(KJ)^d p^k = 2(KJ)^d p_{2q} \frac{1}{1 - 2/3} = 6(KJ)^d p_{2q} = 6 \frac{N}{q} p_{2q} \leq 6N\kappa^q,$$

which proves the theorem. ♣

Theorem 5. *Given $x \in X$, let $D(x)$ be the smallest D such that x belongs to a giant component of $(X, \mathcal{N}_D(X))$, $\ell(x)$ be the distance of x and its $D(x)$ -th nearest neighbor. There is a constant c such that the expectation of the sum $\sum_{x \in X} (D(x) + 1)\ell^d(x)$ is at most $c^L d$.*

The proof of the theorem follows easily from the previous theorem that bounds the number of points outside of the largest component of $(X, \mathcal{N}_D(X))$ for a given

D , but its proof is not given in this abstract. Now, let us suppose that $(x, y) \in \mathcal{N}_D(X)$. If x and y are in the same component of $\bar{\mathcal{N}}_{D-1}(X)$, then (x, y) need not be added to the graph, because it would not change the structure of connected components. Let x and y belong to different components, and assume without loss of generality that the component of x is not larger than the component of y . Then the algorithm increases $r(x)$ to involve y among neighbours of x and, if necessary, increases $r(y)$ to add the edge (y, x) as well. Thus, the structure of components of the graph created by the algorithm remains at least as coarse as the structure of components of $\bar{\mathcal{N}}_{D-1}(X)$. Note that the term $(D(x) + 1)$ in the sum is used to cope with the fact that a passive increase of $r(y)$ is charged to the account of x . All this proves Theorem 1.

7 Experiments

Experiments performed on different uniform random subsets of a square confirmed what was already observed many years ago: six [3] or eight [6] is a magic number of neighbors that are sufficient to obtain a connected network in practically all instances. Theoretical results are mostly of asymptotic nature, but it seems from simulations that many millions of nodes would be necessary to get a non-negligible probability of generating a single small isolated cluster in a network where each node connects to 6 nearest neighbors. However, it is also clear that connecting to 4 or 5 nearest neighbors is sufficient to obtain a giant component that covers most vertices, and therefore the algorithm presented in Section 2 would use smaller average degree of a network than could be obtained by constant degree algorithm. Complete results of experiments will be presented in the full version of the paper.

References

1. Clementi, A., Penna, P., Silvestri, R., On the Power Assignment Problem in Radio Networks, *Mobile Networks and Applications*, **9** (2004), 125-140.
2. Deuschel, J.-D., Pisztora, A., Surface order large deviations fo high-density percolation, *Probability Theory and Related Fields*, **104** (1996), 467-482.
3. Kleinrock, L., Silvester, J.A., Optimum transmission radii for packet radio networks or why six is a magic number, *IEEE Nat. Telecommun. Conf.*, 1978, pp. 4.3.1-4.3.5.
4. Peierls, R., On Ising's model of ferromagnetism, *Proceedings of the Cambridge Philosophical Society*, **36**, 477-481.
5. Penrose M., *Random Geometric Graphs*, Oxford University Press, 2003.
6. Takagi, H., Kleinrock, L., Optimal transmission ranges for randomly distributed packet radioterminals, *IEEE Trans. Commun.*, COM-32 (1984), 246-257.
7. Xue, F., Kumar, P.R., The number of neighbors needed for connectivity of wireless networks, *Wireless Networks*, **10** (2004), 169-181.

5-Regular Graphs are 3-Colorable with Positive Probability*

J. Díaz¹, G. Grammatikopoulos^{2,3}, A.C. Kaporis², L.M. Kirousis^{2,3}, X. Pérez¹,
and D.G. Sotiropoulos⁴

¹ Universitat Politècnica de Catalunya, Departament de Llenguatges i Sistemes
Informàtics, Campus Nord – Ed. Omega, 240, Jordi Girona Salgado,
1–3, E-08034 Barcelona, Catalunya
{diaz, xperez}@lsi.upc.edu

² University of Patras, Department of Computer Engineering and Informatics,
GR-265 04 Patras, Greece
{grammat, kaporis, kirousis}@ceid.upatras.gr

³ Research Academic Computer Technology Institute,
P.O. Box 1122, GR-261 10 Patras, Greece

⁴ University of Patras, Department of Mathematics, GR-265 04 Patras, Greece
dgs@math.upatras.gr

Abstract. We show that uniformly random 5-regular graphs of n vertices are 3-colorable with probability that is positive independently of n .

1 Introduction

The problem of finding the chromatic number of a graph has been a cornerstone in the field of discrete mathematics and theoretical computer science. Recall that the chromatic number of a graph is the minimum number of colors needed to legally color the vertices of the graph, where a coloring is *legal* if no two adjacent vertices share the same color. The problem is trivial for two colors but became difficult for three or more colors. Due to the difficulty of the problem, a large effort has been made in looking into structural properties of the problem, in the hope of finding more efficient procedures which apply to larger families of graphs.

An active line of research has been the characterization of classes of graphs due to their chromatic number. In particular, intense effort has been devoted to

* The 1st, 2nd, 4th and 5th authors are partially supported by Future and Emerging Technologies programme of the EU under contract 001907 “*Dynamically Evolving, Large-Scale Information Systems (DELIS)*”. The 1st author was partially supported by the Distinció de la Generalitat de Catalunya per a la promoció de la recerca, 2002. The 3rd and 4th authors are partially supported by European Social Fund (ESF), Operational Program for Educational and Vocational Training II (*EPEAEK II*), and particularly *Pythagoras*. Part of the research of the 4th author was conducted while visiting on a sabbatical the Departament de Llenguatges i Sistemes Informàtics of the Universitat Politècnica de Catalunya.

the study of the chromatic number on random structures. One active approach consists in finding the threshold p_k of k -colorability for the binomial model G_{n,p_k} . An early result in this line is presented in [10], where it is shown that, for any $d \in \mathbb{N}$, $d > 0$, $\exists k \in \mathbb{N}$ such that *asymptotically almost surely* (a.a.s.) the chromatic number of $G_{n,d/n}$ is either k or $k + 1$. Recall that a sequence of events \mathcal{E}_n holds a.a.s. if $\lim_{n \rightarrow \infty} \Pr[\mathcal{E}_n] = 1$. Further results on the particular threshold p_3 for 3-colorability can be found in the introduction of [2].

A related approach is to consider the chromatic number of random d -regular graphs (i.e., graphs for which every vertex has degree d). For a comprehensive review on random regular graphs, the reader is referred to [14].

For $d \geq 6$, a simple counting argument shows that d -regular graphs are not 3-colorable, a.a.s. In fact, Molloy and Reed [12] proved that 6-regular graphs have chromatic number at least 4, a.a.s. Achlioptas and Moore [2] proved that 4-regular graphs have chromatic number 3 *with uniform positive probability* (w.u.p.p.), where a sequence of events \mathcal{E}_n holds w.u.p.p. if $\liminf_{n \rightarrow \infty} \Pr[\mathcal{E}_n] > 0$. The proof was algorithmic in the sense that a backtracking-free algorithm based on Brelaz' heuristic was designed and shown to produce a 3-coloring w.u.p.p. Subsequently, Achlioptas and Moore [3] showed that a.a.s. the chromatic number of a d -regular graph ($d \geq 3$) is k or $k + 1$ or $k + 2$, where k is the smallest integer such that $d < 2k \log k$. They also showed that if furthermore $d > (2k - 1) \log k$, then a.a.s. the chromatic number is either $k + 1$ or $k + 2$. This result however gives no information for the chromatic number of either 4-regular or 5-regular graphs, apart from the known fact that a.a.s. the former have chromatic number either 3 or 4 and, the latter either 3 or 4 or 5. Shi and Wormald [13] showed that a.a.s. the chromatic number of a 4-regular graph is 3, that a.a.s. the chromatic number of a 6-regular graph is 4 and that a.a.s. the chromatic number of a 5-regular graph is either 3 or 4. They also showed that a.a.s. the chromatic number of a d -regular graph, for all other d up to 10, is restricted to a range of two integers.

The above results leave open the question of whether the chromatic number of a 5-regular graph can take the value 3 w.u.p.p., or perhaps even a.a.s.

On the other hand, building on a statistical mechanics analysis of the space of truth assignments of the 3-SAT problem, which has not been shown yet to be mathematically rigorous, and Survey Propagation (SP) algorithm for 3-SAT inspired by this analysis (see e.g. [11] and the references therein), Krzakała et al. [9] provided strong evidence that 5-regular graphs are a.a.s. 3-colorable by a SP algorithm. They also showed that the space of assignments of three colors to the vertices (legal or not) consists of clusters of *legal* color assignments inside of which one can move from point to point by steps of small Hamming distance. However, to go from one cluster to another by such small steps, it is necessary to go through assignments of colors that grossly violate the requirement of legality. Moreover, the number of clusters that contain points with energy that is a local, but not global, minimum is exponentially large. As a result, local search algorithms are easily trapped into such local minima. These considerations left as the only

plausible alternative to try to prove that 5-regular graphs are 3-colorable w.u.p.p. in an analytic way.

In this paper, we solve in the positive the question, showing that 5-regular graphs are 3-colorable w.u.p.p. The technique used is the Second Moment Method: Let X be a non-negative random variable (r.v.), then

$$\Pr[X > 0] \geq \frac{(\mathbf{E}(X))^2}{\mathbf{E}(X^2)}. \quad (1)$$

Thus, if this ratio is $\Theta(1)$, then we have that $X > 0$ w.u.p.p. This well known technique (see Remark 3.1 in [8]) was used by Achiloptas and Naor [4] to solve the long standing open problem of computing the two possible values of the chromatic number of a random graph.

To apply the Second Moment Method, we work under the *configuration model* (see [6]) and consider the r.v. X that counts a special type of colorings, which we call the *stable balanced 3-colorings*. In Section 2, we give exact expressions for the first and second moments of X , as sums of terms where each term consists of the product of polynomial (in n), and exponential terms. In Section 3 we compute the asymptotic values of those exact expressions, which turned out to be a non-trivial task. We show that $E(X^2) = O((E(X))^2)$, so that the probability in (1) is uniformly positive. Finally, the result is transferred from configurations to random 5-regular graphs.

An important remaining open question that we are working on is the extension of our result to a.a.s. It is plausible that an affirmative answer to the question can be obtained by concentration results similar to those in [3].

2 Exact Expressions for the First and Second Moments

Everywhere in this paper n will denote a positive integer divisible by 6. Asymptotics are always in terms of n .

In the sense of the configuration model (see, e.g., [6] or [14]), let $\mathcal{C}_{n,5}$ be the probability space of 5-regular *configurations*, obtained by considering a set of n vertices, labelled $1, \dots, n$, for each of these vertices a set of 5 *semi-edges*, labelled $1, \dots, 5$, and a uniform random perfect matching of all $5n$ semi-edges. Each pair of semi-edges according to the matching defines one edge of the configuration.

Definition 1. A 3-coloring of a configuration $G \in \mathcal{C}_{n,5}$ is called *stable* if for every vertex v and every color $i = 0, 1, 2$, either v itself or one of its neighbors are colored by i . Equivalently, for no single vertex v can we change its color without the appearance of an edge with the same color at its endpoints. A 3-coloring is called *balanced* if for each $i = 0, 1, 2$, the number of vertices with color i is $n/3$.

Given a configuration $G \in \mathcal{C}_{n,5}$, let \mathcal{S}_G be the class of balanced stable 3-colorings of G . Let X be the random variable that counts the number of balanced stable 3-colorings of $\mathcal{C}_{n,5}$. Then, the following equations can be easily shown:

$$|\{G \mid G \in \mathcal{C}_{n,5}\}| = \frac{(5n)!}{2^{5n/2}(5n/2)!}, \quad \mathbf{E}(X) = \frac{|\{(G, C) \mid G \in \mathcal{C}_{n,5}, C \in \mathcal{S}_G\}|}{|\{G \mid G \in \mathcal{C}_{n,5}\}|}, \quad (2)$$

$$\mathbf{E}(X^2) = \frac{|\{(G, C_1, C_2) \mid G \in \mathcal{C}_{n,5}, C_1, C_2 \in \mathcal{S}_G\}|}{|\{G \mid G \in \mathcal{C}_{n,5}\}|}. \tag{3}$$

2.1 First Moment

Below we assume we are given a configuration G and a balanced stable 3-coloring C on G . The arithmetic in the indices is modulo 3. We start by giving some useful terminology and notation:

Definition 2. A 1-spectrum s is an ordered pair of non-negative integers, $s = (s_{-1}, s_1)$, such that $s_{-1} + s_1 = 5$ and $s_{-1}, s_1 > 0$.

Notice that there are four 1-spectra. They are intended to express the distribution of the five edges stemming from a given vertex according to the color of their other endpoint. Formally, a vertex v of color i is said to have 1-spectrum $s = (s_{-1}, s_1)$, if s_{-1} out of its five edges are incident on vertices of color $i - 1$ and the remaining s_1 edges are incident on vertices of color $i + 1$. The condition $s_{-1}, s_1 > 0$ expresses the fact that the 3-coloring is a stable one.

For each $i = 0, 1, 2$ and 1-spectrum s , we denote by $d(i; s)$ the scaled (with respect to n) number of vertices of G which are colored by i and have 1-spectrum s . Then, $\sum_s d(i; s) = 1/3$ and therefore $\sum_{i,s} d(i; s) = 1$.

Let $N_1 = |\{(G, C) \mid G \in \mathcal{C}_{n,5}, C \in \mathcal{S}_G\}|$. Given any two colors i and j , observe that there are exactly $5n/6$ edges connecting vertices with color i and j , respectively.

Given a fixed sequence $(d(i; s)n)_{i,s}$ that corresponds to a balanced stable 3-coloring, let us denote by $\binom{n}{(d(i; s)n)_{i,s}}$ the multinomial coefficient that counts the number of ways to distribute the n vertices into classes of cardinality $d(i; s)n$ for all possible values of i and s . Let also $\binom{5}{s}$ stand for $\binom{5}{s_{-1}} = \binom{5}{s_1}$.

By a counting argument, we have that

$$N_1 = \sum_{d(i; s)_{i,s}} \binom{n}{(d(i; s)n)_{i,s}} \left(\prod_{i,s} \binom{5}{s}^{d(i; s)n} \right) \left(\frac{5n!}{6} \right)^3, \tag{4}$$

where the summation above is over all possible sequences $(d(i; s))_{i,s}$ that correspond to balanced stable 3-colorings.

2.2 Second Moment

Below we assume we are given a configuration G and two balanced stable 3-colorings C_1 and C_2 on G . For $i, j = 0, 1, 2$, let V_i^j be the set of vertices colored with i and j with respect to colorings C_1 and C_2 , respectively. Let $n_i^j = |V_i^j|/n$, and let E_i^j be the set of semi-edges whose starting vertex is in V_i^j . Also, for $r, t \in \{-1, 1\}$, let $E_{i,r}^{j,t}$ be the set of semi-edges in E_i^j which are matched with one in E_{i+r}^{j+t} . Let $m_{i,r}^{j,t} = |E_{i,r}^{j,t}|/n$. We have that $\sum_{r,t} m_{i,r}^{j,t} = 5n_i^j$, $\sum_{i,j} n_i^j = 1$, and therefore $\sum_{i,j,r,t} m_{i,r}^{j,t} = 5$. And, since matching sets of semi-edges should have equal cardinalities, we also have that $m_{i,r}^{j,t} = m_{i+r,-r}^{j+t,-t}$.

Definition 3. A 2-spectrum s is an ordered quadruple of non-negative integers, $s = (s_{-1}^{-1}, s_{-1}^1, s_1^{-1}, s_1^1)$, such that $s_{-1}^{-1} + s_{-1}^1 + s_1^{-1} + s_1^1 = 5$ and $(s_{-1}^{-1} + s_{-1}^1)(s_1^{-1} + s_1^1)(s_{-1}^{-1} + s_1^{-1})(s_{-1}^1 + s_1^1) > 0$.

Notice that the number of 2-spectra is 36. Let v be a vertex in V_i^j . Vertex v is said to have 2-spectrum $s = (s_{-1}^{-1}, s_{-1}^1, s_1^{-1}, s_1^1)$ if s_r^t out of its five edges, $r, t \in \{-1, 1\}$, are incident on vertices in V_{i+r}^{j+t} . The condition $(s_{-1}^{-1} + s_{-1}^1)(s_1^{-1} + s_1^1)(s_{-1}^{-1} + s_1^{-1})(s_{-1}^1 + s_1^1) > 0$ expresses the fact that both C_1 and C_2 are stable.

For each $i, j = 0, 1, 2$ and 2-spectrum s , we denote by $d(i, j; s)$ the scaled number of vertices which belong to V_i^j and have 2-spectrum s . We have:

$$\sum_s s_r^t d(i, j; s) = m_{i,r}^{j,t}, \quad \sum_s d(i, j; s) = n_i^j \quad \text{and therefore} \quad \sum_{i,j,s} d(i, j; s) = 1.$$

Throughout this paper we refer to the set of the nine numbers n_i^j as the set of the *overlap matrix variables*. We also refer to the set of the thirty-six numbers $m_{i,r}^{j,t}$ as the set of the *matching variables*. Finally, we refer to the 9×36 numbers $d(i, j; s)$ as the *spectral variables*.

Let $N_2 = |\{(G, C_1, C_2) \mid G \in \mathcal{C}_{n,5}, C_1, C_2 \in \mathcal{S}_G\}|$. Given a fixed sequence $(d(i, j; s)n)_{i,j,s}$ that corresponds to a pair of balanced stable 3-colorings, let us denote by $\binom{n}{(d(i,j;s)n)_{i,j,s}}$ the multinomial coefficient that counts the number of ways to distribute the n vertices into classes of cardinality $d(i, j; s)n$ for all possible values of i, j and s . Let also $\binom{5}{s}$ stand for $\binom{5}{s_{-1}^{-1}, s_{-1}^1, s_1^{-1}, s_1^1}$ (the distinction from a similar notation for 1-spectra will be obvious from the context). Now, by an easy counting argument, we have:

$$N_2 = \sum_{d(i,j;s)_{i,j,s}} \left\{ \binom{n}{(d(i,j;s)n)_{i,j,s}} \left(\prod_{i,j,s} \binom{5}{s}^{d(i,j;s)n} \right) \left(\prod_{i,j,r,t} ((m_{i,r}^{j,t}n)!)^{\frac{1}{2}} \right) \right\}, \quad (5)$$

where the summation above is over all possible sequences $(d(i, j; s))_{i,j,s}$ that correspond to pairs of balanced stable 3-colorings.

3 Asymptotics

In this section we will show that $E(X^2) = O((E(X))^2)$. An immediate consequence of this is that 5-regular configurations have a balanced stable 3-coloring and hence a generic 3-coloring w.u.p.p.

By applying Stirling approximation to formulae (2), (4) and (5), we get:

$$\mathbf{E}(X) \sim \sum_{d(i;s)_{i,s}} f_1(n, d(i; s)_{i,s}) \left(6^{-\frac{5}{2}} \prod_{i,s} \left(\frac{\binom{5}{s}}{d(i; s)} \right)^{d(i; s)} \right)^n, \quad (6)$$

$$\mathbf{E}(X^2) \sim \sum_{d(i,j;s)_{i,j,s}} f_2(n, d(i, j; s)_{i,j,s}) \left[5^{-\frac{5}{2}} \prod_{i,j,s} \left(\frac{\binom{5}{s}}{d(i, j; s)} \right)^{d(i,j;s)} \left(\prod_{i,j,r,t} (m_{i,r}^{j,t})^{\frac{1}{2}} m_{i,r}^{j,t} \right) \right]^n, \quad (7)$$

where f_1 and f_2 are functions that are sub-exponential in n and also depend on the sequences $d(i; s)_{i,s}$ and $d(i, j; s)_{i,j,s}$, respectively. By a result similar to Lemma 3 in [3], by using a standard Laplace-type integration technique, we can prove that the Moment Ratio is asymptotically positive if

$$(E(X))^2 \asymp E(X^2), \text{ i.e. } \ln((E(X))^2) \sim \ln(E(X^2)). \tag{8}$$

Let M_1 be the maximum base $6^{-5/2} \prod_{i,s} \left(\frac{\binom{5}{s}}{d(i;s)}\right)^{d(i;s)}$ as $d(i; s)_{i,s}$ ranges over all possible sequences that correspond to balanced stable 3-colorings. Analogously, let M_2 be the maximum base

$$5^{-5/2} \left(\prod_{i,j,s} \left(\frac{\binom{5}{s}}{d(i,j;s)}\right)^{d(i,j;s)} \right) \left(\prod_{i,j,r,t} (m_{i,r}^{j,t})^{\frac{1}{2}m_{i,r}^{j,t}} \right).$$

From the equations (6) and (7) one can immediately deduce that the relation (8) is true if $(M_1)^2 = M_2$. We need to compute the exact values of M_1 and M_2 .

3.1 First Moment: Computing M_1

Let $f = \prod_{i,s} \left(\frac{\binom{5}{s}}{d(i;s)}\right)^{d(i;s)}$ be a real function of 12 non-negative real variables $d(i; s)$ defined over the polytope $\sum_s d(i; s) = 1/3$, where $i = 0, 1, 2$ and s runs over 1-spectra. The following lemma follows from the application of elementary analysis techniques and the computation of Lagrange multipliers.

Lemma 1. *The function $\ln f$ is strictly convex. Let $\mathcal{D}_1 = \sum_{i,s} \binom{5}{s} = 3 \times 30$. The function f has a maximizer at the point where $d(i; s) = \frac{\binom{5}{s}}{\mathcal{D}_1}, \forall i, s$.*

By direct substitution, we obtain:

Lemma 2. $M_1 = 6^{-5/2} \left(\prod_{i,s} \mathcal{D}_1^{d(i,s)}\right) = \left(\frac{1}{6}\right)^{5/2} \mathcal{D}_1 = \sqrt{\frac{25}{24}}.$

From the above and from (6), we get:

Theorem 1. *The expected number of balanced stable 3-colorings of a random 5-regular configuration approaches infinity as n grows large.*

3.2 Second Moment: Computing M_2

Let
$$F = \left(\prod_{i,j,s} \left(\frac{\binom{5}{s}}{d(i,j;s)}\right)^{d(i,j;s)} \right) \left(\prod_{i,j,r,t} (m_{i,r}^{j,t})^{\frac{1}{2}m_{i,r}^{j,t}} \right) \tag{9}$$

be a real function of non-negative real variables $d(i, j; s)$ (where $i, j = 0, 1, 2, r, t = -1, 1$ and s runs over 2-spectra) defined over the polytope determined by:

$$\sum_{j,s} d(i, j; s) = 1/3, \forall i; \sum_{i,s} d(i, j; s) = 1/3, \forall j \text{ and } m_{i,r}^{j,t} = m_{i+r,-r}^{j+t,-t}, \tag{10}$$

where $m_{i,r}^{j,t} = \sum_s s_r^t d(i, j; s)$. Notice that F is a function of 9×36 variables.

We will maximize F in three phases. In the first one, we will maximize F assuming the matching variables $m_{i,r}^{j,t}$ are fixed constants such that their values are compatible with the polytope over which F is defined. Thus we will get a function F_m of the 36 matching variables $m_{i,r}^{j,t}$. At the second phase we will maximize F_m assuming that the nine overlap matrix variables $5n_i^j = \sum_{r,t} m_{i,r}^{j,t}$ are fixed constants compatible with the domain of the matching variables. Thus we will get a function F_n of the overlap matrix variables. The preceding two maximizations will be done in an analytically exact way. Observe that since we consider balanced 3-colorings, F_n depends only on the values of four n's. We will maximize F_n by going through its 4-dimensional domain over a fine grid. Let us point out that the maximizations above will not be done *ex nihilo*. Actually, we know (see below) the point where we would like the maximizer to occur. Therefore all we do is not find the maximizer but rather prove that it is where we want it to be. The proof of the next lemma is done by direct substitution.

Lemma 3. *Let $\mathcal{D}_2 = \sum_{i,j,s} \binom{5}{s} = 9 \times 900$ and let $d(i, j; s) = \binom{5}{s} / \mathcal{D}_2, \forall i, j, s$. Then the value of the base $5^{-5/2} F$ at the above values of $d(i, j; s)_{i,j,s}$ is equal to $(M_1)^2 = 25/24$.*

We call the sequence $d(i, j; s) = \binom{5}{s} / \mathcal{D}_2$ the *barycenter*. Barycenter as well we call the corresponding point in the domain of F_m , i.e. the point $m_{i,r}^{j,t} = 5/36, \forall i, j, r, t$. Finally, barycenter also we call the corresponding point in the domain of F_n , i.e. the point $n_i^j = 1/9, \forall i, j$. We will see, by direct substitutions, that the functions $5^{-5/2} F_m$ and $5^{-5/2} F_n$ as well take the value $(M_1)^2 = 25/24$ at their corresponding barycenters. Therefore, after computing F_m and F_n , all that will remain to be proved is that F_n has a maximizer at its barycenter $n_i^j = 1/9, i, j = 0, 1, 2$. Below, we compute the functions F_m and F_n and then we show that the barycenter is a maximizer for F_n by sweeping its 4-dimensional domain.

From the Spectral to the Matching Variables. Everywhere below we assume that the 36 matching variables $m_{i,r}^{j,t}$ are non-negative and moreover take only values for which there exist 9×36 spectral non-negative variables $d(i, j, s)$ such that

$$m_{i,r}^{j,t} = \sum_s s_r^t d(i, j; s), \quad i, j = 0, 1, 2, \quad r, t = -1, +1. \tag{11}$$

and such that the equations in (10) hold. It is not hard to see that the above restrictions on the matching variables are equivalent to assuming that $\forall i, j = 0, 1, 2$ and $\forall r, t = -1, 1$,

$$m_{i,r}^{j,t} = m_{i+r,-r}^{j+t,-t}, \quad \sum_{i,r,t} m_{i,r}^{j,t} = 5/3, \quad \sum_{j,r,t} m_{i,r}^{j,t} = 5/3 \quad \text{and} \tag{12}$$

$$m_{i,r}^{j,t} \geq 0, \quad m_{i,r}^{j,t} + m_{i,r}^{j,-t} \leq 4(m_{i,-r}^{j,t} + m_{i,-r}^{j,-t}), \quad m_{i,r}^{j,t} + m_{i,-r}^{j,t} \leq 4(m_{i,r}^{j,-t} + m_{i,-r}^{j,-t}). \tag{13}$$

Fix such values for the $m_{i,r}^{j,t}$. To maximize the function F given by equation (9) over the polytope described in (10) for the fixed values of the matching variables $m_{i,r}^{j,t}, i, j = 0, 1, 2, r, t = \{-1, 1\}$, it is sufficient to maximize the function F subject to the 36 constraints in (11). We call this maximum F_m .

Since for different pairs of (i, j) , $i, j = 0, 1, 2$, neither the variables $d(i, j; s)$ nor the constraints in (11) have anything in common, and since the matching variables are fixed, it is necessary and sufficient to maximize separately for each $i, j = 0, 1, 2$ the function $F_{i,j} = \prod_s \binom{5}{s} / d(i, j; s)^{d(i,j;s)}$, subject to the four constraints: $\sum_s s_r^t d(i, j; s) = m_{i,r}^{j,t}$, $r, t = -1, 1$. We will use Lagrange multipliers to maximize the logarithm of these functions. Notice that the functions $\ln F_{i,j}$ are strictly convex. We define the following function:

$$\Phi(x, y, z, w) = (x + y + z + w)^5 - (x + y)^5 - (x + z)^5 - (y + w)^5 - (z + w)^5 + x^5 + y^5 + z^5 + w^5.$$

Also for each of the nine possible pairs (i, j) , $i, j = 0, 1, 2$, consider the 4×4 system:

$$\frac{\partial \Phi(\mu_{i,-1}^{j,-1}, \mu_{i,-1}^{j,1}, \mu_{i,1}^{j,-1}, \mu_{i,1}^{j,1})}{\partial \mu_{i,r}^{j,t}} \mu_{i,r}^{j,t} = m_{i,r}^{j,t}, \quad r, t = -1, 1, \tag{14}$$

where $\mu_{i,r}^{j,t}$ denote the 36 unknowns of these nine 4×4 systems. Applying the method of the Lagrange multipliers, we get

Lemma 4. *Each of the nine systems in (14) has a unique solution. Moreover in terms of the solutions of these systems*

$$F_m = \prod_{i,j,r,t} \left(\frac{(m_{i,r}^{j,t})^{\frac{1}{2}}}{\mu_{i,r}^{j,t}} \right)^{m_{i,r}^{j,t}}.$$

By the above Lemma, we have computed in an analytically exact way the function F_m . Notice that the function F_m is a function of the 36 matching variables $m_{i,r}^{j,t}$, $i, j = 0, 1, 2$ and $r, t = -1, 1$, over the domain given by (12) and (13). However its value is given through the solutions of the systems in (14), which have a unique solution.

From the Matching to the Overlap Matrix Variables. We assume now that we fix nine non-negative overlap matrix variables n_i^j such that $\sum_i n_i^j = 1/3$, $\forall j$ and $\sum_j n_i^j = 1/3$, $\forall i$. Using again multiple Lagrange multipliers, we will find the maximum, call it F_n , of the function F_m given in Lemma 4 under the constraints:

$$\sum_{r,t} m_{i,r}^{j,t} = 5n_i^j, \quad \text{and} \quad m_{i,r}^{j,t} = m_{i+r,-r}^{j+t,-t}, \quad \text{for } i, j = 0, 1, 2 \quad \text{and } r, t = \{-1, 1\}. \tag{15}$$

assuming in addition that the $m_{i,r}^{j,t}$ satisfy the inequalities in (13). We consider the latter inequality restrictions not as constraints to be dealt with Lagrange multipliers, but as restrictions of the domain of F_m that must be satisfied by the maximizer to be found.

We will need that the function $\ln F_m$ over the polytope determined by the constraints (15) (for fixed values of the variables n_i^j) is strictly convex. To show this it is sufficient to fix an arbitrary $i, j = 0, 1, 2$ and show that the 4-variable function $\ln \left(\prod_{r,t} \left((m_{i,r}^{j,t})^{\frac{1}{2}} / \mu_{i,r}^{j,t} \right)^{m_{i,r}^{j,t}} \right)$, subject to the single linear constraint

$\sum_{r,t} m_{i,r}^{j,t} = n_i^j$, is strictly convex. To show the latter, we computed the Hessian and its LPMD's after solving the single linear constraint for one of its variables (thus we obtained a function of three variables). Notice that the value of the function under examination is given through the unique solutions of a 4×4 system. The Hessian and the LPMD's were analytically computed in terms of these solutions by implicit differentiation of the equations of the system. The strict convexity then would follow if we showed that at every point of the domain of this 3-variable function, the LPMD's were non-zero and of alternating sign. We demonstrated this by going over this domain over a fine grid and computing at all its points the LPMD's. The values of the LPMD's that we got were safely away from zero and with the desired algebraic sign. Notice that although to prove the convexity of the function $\ln F_m$, subject to the constraints in (15), we relaxed the constraints $m_{i,r}^{j,t} = m_{i+r,-r}^{j+t,-t}$, the latter ones are essential for correctly computing F_n .

To apply Lagrange multipliers, we have to find the partial derivatives of the function $\ln F_m = \sum_{i,j,r,t} (\frac{1}{2} m_{i,r}^{j,t} \ln m_{i,r}^{j,t} - m_{i,r}^{j,t} \ln \mu_{i,r}^{j,t})$. In fact, after a few manipulations, we obtain:

Lemma 5.
$$\sum_{r',t'} m_{i,r'}^{j,t'} \frac{\partial \ln \mu_{i,r'}^{j,t'}}{\partial m_{i,r}^{j,t}} = \frac{1}{5}, \text{ and thus } \frac{\partial \ln F_m}{\partial m_{i,r}^{j,t}} = \frac{3}{10} + \frac{1}{2} \ln m_{i,r}^{j,t} - \ln \mu_{i,r}^{j,t}$$

By applying now the technique of multiple Lagrange multipliers, we get:

Lemma 6. *Consider the 45×45 system with unknowns $\mu_{i,r}^{j,t}$ and x_i^j :*

$$\begin{cases} \frac{\partial}{\partial \mu_{i,r}^{j,t}} \Phi(\mu_{i,-1}^{j,-1}, \mu_{i,-1}^{j,1}, \mu_{i,1}^{j,1}, \mu_{i,1}^{j,-1}) \mu_{i,r}^{j,t} = \mu_{i,r}^{j,t} \mu_{i+r,-r}^{j+t,-t} x_i^j x_{i+r}^{j+t}, \\ 5n_i^j = \sum_{r,t} (\mu_{i,r}^{j,t} \mu_{i+r,-r}^{j+t,-t} x_i^j x_{i+r}^{j+t}), \end{cases} \quad \begin{matrix} i, j = 0, 1, 2, \\ r, t = -1, 1. \end{matrix}$$

This system has a unique solution. Moreover in terms of the solution of this system:

$$F_n = \prod_{i,j} (x_i^j)^{5n_i^j}.$$

So we have computed in an analytically exact way the function F_n . Since solving the 45×45 system in Lemma 6 when $n_i^j = 1/9, i, j = 0, 1, 2$ is trivial, we get by direct substitution:

Lemma 7. *The value of $5^{-5/2} F_n$ at the barycenter $n_{i,j} = 1/9, i, j = 0, 1, 2$ is equal to $(M_1)^2 = 25/24$. Therefore the value of F_n at the barycenter is > 58.2309 .*

Therefore all that it remains to be proved is that the function F_n maximizes at the barycenter.

From the Overlap Matrix Variables to the Conclusion. We have to prove the function F_n maximizes at the barycenter. Since we have assumed that the 3-coloring is balanced, i.e. $\forall i, \sum_j n_i^j = 1/3$ and $\forall j, \sum_i n_i^j = 1/3$, the domain of F_n has four degrees of freedom, all in the range $[0, 1/3]$. We swept over this domain

going over the points of a grid with 200 steps per dimension. The sweeping avoided a thin layer (of width $1/1000$) around the boundary (the points in the domain where at least one of the $n_i^j = 0$), because at the boundary the derivative of the original function F is infinity, thus no maximum occurs there. Moreover, we have computed the Hessian at the barycenter and proved that it is negative definite so in a neighborhood of the barycenter F_n is convex, and we know that F_n will be smaller than at the barycenter. At all points where we got a value for F_n greater than 58, we made an additional sweep at their neighborhood of step-size $1/7500$ (all these points were close the barycenter). Nowhere did we get a value greater than the value at the barycenter. To solve the 45×45 systems efficiently, we designed a fast search algorithm based on an algorithm by Byrd et al. [7]. We also devised a way to select good starting points for each system, whose basic principle was to select for each successive system a starting point that belonged to the convex hull of the solutions to the previous systems. The algorithm was implemented in Fortran and run on the IBM's supercomputer in the Barcelona Supercomputing Center, which consists of 2.268 dual 64-bit processor blade nodes with a total of 4.536 2.2 GHz PPC970FX processors. Therefore,

Theorem 2. *Random 5-regular configurations are 3-colorable with uniformly positive probability.*

In order to transfer this result to random 5-regular graphs, we need to consider the restriction of $\mathcal{C}_{n,5}$ to *simple* configurations (i.e. those without loops and multiple edges). We write \Pr^* and \mathbf{E}^* to denote probability and expectation conditional to the event “ $G \in \mathcal{C}_{n,5}$ is simple”. By using similar techniques to the ones developed in [5] (see also Theorem 2.6 in [14]), we get:

Lemma 8. *Let C be any fixed balanced 3-coloring of n vertices. Then, $\Pr[G \text{ is simple} \mid C \text{ is stable coloring of } G]$ is bounded away from 0, independently of C and n .*

From this lemma, we obtain: $\mathbf{E}^*(X) = \Theta(\mathbf{E}(X))$ and $\mathbf{E}^*(X^2) = O(\mathbf{E}(X^2))$. Therefore, $\Pr^*[X > 0] \geq \frac{(\mathbf{E}^*(X))^2}{\mathbf{E}^*(X^2)} = \Theta(1)$, and we can conclude:

Theorem 3. *The chromatic number of random 5-regular graphs is 3 with uniformly positive probability.*

Acknowledgement

We wish to thank D. Achlioptas and C. Moore for their essential help at all phases of this research, without which we would not have obtained the results of this work. We are also indebted to N.C. Wormald for his useful suggestions and we are thankful to the Barcelona Supercomputing Center and in particular to David Vicente for the help in running the optimization programs on the Mare Nostrum supercomputer.

References

1. D. Achlioptas and C. Moore. The asymptotic order of the random k -SAT threshold. In: *Proc. 43th Annual Symp. on Foundations of Computer Science (FOCS)*, 126–127, 2002.
2. D. Achlioptas and C. Moore. Almost all graphs with degree 4 are 3-colorable. *Journal of Computer and Systems Sciences* 67(2), 441–471, 2003.
3. D. Achlioptas and C. Moore. The chromatic number of random regular graphs. In: *Proc. 7th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX) and 8th International Workshop on Randomization and Computation (RANDOM)* (Springer, LNCS, 2004) 219–228.
4. D. Achlioptas and A. Naor. The two possible values of the chromatic number of a random graph. In: *36th Symposium on the Theory of Computing (STOC)*, 587–593, 2004.
5. B. Bollobás. A probabilistic proof of an asymptotic formula for the number of labelled regular graphs. *European Journal of Combinatorics* 1, 311–316, 1980.
6. B. Bollobás. *Random Graphs*. Academic Press, London-New York, 1985.
7. R.H. Byrd, P. Lu and J. Nocedal. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific and Statistical Computing* 16(5), 1190–1208, 1995.
8. S. Janson, T. Łuczak and A. Ruciński. *Random Graphs*. John Wiley, 2000.
9. F. Krzakała, A. Pagnani and M. Weigt. Threshold values, stability analysis and high- q asymptotics for the coloring problem on random graphs. *Phys. Rev. E* 70, 046705 (2004).
10. T. Łuczak. The chromatic number of random graphs *Combinatorica* 11, 45–54, 1991.
11. M. Mézard and R. Zecchina. Random K -satisfiability: from an analytic solution to a new efficient algorithm. *Phys.Rev. E* 66, 056126 (2002).
12. M. Molloy. *The Chromatic Number of Sparse Random Graphs*. Master’s Thesis, University of Waterloo, 1992.
13. L. Shi and N. Wormald. *Colouring random regular graphs* Research Report CORR 2004-24, Faculty of Mathematics, University of Waterloo, 2004.
14. N.C. Wormald. Models of random regular graphs. In: J.D. Lamb and D.A. Preece, eds., *Surveys in Combinatorics* (London Mathematical Society Lecture Notes Series, vol. 267, Cambridge U. Press, 1999) 239–298.

Optimal Integer Alphabetic Trees in Linear Time

T.C. Hu¹, Lawrence L. Larmore^{2,*}, and J. David Morgenthaler³

¹ Department of Computer Science and Engineering,
University of California, San Diego CA 92093, USA
`hu@cs.ucsd.edu`

² Department of Computer Science, University of Nevada, Las Vegas NV 89154, USA
`larmore@cs.unlv.edu`

³ Applied Biosystems, Foster City CA 94404, USA
`jdm123@gmail.com`

Abstract. We show that optimal alphabetic binary trees can be constructed in $O(n)$ time if the elements of the initial sequence are drawn from a domain that can be sorted in linear time. We describe a hybrid algorithm that combines the bottom-up approach of the original Hu-Tucker algorithm with the top-down approach of Larmore and Przytycka's Cartesian tree algorithms. The hybrid algorithm demonstrates the computational equivalence of sorting and level tree construction.

1 Introduction

Binary trees and binary codes are fundamental concepts in computer science, and have been intensively studied for over 50 years. In his 1952 paper, Huffman described an algorithm for finding an optimal code that minimizes the average codeword length [1]. Huffman coding is a classic, well known example of binary tree or binary code optimization, and has led to an extensive literature [2]. The problem of computing an optimal Huffman code has $\Theta(n \log n)$ time complexity, but requires only $O(n)$ time if the input is already sorted.

The problem of finding an optimal search tree where all data are in the leaves, also called an *optimal alphabetic binary tree* (OABT), was originally proposed by Gilbert and Moore [3], who give an $O(n^3)$ time algorithm based on dynamic programming, later refined by Knuth to $O(n^2)$ [4]. The first of several related $O(n \log n)$ time algorithms, the Hu-Tucker algorithm (HT), was discovered in 1971 [5]. Similar algorithms with better performance in special cases, though all $O(n \log n)$ time in the general case, are given in [6–9]. Different proofs of the correctness of these algorithms appear in [10–13].

We give a new algorithm for the OABT problem that takes advantage of additional structure of the input to allow construction of an OABT in $O(n)$ time if weights can be sorted in linear time, *e.g.*, if the weights are all integers in a small range. Our algorithm combines the bottom-up approach of the original

* Research supported by NSF grant CCR-0312093.

Hu-Tucker algorithm [5] with the top-down approach of Larmore and Przytycka's Cartesian tree algorithms [9].

Klawe and Mumej reduced sorting to Hu-Tucker based algorithms, resulting in a $\Omega(n \log n)$ lower bound for such *level tree* based solutions [8]. Larmore and Przytycka related the complexity of the OABT problem to the complexity of sorting, and gave an $O(n\sqrt{\log n})$ time algorithm for the integer case when sorting requires only $O(n)$ time [9]. Their Cartesian tree based algorithm provided new insight into the structure of the OABT problem, which we elaborate here. Our new algorithm requires sorting $O(n)$ items, which together with the reduction given by Klawe and Mumej [8], shows the computational equivalence of sorting and level tree construction.

2 Problem Definition and Earlier Results

Recall the definitions used in the Hu-Tucker algorithm [11]. We wish to construct an optimal alphabetic tree T from an initial ordered sequence of weights $S = \{s_1, \dots, s_n\}$, given as n *square nodes*. The square nodes will be the leaves of T , and retain their linear ordering. An optimal tree has minimal cost subject to that condition, defining the cost of tree T as:

$$\text{cost}(T) = \sum_{i=1}^n s_i l_i$$

where l_i is the distance from s_i to the root.

The first phase of the Hu-Tucker algorithm combines the squares to form internal nodes of a *level tree*. The second and third phases use the level tree to create an OABT in $O(n)$ time. The internal nodes created during the first (combination) phase are called *circular nodes*, or *circles*, to differentiate them from the square nodes of the original sequence. The *weight* of a square node is defined to be its original weight in the initial sequence, while the *weight* of a circular node is defined to be the sum of the weights of its children. The level tree is unique if there are no ties, or if a consistent tie-breaking scheme (such as the one given in [12]) is adopted. Algorithms that use the level tree method produce the same level tree, although the circular nodes are computed in different orders. The (non-deterministic) *level tree algorithm* (LTA) given below generalizes those algorithms. As an example, the Hu-Tucker algorithm is an LTA where the circular nodes are constructed in order of increasing weight.

We remark that circular nodes have also been called *packages* [9], *crossable nodes* [8], and *transparent nodes* [12].

The *index* of each square node is its index in the initial sequence, and the *index* of any node of the level tree is the smallest index any leaf descendant. We will refer to square nodes as s_i , circular nodes as c_i and nodes in general as v_i , where i is the index of the node. By an abuse of notation, we also let v_i denote the weight of the node v_i . If two weights are equal, we use indices as tie-breakers. See [12] for the detailed description of this tie-breaking scheme.

Initially, we are given the sequence of items which are all square nodes. As the algorithm progresses, nodes are deleted and circular nodes are inserted into the node sequence.

Definition 1. *Two nodes in a node sequence are called a compatible pair if all nodes between them are circular nodes. We write (v_a, v_b) to represent the pair itself, and also, by an abuse of notation, the combined weight of the pair, $v_a + v_b$.*

Definition 2. *A compatible pair of nodes (v_b, v_c) is a locally minimum compatible pair, written $\text{lmcp}(v_b, v_c)$, when the following is true: $v_a > v_c$ for all other nodes v_a compatible with v_b and $v_b < v_d$ for all other nodes v_d compatible with v_c . By an abuse of notation, we will use $\text{lmcp}(v_b, v_c)$ to refer to the pair of nodes and also to their combined weight.*

2.1 The Hu-Tucker Algorithm and LTA

We now describe the Hu-Tucker Algorithm [5]. Define $\text{COMBINE}(v_a, v_b)$ to be the operation that deletes v_a and v_b from S and returns the new circular node $c_a = (v_a, v_b)$. (Note that if v_a is a circular node, then the new node will have the same name as its left child, but that will not cause confusion in the algorithm since the child will be deleted from the node sequence.)

HU-TUCKER ALGORITHM returns the OABT for $S = \{s_1, \dots, s_n\}$.

1. While S contains more than one node:
 - 1.1. Let (v_a, v_b) be the least weight compatible pair in S .
 - 1.2. Insert $c_a = \text{COMBINE}(v_a, v_b)$ into S at the same position as v_a .
2. Let c^* be the single circular node remaining in S .
3. For each $1 \leq i \leq n$, let d_i be the depth of s_i in the tree rooted at c^* .
4. Let T be the unique alphabetic tree whose leaves are s_1, \dots, s_n such that, for each i , the depth of s_i in T is d_i .
5. Return T .

The tree rooted at c^* is the level tree of the original sequence. The level tree can be constructed by combining locally minimal compatible pairs in any order. Thus, we generalize HT to the *level tree algorithm* (LTA) as follows:

LEVEL TREE ALGORITHM(LTA) returns the OABT for $S = \{s_1, \dots, s_n\}$.

1. While S contains more than one node:
 - 1.1. Let (v_a, v_b) be **any** lmcp in S .
 - 1.2. Insert $c_a = \text{COMBINE}(v_a, v_b)$ into S .
2. Let c^* be the single node remaining in S .
3. For each $1 \leq i \leq n$, let d_i be the depth of s_i in the tree rooted at c^* .
4. Let T be the unique alphabetic tree whose leaves are s_1, \dots, s_n such that, for each i , the depth of s_i in T is d_i .
5. Return T .

In the insertion step of LTA, namely step 1.2 above, the new circular node c_a is placed in the position vacated by its left child. However, LTA gives the correct level tree if c_a is placed anywhere between *leftWall*(c_a) and *rightWall*(c_a), where

$leftWall(c_a)$ is the nearest square node to the left of c_a whose weight is greater than c_a , and $rightWall(c_a)$ is the nearest square node to the right of c_a whose weight is greater than c_a . We will place fictitious infinite squares s_0 and s_∞ at either end of the initial sequence, as described in Section 2.2, so $leftWall(c_a)$ and $rightWall(c_a)$ are always defined. The choices in LTA do not alter the level tree, but may change the order in which the circular nodes are computed [6–8].

LTA consists of $n - 1$ iterations of the main loop. Its time complexity is dominated by the amortized time to execute one iteration of this loop. The Hu-Tucker algorithm takes $O(n \log n)$ to construct the level tree, because it requires $O(\log n)$ time to find the minimum compatible pair and update the structure.

Since the number of possible level trees on a list of length n is $2^{\Theta(n \log n)}$, the time complexity of LTA, in the decision tree model of computation, must be $\Omega(n \log n)$ in general. Our algorithm, also a deterministic version of LTA, makes use of $O(n)$ -time sorting for integers in a restricted range, and takes $O(n)$ time to construct the OABT, provided all weights are integers in the range $0 \dots n^{O(1)}$.

In our algorithm, we do not necessarily actually insert a circular node into the node sequence. Instead, we make use of data structures which are associated with certain parts of the sequence, which we call *mountains* and *valleys* (see Section 2.2). Each circular node is *virtually* in the correct position, and our data structures ensure that we can always find a pair of nodes which *would have been* a locally minimal compatible pair if the nodes had actually been inserted.

During the course of the algorithm, nodes can be moved from one data structure to another before being combined, and nodes are also sorted within data structures. We achieve overall linear time by making sure that the combined time of all such steps amortizes to $O(1)$ per node, and that the next locally minimal compatible pair can always be found in $O(1)$ time.

The contribution of this paper is that by restricting the Cartesian tree (see Section 2.3) to *mountains* we reduce the complexity of the integer algorithm given in [9] from $O(n\sqrt{\log n})$ to linear.

2.2 Mountains and Valleys

Any input sequence contains an alternating series of pairwise local minima (the lmcps) and local maxima, which we call *mountains*. It is the mountains and their structure that enable us to relocate each new circle in constant amortized time.

“Dual” to the definition of locally minimal compatible pair, we define:

Definition 3. *A compatible pair of square nodes (s_d, s_e) is a locally maximum adjacent pair if its weight is greater than all other compatible pairs containing either element.*

Definition 4. *A mountain is the larger of a locally maximum adjacent pair of nodes in the initial weight sequence. If node s_i is a mountain, we also label it M_i .*

We extend the initial sequence S by adding two “virtual” mountains $s_0 = M_0$ and $s_\infty = M_\infty$ of infinite weight to the ends. Write S^+ for the resulting *extended weight sequence*. The virtual mountains are never combined, but are only for notational convenience, giving us a uniform definition of “valley” in S^+ .

Definition 5. A valley is a subsequence of the initial weights between and including two adjacent mountains in the extended weight sequence S^+ . We label the valley formed by mountains M_i and M_j as $V(M_i, M_j)$. The top of valley $V(M_i, M_j)$ is its minimum mountain $\min\{M_i, M_j\}$, while the bottom is its *lmc*p.

Valleys (or the equivalent *easy tree* [9]) are a basic unit of the combination phase. All nodes within a single valley can be combined in linear time (see Section 3). During the combination phase of our algorithm, we repeatedly compute the minimum compatible pair in each valley; As nodes combine, this pair may not be an *lmc*p. To handle this situation, we first generalize locally minimum compatible pair to apply to any subsequence.

Definition 6. Let $S_i^p = (v_i, v_j, \dots, v_p)$ be a subsequence of nodes. Then the minimum compatible pair of S_i^p , written $\text{mcp}(i, p)$, is the compatible pair of minimum weight: $\text{mcp}(i, p) = \min \{(v_a, v_d) \mid i \leq a < d \leq p\}$

Note that if $\text{mcp}(i, p)$ does not contain either v_i or v_p , it must be an *lmc*p. That is, the local minimum in the subsequence is also a local minimum in the full sequence because we must see a full ‘window’ of four nodes in order to apply the definition of *lmc*p. If $\text{mcp}(i, p)$ includes either end, we cannot do so. This fact motivates the use of valleys in our algorithm, and the need to distinguish mountains for special handling.

2.3 Cartesian Trees

The Cartesian tree data structure is originally described in [14]. Larmore and Przytycka base their OABT algorithms on Cartesian trees over the entire input sequence [9], but here we will limit the tree to contain only mountains. We recursively define the Cartesian tree of any sequence of weights:

Definition 7. The Cartesian tree for an empty sequence is empty. The root of a Cartesian tree for a non-empty sequence is the maximum weight, and its children are the Cartesian trees for the subsequences to the right and left of the root.

We construct the Cartesian tree of the sequence of mountains. We label mountain M_i 's parent in the Cartesian tree as $p(M_i)$.

2.4 Algorithm Overview

At each phase of our algorithm, nodes in every valley are combined independently as much as possible. Any new circular node which is greater than the top of its valley is stored in a global set U for later distribution.

When no more combinations are possible, we remove all mountains which have fewer than two children in the Cartesian tree (that is always more than half the mountains) and move the contents of the global set of circles U to sets associated with the remaining mountains. As mountains are removed, their associated sets are combined and sorted to facilitate the next round of combinations.

3 Single Valley LTA

This section gives an LTA for a single valley containing only one `lmc`p, to improve the reader's intuition. We use a queue of circles in each valley to help find each new `lmc`p in turn. We label this queue $Q_{f,g}$ for valley $V(M_f, M_g)$. Rather than adding the circles back into the sequence, we put them into the queue instead. For purposes of operation `COMBINE`, we consider the queue to be a part of the sequence, located at the index of its initial circle.

`SINGLE VALLEY LTA` computes an optimal level tree in linear time since constant work is performed for each iteration. Only the main loop differs from the general LTA given above, so we omit steps 2–5.

`SINGLE VALLEY LTA` for S^+ containing single valley $V(M_0, M_\infty)$.

1. While the sequence S contains more than one node:
 - 1.1. Let `lmc`p(v_a, v_b) be the result of `VALLEY MCP` for valley $V(M_0, M_\infty)$.
 - 1.2. Add $c_a = \text{COMBINE}(v_a, v_b)$ to the end of $Q_{0,\infty}$.

A little bookkeeping helps us determine each `lmc`p in constant time. We only need to consider the six nodes at the bottom of the valley in order to find the next `lmc`p. These six nodes are the valley's *active* nodes. Let the bottom two nodes on $Q_{i,j}$ be c_x and c_y , if they exist, where $c_x < c_y$. Any nodes that do not exist (e.g. the queue contains only one node) are ignored.

`SUBROUTINE VALLEY MCP` returns `mcp`(i, j) for valley $V(M_i, M_j)$.

1. Let s_a be the square left adjacent to $Q_{i,j}$.
2. Let s_f be the square right adjacent to $Q_{i,j}$.
3. Return $\min \{(s_a, s_{a-1}), (s_a, c_x), (c_x, c_y), (c_x, s_f), (s_f, s_{f+1}), (s_a, s_f)\}$, ignoring any pairs with missing nodes. In addition, we require $a > i$ (otherwise ignore s_{a-1}), and $f < j$ (otherwise ignore s_{f+1}).¹

`SINGLE VALLEY LTA` relies on the fact that each `lmc`p must be larger than the last within a valley. It also solves the Huffman coding problem in linear time if the input is first sorted [11].

4 Multiple Valleys

Consider the operation of `SINGLE VALLEY LTA` for an input sequence containing more than one valley. We can independently combine nodes in each valley only to a certain point. When the weight of a new circle is greater than one of the adjacent mountains, that circle no longer belongs in the current valley's queue, and must be *exported*, to use the terminology of [9]. But the valley into which the circle must be imported does not yet exist. Eventually, when the mountain between them combines, two (or more) valleys will merge into a new valley. If its adjacent mountains both weigh more than the new circle, this valley will import the circle. Mountains separate valleys in a hierarchical fashion, concisely represented by a Cartesian tree.

¹ The additional restrictions ensure that we stay within the valley.

4.1 Cartesian Tree Properties

For the k initial valleys separated by the nodes of this Cartesian tree, we have $k - 1$ latent valleys that will emerge as mountains combine. Adding the initial valleys as leaves to the Cartesian tree of mountains yields a full binary tree of nested valleys with root $V(M_0, M_\infty)$. The internal nodes of this valley tree (the mountains) precisely correspond to the merged valleys created as our algorithm progresses. A mountain node *branches* in the Cartesian tree if its children are both mountains. Our algorithm takes advantage of the following property of Cartesian trees:

Property 1. Between each pair of adjacent leaf nodes in a Cartesian tree, there is exactly one branching node. Proof is by construction.

This property implies that for k leaves, there are $k - 1$ branching nodes in a Cartesian tree. A tree may have any number of additional internal nodes with a single child, which we call *non-branching nodes*. Our algorithm handles each of these three types of mountains differently, so we will take a closer look at their local structure.

Consider three adjacent mountain M_h, M_i , and M_j , where $h < i < j$. We can determine the type of mountain M_i in the Cartesian tree by comparing its weight to the weights of its two neighbors. There are three cases:

- If $M_h > M_i < M_j$, then M_i is a leaf in the Cartesian tree.
- If $M_h < M_i > M_j$, then M_i is a branching node separating two *active regions*.
- If $M_h < M_i < M_j$, or $M_h > M_i > M_j$, then M_i is a non-branching node and the top of the valley separating M_i and its larger neighbor.

An *active region* is the set of all the valleys between two adjacent branching nodes. During execution of our algorithm, the active regions of each iteration will form single valleys in the next iteration. To bound the number of iterations our algorithm must execute, we need one additional Cartesian tree property. Let $|C|$ be the number of nodes in tree C . As every non-empty binary tree C contains between 1 and $\lceil |C|/2 \rceil$ leaves, with the help of Property 1 we have:

Property 2. Cartesian tree C contains fewer than $|C|/2$ branching nodes.

4.2 Filling a Valley

The structure of the mountains and valleys creates two transition points during the combination phase in each valley. The first is the export of the first circle. Before that point, all nodes stay within a valley and new circles are added to its queue. After the first transition point, all new circles are exported until the valley is eliminated. The removal of an adjacent mountain is the second transition point, which merges the current valley with one or more of its neighbors. After each combination, we maintain the following valley invariants needed by VALLEY MCP. Each valley $V(M_i, M_j)$ must contain:

- Two adjacent mountains.
- A possibly empty sorted queue of circles $Q_{i,j}$, with all circles $< \min\{M_i, M_j\}$.
- A cursor pointing into $Q_{i,j}$, initialized to point to the front of the queue.

- A possibly empty pairwise monotonic decreasing subsequence of squares to the right of M_i .
- A possibly empty pairwise monotonic increasing subsequence of squares to the left of M_j .

First, we note that while $\text{mcp}(i, j) < \min\{M_i, M_j\}$, a valley is isolated and we define the following subroutine INITIAL FILL to reach the first transition point. We call this subroutine the first time we handle a new valley, whether that valley appears in the initial sequence or after merging.

As new valleys are created, they will be prepopulated with a sorted queue containing imported circles. We will use the cursor to help us insert newly created circles into this queue in case it contains circles larger than the next lmcp .

SUBROUTINE INITIAL FILL for valley $V(M_i, M_j)$.

1. Find the initial $\text{lmcp}(v_a, v_b)$ in $V(M_i, M_j)$.
2. While $\text{lmcp}(v_a, v_b) < \min\{M_i, M_j\}$:
 - 2.1. Insert $c_a = \text{COMBINE}(v_a, v_b)$ into $Q_{i,j}$, advancing the cursor as needed.
 - 2.2. Let $\text{lmcp}(v_a, v_b)$ be the result of VALLEY MCP for valley $V(M_i, M_j)$.

Notice that INITIAL FILL is nearly the same as SINGLE VALLEY LTA. We have added a stronger condition on the while loop, and we need to use the cursor to merge new circles into the queue. We now show how to fill a valley:

SUBROUTINE FILL VALLEY returns queue of circles Q for valley $V(M_i, M_j)$.

1. Call INITIAL FILL for $V(M_i, M_j)$.
2. Create empty queue Q .
3. While $\min\{M_i, M_j\} \notin \{\text{VALLEY MCP for } V(M_i, M_j)\}$:
 - 3.1. Add $c_a = \text{COMBINE}(\text{VALLEY MCP for } V(M_i, M_j))$ to end of Q .
4. Return Q .

After FILL VALLEY returns, the top of the valley, $\min\{M_i, M_j\}$, is an element of $\text{mcp}(i, j)$, and we say that valley $V(M_i, M_j)$ is *full*. Combining or otherwise removing the minimum mountain requires merging valleys and reestablishing the necessary valley invariants, and also handling the exported queue of circles. For each mountain M_i , we will store the circles imported by its corresponding latent valley in an unsorted set U_i , where $c_s \in U_i \implies M_i < c_s < p(M_i)$. When a new valley is created, this set will become its imported queue Q .

4.3 Merging Valleys

For valley $V(M_i, M_j)$, $\text{mcp}(i, j)$ is an lmcp if it does not contain M_i or M_j , since nodes inside a valley, *i.e.*, between the mountains, cannot be compatible with any outside the valley (lemma 5 in [11]). However, the mountains themselves each participate in two valleys, thus these nodes are compatible with other nodes outside the single valley subsequence $\{M_i, \dots, M_j\}$.

First, consider a mountain M_i smaller than its neighbors M_h and M_j . M_i must be a leaf in the Cartesian tree of mountains, and after adjacent valleys $V(M_h, M_i)$ and $V(M_i, M_j)$ are full, $M_i \in \text{mcp}(h, i)$ and $M_i \in \text{mcp}(i, j)$. We are

ready to merge these two valleys into $V(M_h, M_j)$ by combining the mountain. Note that $\text{mcp}(h, j) = \min \{ \text{mcp}(h, i), \text{mcp}(i, j) \}$, which will form an lmcp if it does not contain M_h or M_j . To explain the combination of valleys, we start with this leaf mountain case where $M_h \notin \text{mcp}(h, j)$ and $M_j \notin \text{mcp}(h, j)$.

Two Valley Case. When both valleys adjacent to leaf mountain M_i are full, we merge them into a single new valley and establish the valley invariants needed by VALLEY MCP. Subroutine MERGE TWO VALLEYS provides the bookkeeping for the merge. The result is new valley $V(M_h, M_j)$, ready to be filled. MERGE TWO VALLEYS requires that both valleys are full, and so each contain at most one node (square or circle) smaller than and compatible with M_i . In addition, the lmcp may not contain two mountains; this omitted three valley case is in fact even simpler.

SUBROUTINE MERGE TWO VALLEYS returns set of circles W for mountain M_i .

1. Create empty set W .
2. Let $V(M_h, M_i)$ and $V(M_i, M_j)$ be the valleys to the left and right of M_i .
3. Merge queues $Q_{h,i}$ and $Q_{i,j}$, each of which contains at most one circle, to create $Q_{h,j}$. Initialize a new cursor to point to the front of $Q_{h,j}$.
4. Sort U_i , the imported circles for the new valley, and add to the end of $Q_{h,j}$.
5. Find the smallest node v_{\min} compatible with M_i from among the only three possible: the front of $Q_{h,j}$, or the squares left or right adjacent to M_i .
6. Let circle $c_m = \text{COMBINE}(v_{\min}, M_i)$, assuming v_{\min} is to the left of M_i .
7. Create new valley $V(M_h, M_j)$ with $Q_{h,j}$, removing M_i , U_i , $V(M_h, M_i)$ and $V(M_i, M_j)$ from further consideration.
8. If circle $c_m < \min\{M_h, M_j\}$ then:
 - 8.1. Insert c_m into $Q_{h,j}$, advancing the cursor as needed.
 - else:
 - 8.2. Add c_m to W .
9. Return W .

4.4 Removing Non-branching Mountains

In this section we explain the removal of non-branching mountains. These mountains are adjacent to one larger and one smaller neighbor, and lie between a branching mountain and a leaf mountain, or at either end of the sequence.

For example, consider valley $V(M_i, M_j)$, where $M_i < M_j$. Once $V(M_i, M_j)$ is full, we remove mountain M_i by either combining it or turning it into a normal square. When M_i no longer separates its adjacent valleys, we merge them into one. This process continues repeatedly until a single valley separates each adjacent branching and leaf mountain.

Since a full valley contains at most one node smaller than the minimum mountain M_i , we need to consider three cases. Let $M_h < M_i < M_j$, and let s_a and s_d be the squares left and right adjacent to M_i , respectively.

1. The filled valley has an empty queue and no square smaller than M_j . Then M_i is no longer a mountain, as $s_a < M_i < s_d$. We make M_i into a normal

- square s_i by removing the mountain label. Square s_i becomes part of the pairwise monotonic increasing sequence of squares to the left of M_j .
2. The filled valley contains a single square s_d smaller than M_i . So $s_d < M_i < v_b$ for all v_b compatible with s_d . If $s_a > s_d$, then M_i and s_d must form $\text{lmcp}(M_i, s_d)$. Combining $\text{lmcp}(M_i, s_d)$ lets us connect the pairwise monotonic increasing subsequence of squares to the left of M_i with the subsequence to the left of M_j , forming a single pairwise increasing subsequence. Otherwise, we can convert M_i into a normal square node as in the case above.
 3. The filled valley contains a single circle c_d . This case is similar to the previous case, but may require that we convert c_d into a *pseudo-square*. We can treat this circle as though it were square because it is smaller than either neighbor and must combine before its transparency has any effect.

Subroutine REMOVE MOUNTAIN removes non-branching, internal nodes from the Cartesian tree. Without loss of generality, assume that the left adjacent mountain $M_h < M_i$, while the right adjacent mountain $M_j > M_i$.

SUBROUTINE REMOVE MOUNTAIN returns set of circles W for mountain M_i .

1. Create empty set W .
2. Let $(M_i, v_a) = \text{mcp}(i, j)$.
3. If (M_i, v_a) is an lmcp then:²
 - 3.1. Add $c_i = \text{COMBINE}(M_i, v_a)$ to W .
else:
 - 3.2. Rename M_i to s_i and remove M_i from other data structures.
 - 3.3. If $Q_{i,j}$ is not empty, convert remaining circle into a pseudo-square.
4. Add U_i to U_h .
5. Create new valley $V(M_h, M_j)$, removing U_i , $V(M_h, M_i)$ and $V(M_i, M_j)$.
6. Return W .

When REMOVE MOUNTAIN completes, mountain M_i has been removed from the sequence and many valley invariants established for the newly created valley. The missing invariants involve $Q_{h,j}$, which we will postpone creating until we have merged all the valleys between adjacent branching mountains. All that is needed is to sort U_h , which occurs in subroutine MERGE TWO VALLEYS.

5 K Valley LTA

Let \hat{C} be the Cartesian tree formed from the mountains in the original sequence. Each node of \hat{C} separates two valleys in that sequence. As mentioned earlier, we call a non-leaf node in \hat{C} with a single child a *non-branching node*. Our final algorithm shrinks \hat{C} by removing at least half of the nodes at each iteration. When all the nodes of \hat{C} have been combined, we compute the level tree using INITIAL FILL. The input sequence S^+ is the extended sequence created by adding two nodes of infinite weight to either end of the original sequence S .

K VALLEY LTA computes level tree for S^+ .

1. Compute the Cartesian tree \hat{C} from the $k - 1$ mountains in the original S .
2. Create a global empty set U , and an empty set U_i for each mountain M_i .

² (M_i, v_a) is an lmcp if and only if $s_b > v_a$ for the square node s_b to the left of M_i .

3. While \widehat{C} contains more nodes:
 - 3.1. For each valley $V(M_i, M_j)$ in the current sequence:
 - 3.1.1. Add $Q = \text{FILL VALLEY}$ for $V(M_i, M_j)$ to U .
 - 3.2. For each non-branching mountain M_i in \widehat{C} :
 - 3.2.1. Add $W = \text{REMOVE MOUNTAIN}$ for M_i to U .
 - 3.2.2. Delete all the mountains combined in the previous step from \widehat{C} .
 - 3.3. Use Static Tree Set Union (see below) to place U 's circles into the U_i .
 - 3.4. For each leaf mountain M_j in \widehat{C} :
 - 3.4.1. Add $W = \text{MERGE TWO VALLEYS}$ for M_j to U .
4. Use the sorted set from the final step 3.3 to create $Q_{0,\infty}$ and call INITIAL FILL for $V(M_0, M_\infty)$. The final circle remaining between M_0 and M_∞ is the root of the level tree.

Static Tree Set Union (STSU) is a specialized version of the Union-Find problem that applies to a tree, and computes a sequence of *link* and *find* operations in linear time [15]. We can use this algorithm to find the set U_i in which each circle in global set U belongs in linear time if we first sort U together with the mountains remaining in \widehat{C} .

We apply the approach described in [9], first initializing the family of named sets with a singleton set for each node of \widehat{C} . We attach each circle of U to \widehat{C} as a child of its exporter, the argument of the call to REMOVE MOUNTAIN where it was created. Next, we sort all the nodes of this extended tree by weight to create sorted list L' . Processing nodes of L' in increasing weight order, we execute a sequence of *link* and *find* operations in $O(n)$ time as follows:

- If the current node is M_j , perform *link*(M_j), which adds the contents of the set containing M_j to the set containing $p(M_j)$, then deletes the old set containing M_j .
- If the current node is circle c_m , perform *find*(c_m), obtaining the set containing mountain $p(c_m)$. That set is named with the minimum ancestor mountain M_j in \widehat{C} that dominates c_m . Determine whether c_m is to the right or left of M_j . Add c_m to the U_i of the mountain child on that side of M_j .

6 Complexity

The algorithm is dominated by sorting, which occurs in steps 3.3 and 3.4.1 We now show that the algorithm sorts only $O(n)$ nodes.

Lemma 1. *The STSU in step 3.3 sorts $O(n)$ nodes altogether.*

Proof. Initially $|\widehat{C}| = k < \frac{n}{2}$. By Property 2, we remove at least half the mountains at each iteration of step 3. Thus, over all iterations, at most $2k < n$ mountains are sorted. No circle is sorted more than once by step 3.3, since set U contains only newly created circles. The total of all circles sorted by step 3.3 is therefore less than n . Over all steps 3.3, we sort a total of less than $2n$ items.

Lemma 2. *Step 3.4.1 sorts $O(n)$ nodes altogether.*

Proof. Each U_i is sorted at most once. Each circle appears in only one U_i being sorted, as it is moved to some $Q_{a,b}$ after sorting.

Theorem 1. K VALLEY LTA sorts $O(n)$ weights.

Proof. No other sorting is performed; apply lemmas 1 and 2.

Theorem 2. For weights taken from an input domain that can be sorted in linear time, an optimal alphabetic binary tree can be constructed in $O(n)$ time.

Proof. All other operations are linear in the size of the input. Proof follows from Theorem 1 and the linear creation of the OABT from the level tree.

7 Conclusion

We have given an algorithm to construct optimal alphabetic binary trees in time bounded by sorting. This algorithm shows that sorting and level tree construction are equivalent problems, and leads to an $O(n)$ time solution to the integer alphabetic binary tree problem for integers in the range $0 \dots n^{O(1)}$.

References

1. Huffman, D.A.: A method for the construction of minimum redundancy codes. *Proceedings of the IRE* **40** (1952) 1098–1101
2. Abrahams, J.: Code and parse trees for lossless source encoding. In: *Proceedings Compression and Complexity of Sequences*. (1997) 146–171
3. Gilbert, E.N., Moore, E.F.: Variable length binary encodings. *Bell System Technical Journal* **38** (1959) 933–968
4. Knuth, D.E.: Optimum binary search tree. *Acta Informatica* **1** (1971) 14–25
5. Hu, T.C., Tucker, A.C.: Optimal computer search trees and variable-length alphabetic codes. *SIAM Journal on Applied Mathematics* **21** (1971) 514–532
6. Garsia, A.M., Wachs, M.L.: A new algorithm for minimal binary search trees. *SIAM Journal on Computing* **6** (1977) 622–642
7. Hu, T.C., Morgenthaler, J.D.: Optimum alphabetic binary trees. In: *Combinatorics and Computer Science: 8th Franco-Japanese and 4th Franco-Chinese Conference*. *Lecture Notes in Computer Science*, volume 1120, Springer-Verlag (1996) 234–243
8. Klawe, M.M., Mumey, B.: Upper and lower bounds on constructing alphabetic binary trees. *SIAM Journal on Discrete Mathematics* **8** (1995) 638–651
9. Larmore, L.L., Przytycka, T.M.: The optimal alphabetic tree problem revisited. *Journal of Algorithms* **28** (1998) 1–20
10. Hu, T.C.: A new proof of the T-C algorithm. *SIAM Journal on Applied Mathematics* **25** (1973) 83–94
11. Hu, T.C., Shing, M.T.: *Combinatorial Algorithms*, Second Edition. Dover (2002)
12. Karpinski, M., Larmore, L.L., Rytter, W.: Correctness of constructing optimal alphabetic trees revisited. *Theoretical Computer Science* **180** (1997) 309–324
13. Ramanan, P.: Testing the optimality of alphabetic trees. *Theoretical Computer Science* **93** (1992) 279–301
14. Gabow, H.N., Bentley, J.L., Tarjan, R.E.: Scaling and related techniques for geometry problems. In: *Proceedings of the 16th ACM Symposium on Theory of Computation*. (1984) 135–143
15. Gabow, H.N., Tarjan, R.E.: A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences* **30** (1985) 209–221

Predecessor Queries in Constant Time?

Marek Karpinski* and Yakov Nekrich**

Abstract. In this paper we design a new static data structure for batched predecessor queries. In particular, our data structure supports $O(\sqrt{\log n})$ queries in $O(1)$ time per query and requires $O(n^{\varepsilon\sqrt{\log n}})$ space for any $\varepsilon > 0$. This is the first $o(N)$ space and $O(1)$ amortized time data structure for arbitrary $N = \Omega(n^{\varepsilon\sqrt{\log n}})$ where N is the size of the universe. We also present a data structure that answers $O(\log \log N)$ predecessor queries in $O(1)$ time per query and requires $O(n^{\varepsilon \log \log N})$ space for any $\varepsilon > 0$. The method of solution relies on a certain way of searching for predecessors of all elements of the query *in parallel*.

In a general case, our approach leads to a data structure that supports $p(n)$ queries in $O(\sqrt{\log n/p(n)})$ time per query and requires $O(n^{p(n)})$ space for any $p(n) = O(\sqrt{\log n})$, and a data structure that supports $p(N)$ queries in $O(\log \log N/p(N))$ time per query and requires $O(n^{p(N)})$ space for any $p(N) = O(\log \log N)$.

1 Introduction

Given a set A of integers, the predecessor problem consists in finding for an arbitrary integer x the biggest $a \in A$ such that $a \leq x$. If x is smaller than all elements in A , a default value is returned. This fundamental problem was considered in a number of papers, e.g., [AL62], [EKZ77], [FW94], [A95], [H98], [AT00], [BF02], [BCKM01]. In this paper we present a static data structure that supports predecessor queries in $O(1)$ amortized time.

In the *comparison model*, if only comparisons between pairs of elements are allowed, the predecessor problem has time complexity $O(\log n)$, where n is the number of elements. A standard information-theoretic argument proves that $\lceil \log n \rceil$ comparisons are necessary. This lower bound had for a long time been believed to be also the lower bound for the integer predecessor problem. However in [E77], [EKZ77] a data structure supporting predecessor queries in $O(\log \log N)$ time, where N is the size of the universe, was presented. Fusion trees, presented by Fredman and Willard [FW94], support predecessor queries in $O(\sqrt{\log n})$ time, independently of the size of the universe. This result was further improved in other important papers, e.g., [A95], [AT00],[BF02]. In the paper of Beame and Fich [BF02], it was shown that any data structure using

* Dept. of Computer Science, University of Bonn. E-mail marek@cs.uni-bonn.de. Work partially supported by a DFG grant, Max-Planck Research Prize, and IST grant 14036 (RAND-APX).

** Dept. of Computer Science, University of Bonn. E-mail yasha@cs.uni-bonn.de. Work partially supported by IST grant 14036 (RAND-APX).

$n^{O(1)}$ words of $(\log N)^{O(1)}$ bits, requires $\Omega(\sqrt{\log n / \log \log n})$ query time in the worst case. In [BF02] the authors also presented a matching upper bound and transformed it into a linear space and $O(\sqrt{\log n / \log \log n})$ time data structure, using the exponential trees of Andersson and Thorup [A96],[AT00].

Ajtai, Fredman and Komlós [AFK84] have shown that if word size is $n^{\Omega(1)}$, then predecessor queries have time complexity $O(1)$ in the cell probe model ([Y81]). Obviously, there exists a $O(N)$ space and $O(1)$ query time static data structure for the predecessor queries. Brodnik, Carlsson, Karlsson, and Munro [BCKM01] presented a constant time and $O(N)$ space dynamic data structure. But their data structure uses an unusual notion of the word of memory: an individual bit may occur in a number of different words.

While in real-time applications every query must be processed as soon as it is known to the data base, in many other applications we can collect a number of queries and process the set of queries simultaneously. In this scenario, the size of the query set is also of interest. Andersson [A95] presented a static data structure that uses $O(n^{\varepsilon \log n})$ space and answers $\log n$ queries in time $O(\log n \log \log n)$. Batched processing is also considered in e.g., [GL01], where batched queries to unsorted data are considered.

In this paper we present a static data structure that uses $O(n^{p(n)})$ space and answers $p(n)$ queries in $O(\sqrt{\log n})$ time, for any $p(n) = O(\sqrt{\log n})$. In particular, we present a $O(n^{\varepsilon \sqrt{\log n}})$ space data structure that answers $\sqrt{\log n}$ queries in $O(\sqrt{\log n})$ time. The model used is RAM model with word size b , so that the size of the universe $N = 2^b$. To the best of our knowledge, this is the first algorithm that uses $o(N)$ space and words with $O(\log N)$ bits, and achieves $O(1)$ amortized query time for arbitrary $N = \Omega(n^{\varepsilon \sqrt{\log n}})$.

If the universe is bounded (e.g., $\log \log N = o(\sqrt{\log n})$), our approach leads to a $O(n^{p(N)})$ space data structure that answers $p(N)$ queries in time $O(\log \log N)$, where $p(N) = O(\log \log N)$. Thus, there exists a data structure that answers $\log \log N$ queries in $O(\log \log N)$ time and uses $O(n^{\varepsilon \log \log N})$ space. For instance, for $N = n^{\log^{O(1)} n}$, there is a $O(n^{\varepsilon \log \log n})$ space and constant amortized time data structure.

The main idea of our method is to search in a certain way for predecessors of all elements of the query set *simultaneously*. We reduce the key size for all elements by multiple membership queries in the spirit of [BF02]. When the key size is sufficiently small, predecessors can be found by multiple comparisons.

After some preliminary definitions in Section 2, we give an overview of our method in Section 3. In Section 3 a $O(n^{2\sqrt{\log n+2}})$ space and $O(1)$ amortized time data structure is also presented. We generalize this result and describe its improvements in Section 4.

2 Preliminaries and Notation

In this paper we use the RAM model of computation that supports addition, multiplication, division, bit shifts, and bitwise boolean operations in $O(1)$ time. Here and further w denotes the word size; b denotes the size of the keys, and

we assume without loss of generality that b is a power of 2. Query set $Q = \{x_1, x_2, \dots, x_q\}$ is the set of elements whose predecessors should be found. Left and right bit shift operations are denoted with \ll and \gg respectively, i.e. $x \ll k = x \cdot 2^k$ and $x \gg k = x \div 2^k$, where \div is the integer division operation. Bitwise logical operations are denoted by AND, OR, XOR, and NOT. If x is a binary string of length k , where k is even, x^u denotes the prefix of x of length $k/2$, and x^l denotes the suffix of x of length $k/2$.

In the paper of Beame and Fich [BF02], it is shown how multiple membership queries, can be answered simultaneously in $O(1)$ time, if the word size is sufficiently large. The following statement will be extensively used in our construction.

Lemma 1. *Given sets S_1, S_2, \dots, S_q such that $|S_i| = n_i$, $S_i \subset [0, 2^b - 1]$, and $q \leq \sqrt{\frac{w}{b}}$, there is a data structure that uses $O(bq \prod_{i=1}^q 2^{\lceil \log n_i \rceil + 1})$ bits, can be constructed in $O(q \prod_{i=1}^q 2^{\lceil \log n_i \rceil + 1})$ time, and answers q queries $p_1 \in S_1?, p_2 \in S_2?, \dots, p_q \in S_q?$ in $O(1)$ time.*

This Lemma is a straightforward extension of Lemma 4.1 in [BF02].

A predecessor query on a set S of integers in the range $[0, 2^b - 1]$ can be reduced in $O(1)$ time to a predecessor query on set S' with at most $|S|$ elements in the range $[0, 2^{b/2} - 1]$. This well known idea and its variants are used in van Emde Boas data structure [E77], x-fast trie [W83], as well as in the number of other important papers, e.g., [A95], [A96], [AT00].

In this paper the following (slightly modified) construction will be used. Consider a binary trie T for elements of S . Let $T_0 = T$. Let $H(S)$ be the set of non-empty nodes of T_0 on level $b/2$. That is, $H(S)$ is the set of prefixes of elements in S of length $b/2$. If $|S| \leq 4$, elements of S are stored in a list and predecessor queries can obviously be answered in constant time. Otherwise, a data structure that answers membership queries $e' \in H(S)?$ in constant time is stored. Using hash functions, such a data structure can be stored in $O(n)$ space. A recursively defined data structure $(D)_u$ contains all elements of $H(S)$. For every $e' \in H(S)$ data structure $(D)_{e'}$ is stored; $(D)_{e'}$ contains all length $b/2$ suffixes of elements $e \in S$, such that e' is a prefix of e . Both D_u and all $D_{e'}$ contain keys in the range $[0, 2^{b/2} - 1]$. $(S)_u$ and $(S)_{e'}$ denote the sets of elements in $(D)_u$ and $(D)_{e'}$ respectively. For every node v of the global tree T that corresponds to an element stored in a data structure on some level, we store $v.min$ and $v.max$, the minimal and maximal leaf descendants of v in T . All elements of S are also stored in a doubly linked list, so that the predecessor $pred(x)$ of every element x in S can be found in constant time.

Suppose we are looking for a predecessor of $x \in [0, 2^b - 1]$. If $x^u \in H(S)$, we look for a predecessor of x^l in D_{x^u} . If x^l is smaller than all elements in D_{x^u} , the predecessor of x is $pred(x^u.min)$. If $x^u \notin H(S)$, the predecessor of x is $m.max$, where m is the node in T corresponding to the predecessor of x^u in D_0 . Using i levels of the above data structure a predecessor query with key length b can be reduced to a predecessor query with key length $b/2^i$ in $O(i)$ time. We will call data structures that contain keys of length $b/2^i$ level i data structures, and the corresponding sets of elements will be called level i sets.

It was shown before that if a word size w is bigger than bk , then predecessor queries can be answered in $O(\log n / \log k)$ time with help of packed B-trees of Andersson [A95] (see also [H98]). Using the van Emde Boas construction described above, we can reduce the key size from w to $w/2^{\sqrt{\log n}}$ in $O(\sqrt{\log n})$ time. After this, the predecessor can be found in $O(\log n / \sqrt{\log n}) = O(\sqrt{\log n})$ time ([A95]).

3 An $O(1)$ Amortized Time Data Structure

We start with a global overview of our algorithm. During the first stage of our algorithm $\sqrt{\log n}$ predecessor queries on keys $x_i \in [0, 2^b - 1]$ are reduced in a certain way to $\sqrt{\log n}$ predecessor queries in $[0, 2^{b/2^{\sqrt{\log n}}} - 1]$. The first phase is implemented using the van Emde Boas [E77] construction described in Section 2. But by performing multiple membership queries in spirit of [BF02] we can reduce the size of the keys for all elements of the query set in parallel. During the second stage we find the predecessors of $\sqrt{\log n}$ elements from $[0, 2^{b/2^{\sqrt{\log n}}} - 1]$. Since $2^{\sqrt{\log n}}$ elements can be now packed into one machine word, we can use the packed B-trees of Andersson and find the predecessor of an element of the query set in $O(\log n / \log(2^{\sqrt{\log n}})) = O(\sqrt{\log n})$ time. We follow the same approach, but we find the predecessors of *all* elements of the query set *in parallel*. This allows us to achieve $O(\sqrt{\log n})$ time for $\sqrt{\log n}$ elements, or $O(1)$ amortized time.

The main idea of the algorithm presented in this paper is to search for predecessors of $\sqrt{\log n}$ elements *simultaneously*. By performing multiple membership queries, as described in Lemma 1, the key size of $\sqrt{\log n}$ elements can be reduced from b to $b/2^{\sqrt{\log n}}$ in $O(\sqrt{\log n})$ time. When the size of the keys is sufficiently reduced, the predecessors of all elements in the query set can be quickly found. If key size $b < w/2^{\sqrt{\log n}}$, the packed B-tree of degree $2^{\sqrt{\log n}}$ can be used to find the predecessor of a single element in $O(\sqrt{\log n})$ time. In our algorithm, we use a similar approach to find predecessors of *all* $\sqrt{\log n}$ elements in $O(\sqrt{\log n})$ time.

In the following lemma we show, how $\sqrt{\log n}$ queries can be answered in $O(\sqrt{\log n})$ time, if the word size is sufficiently large, that is $w = \Omega(b \log n)$. Later in this section we will show that the same time bound can be achieved in the case $w = \Theta(b)$

Lemma 2. *If word size $w = \Omega(b \log n)$, where b is the size of the keys, there exists a data structure that answers $\sqrt{\log n}$ predecessor queries in $O(\sqrt{\log n})$ time, requires space $O(n^{2\sqrt{\log n}+2})$, and can be constructed in $O(n^{2\sqrt{\log n}+2})$ time.*

Proof. Suppose we look for predecessors of elements x_1, x_2, \dots, x_p with $p = \sqrt{\log n}$. The algorithm consists of two stages :

Stage 1. Range reduction. During this stage the size of all keys is simultaneously reduced by multiple look-ups.

Stage 2. Finding predecessors. When the size of the keys is small enough, predecessors of all keys can be found by multiple comparisons in packed B-trees.

Stage 1. We start by giving a high level description; a detailed description will

be given below. Since the word size w is $\log n$ times bigger than the key size b , $\sqrt{\log n}$ membership queries can be performed “in parallel“ in $O(1)$ time. Therefore, it is possible to reduce the key size by a factor 2 in $O(1)$ time *simultaneously* for all elements of the query set.

The range reduction stage consists of $\sqrt{\log n}$ rounds. During round j the key size is reduced from $b/2^{j-1}$ to $b/2^j$. By b' we denote the key size during the current round; $\langle u \rangle$ denotes the string of length b with value u .

Let $X = \langle x_1 \rangle \dots \langle x_q \rangle$ be a word containing all elements of the current query set. We set $X^1 = \langle x_1^1 \rangle \dots \langle x_q^1 \rangle$, where $x_i^1 = x_i$, and we set $S_i^1 = S$ for $i = 1, \dots, q$. During the first round we check whether prefixes of x_1, x_2, \dots, x_q of length $b/2$ belong to $H(S)$, i.e. we answer multiple membership query $(x_1^1)^u \in H(S_1^1)?, (x_2^1)^u \in H(S_2^1)?, \dots, (x_q^1)^u \in H(S_q^1)?$. If $(x_i)^u \notin H(S_i^1)$, $H(S_i^1)$ is searched for the predecessor of $(x_i)^u$, otherwise $S_{(x_i)^u}$ must be searched for the predecessor of $(x_i)^l$.

Now consider an arbitrary round j . At the beginning of the j -th round, we check whether some of the sets $S_1^j, S_2^j, \dots, S_q^j$ contain less than five elements. For every i , such that $|S_i^j| \leq 4$, $\langle x_i^j \rangle$ is deleted from the query set. After this we perform a multiple membership query $(x_1^j)^u \in H(S_1^j)?, (x_2^j)^u \in H(S_2^j)?, \dots, (x_q^j)^u \in H(S_q^j)?$. We set $X^{j+1} = \langle x_1^{j+1} \rangle \dots \langle x_q^{j+1} \rangle$, where $x_i^{j+1} = (x_i^j)^u$ if $x_i^j \notin H(S_i^j)$, otherwise $x_i^{j+1} = (x_i^j)^l$. $S_i^{j+1} = H(S_i^j)$, if $x_i^j \notin H(S_i^j)$, and $S_i^{j+1} = (S_i^j)_{(x_i)^u}$, if $x_i^j \in H(S_i^j)$.

Detailed Description of Stage 1. Words X^j consist of q words of size b for $q \leq \sqrt{\log n}$. Let *set tuple* $S_1^i, S_2^i, \dots, S_q^i$ be an arbitrary combination of sets of elements of level i data structures (the same set can occur several times in a set tuple). For every $q \in [1, \sqrt{\log n}]$ and every set tuple $S_{i_1}^j, S_{i_2}^j, \dots, S_{i_q}^j$, where $S_{i_k}^j$ are sets on level j , data structure $D(S_{i_1}^j, S_{i_2}^j, \dots, S_{i_q}^j)$ is stored. $D(S_{i_1}^j, S_{i_2}^j, \dots, S_{i_q}^j)$ consists of :

1. mask $M(S_{i_1}^j, S_{i_2}^j, \dots, S_{i_q}^j) \in [0, 2^q - 1]$. The $(q + 1 - t)$ -th least significant bit of $M(S_{i_1}^j, S_{i_2}^j, \dots, S_{i_q}^j)$ is 1, iff $|S_{i_t}^j| \leq 4$.
2. word $MIN(S_{i_1}^j, S_{i_2}^j, \dots, S_{i_q}^j) = \langle m_1 \rangle \langle m_2 \rangle \dots \langle m_q \rangle$, where $m_k = \min(S_{i_k}^j)$ is the minimal element in $S_{i_k}^j$
3. if $M(S_{i_1}^j, S_{i_2}^j, \dots, S_{i_q}^j) = 0$, data structure $L(S_{i_1}^j, S_{i_2}^j, \dots, S_{i_q}^j)$, which allows to answer multiple queries $x_1 \in H(S_{i_1}^j)?, x_2 \in H(S_{i_2}^j)?, \dots, x_q \in H(S_{i_q}^j)?$.
4. Array DEL with q elements; $DEL[t]$ contains a pointer to data structure $D(S_{i_1}^j, \dots, S_{i_{t-1}}^j, S_{i_{t+1}}^j, \dots, S_{i_q}^j)$.
5. Array $NEXT$ with less than $\prod_{k=1}^q 4|S_{i_k}^j|$ elements;

For every $F \in [0, 2^q - 1]$, list $LIST[F]$ is stored; $LIST[F]$ contains all indices i , such that the $(q + 1 - i)$ -th least significant bit of F is 1. We store a one-to-one hash function $c : \mathcal{C} \rightarrow [0, 2^q - 1]$, where \mathcal{C} is the set of integers $v \in [0, 2^{qb} - 1]$, such that the ib -th bit of v is either 1 or 0, and all other bits of v are 0. List $BACKLIST[F]$ contains all indices i , such that the $(q+1-i)b$ -th least significant bit of $c^{-1}(F)$ is 1.

Consider a round j , and suppose that the current set tuple is $S_1^j, S_2^j, \dots, S_q^j$. For every element i of $LIST[M(S_1^j, S_2^j, \dots, S_q^j)]$ we do the following:

1. x_i^j is extracted from X^j . We set $A := (X^j \ggg (b(q-i)))AND(1^{b'})$ and find the predecessor of A in S_i^j . The predecessor of x_i^j in S_i^j can be found in constant time, since $|S_i^j| \leq 4$.
2. We delete x_i^j from X^j by $X^j := (X^j AND 1^{(b(q-i))}) + ((X^j \ggg (q-i+1)b) \lll (q-i)b)$, and decrement q by 1.

Then we extract all x_k^j such that $x_k^j < MIN(S_k^j)$. We perform a multiple comparison of X^j with $MIN(S_{i_1}^j, S_{i_2}^j, \dots, S_{i_q}^j)$ and store the result in word C , such that the $(q+1-k)b$ -th bit of C is 1 if and only if $x_k^j < \min(S_k^j)$. Details will be given in the full version of this papers. We compute $f = c(C)$ and process every element of $BACKLIST[f]$ in the same way as elements of $LIST[F]$ were processed.

Now a multiple query $(x_1^j)^u \in H(S_1^j)?, (x_2^j)^u \in H(S_2^j)?, \dots, (x_q^j)^u \in H(S_q^j)?$. must be processed. We compute $X^j AND (0^{b-b'}1^{b'/2}0^{b'/2})^q$ and bit shift the result $b'/2$ bits to the right to get $(X^j)^u$. The resulting word $(X^j)^u$ consists of the prefixes of length $b'/2$ of elements $x_1^j, x_2^j, \dots, x_q^j$. Using $(X^j)^u$ and $L(S_1^j, S_2^j, \dots, S_q^j)$, query $(x_1^j)^u \in H(S_1^j)?, (x_2^j)^u \in H(S_2^j)?, \dots, (x_q^j)^u \in H(S_q^j)?$ can be answered in $O(1)$ time. The result is stored in word R such that the $(q+1-i)b$ -th least significant bit of R is 1, iff $(x_i^j)^u \in H(S_i^j)$, and all other bits of R are 0. We also construct word $(X^j)^l$ that consists of suffixes of $x_1^j, x_2^j, \dots, x_q^j$ of length $b'/2$. $(X^j)^l$ is computed by $(X^j)^l = X^j AND (0^{b-b'}1^{b'/2}1^{b'/2})^q$. We compute the words $R' = (R \ggg (b-1)) \times 1^b$ and $R'' = R' XOR 1^{qb}$. Now we can compute $X^{j+1} = (X^l AND R'') + (X^u AND R')$.

The pointer to the next data structure can be computed in a similar way. Let h_1, h_2, \dots, h_q be hash functions for the sets $S_1^j, S_2^j, \dots, S_q^j$. As shown in the proof of Lemma 1, word $P = h_1(x_1^j)h_2(x_2^j) \dots h_q(x_q^j)$ can be computed in constant time. For every such P , we store in $NEXT[P]$ a pointer to data structure $D(S_1^{j+1}, S_2^{j+1}, \dots, S_q^{j+1})$, such that $S_i^{j+1} = H(S_i^j)$, if $x_i^j \notin H(S_i^j)$, and $S_i^{j+1} = (S_i^j)_{(x_i)^u}$, if $x_i^j \in H(S_i^j)$. Array $NEXT$ has less than $\prod_{k=1}^q 4|S_{i_k}^j|$ elements.

After $\sqrt{\log n}$ rounds, the range of the key values is reduced to $[0, 2^{b/(2\sqrt{\log n})} - 1]$.

Stage 2. Finding Predecessors. Now we can find the predecessors of elements using the approach of packed B-trees (cf. [H98],[A95]). Since more than $\sqrt{\log n}2^{\sqrt{\log n}}$ keys fit into a machine word, each of current queried values can be compared with $2^{\sqrt{\log n}}$ values from the corresponding data structure. Hence after at most $\sqrt{\log n}$ rounds the search will be completed. In this paper we consider an extension of the approach of packed B-trees for a simultaneous search in several data structures, called a *multiple B-tree*.

Let $p = \sqrt{\log n}$ and $t = 2^{\sqrt{\log n}}$. Consider an arbitrary combination of level p sets $S_{i_1}^p, \dots, S_{i_q}^p$ and packed B-trees $T_{i_1}, T_{i_2}, \dots, T_{i_q}$ for these sets. Nodes of

T_{i_j} have degree $\min(2^{\sqrt{\log n}}, |S_{i_j}^p|)$. The root of a *multiple B-tree* contains all elements of the roots of packed B-trees for $S_{i_1}^p, \dots, S_{i_q}^p$. Every node of the multiple B-tree that contains nodes n_1, n_2, \dots, n_q has at most $(2^{\sqrt{\log n}})^q$ children, which correspond to all possible combinations of children of n_1, n_2, \dots, n_q (only non-leaf nodes among n_1, \dots, n_q are considered). Thus a node of the *multiple B-tree* on the level k is an arbitrary combination of nodes of packed B-trees for sets $S_{i_1}^p, \dots, S_{i_q}^p$ on level k . In every node v , word K_v is stored. If node v corresponds to nodes v_1, v_2, \dots, v_q of packed B-trees with values $v_1^1, \dots, v_1^t, v_2^1, \dots, v_2^t, \dots, v_q^1, \dots, v_q^t$ respectively, then $K_v = 0v_1^1 0v_2^1 \dots 0v_1^t \dots 0v_q^1 0v_2^1 \dots 0v_q^t$. Values v_i^1, \dots, v_i^t , for $i = 1, \dots, q$, are stored in K_v in an ascending order. In every node we also store an array *CHILD* with $2n$ elements. Besides that in every node v an array $DEL(v)[]$ and mask $M(v) \in [0, 2^{q+1} - 1]$ are stored; they have the same purpose as the array *DEL* and mask M in the first stage of the algorithm: the $(q + 1 - k)$ -th least significant bit of $M(v)$ is 1, iff the node of T_{i_k} stored in v is a leaf node. The height of the multiple B-tree is $O(\sqrt{\log n})$.

Now we show how every component of X can be compared with $2^{\sqrt{\log n}}$ values in constant time. Let $X = \langle x_1 \rangle \langle x_2 \rangle \dots \langle x_q \rangle$ be the query word after the completion of Stage 1. Although the length of $\langle x_i \rangle$ is b , the actual length of the keys is b' , and b' is less than $b/(2^{\sqrt{\log n}})$. Let $s = 2^{\sqrt{\log n}}(b' + 1)$, $s < b$. We construct the word $X' = \langle x_1 \rangle \langle x_2 \rangle \dots \langle x_q \rangle$, where $\langle x_i \rangle = (0 \ll x_i \gg)^{2^{\sqrt{\log n}}}$, and $\ll x_i \gg$ is a string of length b' with value x_i . To achieve this, we copy X , shift the copy $b' + 1$ bits to the left, and add the result to X . The result is copied, shifted $2b' + 2$ bits to the left, and so on. We repeat this $\sqrt{\log n}$ times to obtain $2^{\sqrt{\log n}}$ copies of each key value x_i .

To compare values stored in X' with values stored in node v , we compute $R = (K_v - X') \text{AND } W$, where W is a word every $(b' + 1)$ -th bit of which is 1, and all other bits are 0. Let \mathcal{R} be the set of possible values of R . Since values v_i^1, \dots, v_i^t are sorted, $|\mathcal{R}| = (2^{\sqrt{\log n}})^{\sqrt{\log n}} = n$. Hence, a hash function $r : \mathcal{R} \rightarrow [1, 2n]$ (one for all nodes) can be constructed. The search continues in a node $v' = \text{CHILD}[r(R)]$. Since the height of multiple B-tree is $O(\sqrt{\log n})$, predecessors are found in $O(\sqrt{\log n})$ time.

Space Analysis. First we analyze the space used during the Stage 1. In an arbitrary data structure $D(S_{i_1}^j, S_{i_2}^j, \dots, S_{i_q}^j)$, $L(S_{i_1}^j, S_{i_2}^j, \dots, S_{i_q}^j)$ and array *NEXT* use $O(\prod_{k=1}^q 4|S_{i_k}^j|)$ space, mask $M(S_{i_1}^j, S_{i_2}^j, \dots, S_{i_q}^j)$ uses constant space, and array *DEL* uses $O(q)$ space. Hence, $D(S_{i_1}^j, S_{i_2}^j, \dots, S_{i_q}^j)$ uses $O(\prod_{k=1}^q 4|S_{i_k}^j|)$ space. The total space for all data structures $D(S_{i_1}^j, S_{i_2}^j, \dots, S_{i_q}^j)$ is

$$O\left(\sum_{\substack{i_1 \in [1, g], \\ \vdots \\ i_q \in [1, g]}} \prod_{k=1}^q 4|S_{i_k}^j|\right) \tag{1}$$

where g is the total number of sets S_i^j . Since $S_1^j + S_2^j + \dots \leq n2^j$, the total number of terms in sum (1) does not exceed $(n2^j)^q$. Every product $\prod_{k=1}^q 4|S_{i_k}^j|$ is less than $n^q 2^{2q}$. Hence the sum (1) is smaller than $n^{2q} 2^{(j+2)q}$. Summing up by

$j = 1, \dots, \sqrt{\log n}$, we get $\sum_{q=1}^{\sqrt{\log n}} \sum_{j=1}^{\sqrt{\log n}} n^{2q} 2^{(j+2)q} \leq \sum_{q=1}^{\sqrt{\log n}} n^{2q} 2^{(\sqrt{\log n}+3)q}$. The last expression does not exceed $n^{2\sqrt{\log n}} 2^{(\sqrt{\log n}+3)\sqrt{\log n}+1} = O(n^{2\sqrt{\log n}+2})$. Therefore the total space used by all data structures in stage 1 is $O(n^{2\sqrt{\log n}+2})$.

Now consider a multiple B-tree for a set tuple $S_{i_1}^p, S_{i_2}^p, \dots, S_{i_q}^p$. Every leaf in this multiple B-tree corresponds to some combination of elements from $S_{i_1}^p, S_{i_2}^p, \dots, S_{i_q}^p$. Hence, the number of leaves is $O(\prod_{k=1}^q |S_{i_k}^p|)$, and the total number of nodes is also $O(\prod_{k=1}^q |S_{i_k}^p|)$. Using the same arguments as above, the total number of elements in all multiple B-trees can be estimated as $\sum_{q=1}^{\sqrt{\log n}} n^{2q} 2^{(p+2)q} = O(n^{2\sqrt{\log n}+2})$. Hence, the total space is $O(n^{2\sqrt{\log n}+2})$.

A data structure $D(S_{i_1}^j, S_{i_2}^j, \dots, S_{i_q}^j)$ used in stage 1 can be constructed in $O(\prod_{k=1}^q 4|S_{i_k}^j|)$ time. Hence, all $D(S_{i_1}^j, S_{i_2}^j, \dots, S_{i_q}^j)$ for fixed j and q can be constructed in $O(n^{2q} 2^{(j+2)q})$, and all data structures for the stage 1 can be constructed in $O(n^{2\sqrt{\log n}+2})$ time. A multiple B-tree for set tuple $S_{i_1}^p, S_{i_2}^p, \dots, S_{i_q}^p$ can be constructed in $O(\prod_{k=1}^q |S_{i_k}^p|)$ time. Therefore, all multiple B-trees can be constructed in $O(n^{2\sqrt{\log n}+2})$ time.

Now we consider the case when a machine word contains only b bits.

Theorem 1. *If word size $w = \Theta(b)$, where b is the size of the keys, there is a data structure that answers $\sqrt{\log n}$ predecessor queries in $O(\sqrt{\log n})$ time, requires space $O(n^{2\sqrt{\log n}+2})$, and can be constructed in $O(n^{2\sqrt{\log n}+2})$ time.*

Proof. Using the van Emde Boas construction described in Section 2, the key size can be reduced from b to $b/\log n$ in $\log \log n \sqrt{\log n}$ time. However, we can speed-up the key size reduction by multiple queries. When the key size is reduced to $k/\log n$, predecessors can be found using Lemma 2.

The key size reduction for elements x_1, x_2, \dots, x_q consists of $\log \log n + 1$ rounds. During the i -th round, the length of the keys b' is reduced from $b/2^{i-1}$ to $b/2^i$. Hence, during the i -th round $w/b' > 2^{i-1}$, and $2^{(i-1)/2}$ membership queries can be performed in constant time. Our range reduction procedure is similar to the range reduction procedure of Stage 1 of Lemma 2, but we do not decrease q if some data structure becomes small, and parameter q grows monotonously. Roughly speaking, the number of keys that are stored in a word and can be queried in constant time grows monotonously. For $q \leq 2^{(j-1)/2}$ and every $S_{i_1}^j, \dots, S_{i_q}^j$, where $S_{i_k}^j$ are arbitrary sets on level j , data structure $D(S_{i_1}^j, S_{i_2}^j, \dots, S_{i_q}^j)$ described in Lemma 2 is stored.

The range reduction consists of $\log \log n + 1$ rounds. At the beginning of round j , keys x_1^j, \dots, x_q^j are stored in $\sqrt{\log n}/2^{(j-1)/2}$ words. Let $t = 2^{(j-1)/2}$. For simplicity, we assume that all words X_i^j contain t keys during each round. Consider an arbitrary word $X_i^j = x_{(i-1)t+1}^j, x_{(i-1)t+2}^j, \dots, x_{it}^j$, where $i = 1, \dots, \sqrt{\log n}/2^{(j-1)/2}$. In the same way as in Lemma 2, words $(X_i^j)^u$ and $(X_i^j)^l$ can be computed. Using the corresponding data structure L , query $(x_1)^u \in H(S_{i_1}^j)?, (x_2)^u \in H(S_{i_2}^j)?, \dots, (x_q)^u \in H(S_{i_q}^j)?$ can be answered in constant time, and X^{j+1} can also be computed in constant time. At the end

of round j , such that $j \equiv 0 \pmod{2}$, elements are regrouped. That is, we duplicate the number of keys stored in one word. Since the key size has decreased by factor 4 during the two previous rounds, word X_i^j is of the form $0^{3b''} x_{(i-1)t+1}^j 0^{3b''} x_{(i-1)t+2}^j \dots 0^{3b''} x_{it}^j$, where $b'' = b'/4$ and $x_k^j \in \{0, 1\}^{b''}$. We construct for each X_i^j a word \tilde{X}_i^j of the form $x_{(i-1)t+1}^j x_{(i-1)t+2}^j \dots x_{it}^j$. First X_i^j is multiplied with $(0^{tb'} - 1)^t$ to get \bar{X}_i^j . Then we perform bitwise AND of \bar{X}_i^j with a word $(0^{tb'} 1^{b'})^t$ and store the result in \hat{X}_i^j . \hat{X}_i^j is of the form $x_{(i-1)t+1}^j 0^{tb'+3b''} x_{(i-1)t+2}^j 0^{tb'+3b''} \dots 0^{tb'+3b''} x_{it}^j$. We can obtain \tilde{X}_i^j from \hat{X}_i^j ; details will be provided in the full version of the paper. Finally, we duplicate the number of keys in a word by setting $X_i^{j+1} = (\tilde{X}_{2i}^j \ll tb') + \tilde{X}_{2i+1}^j$, for $i = 1, \dots, \sqrt{\log n}/2^{(j)/2+1}$.

Therefore, after every second round the number of words decreases by factor 2. The total number of operations is $2O(\sqrt{\log n}) \sum_{i=1}^{\lceil (\lceil \log \log n \rceil / 2) \rceil} \frac{1}{2^{i-1}} = O(\sqrt{\log n})$. Space requirement and construction time can be estimated in the same way, as in the proof of Lemma 2.

4 Other Results

In this section we describe several extensions and improvements of Theorem 1.

Theorem 2. *For $p(n) = O(\sqrt{\log n})$, there exists a data structure that answers $p(n)$ predecessor queries in time $O(\sqrt{\log n})$, uses space $O(n^{2p(n)+2})$, and can be constructed in time $O(n^{2p(n)+2})$*

Proof Sketch. The proof is analogous to the proof of Theorem 1, but query set Q contains $p(n)$ elements.

Corollary 1. *For any $\varepsilon > 0$, there exists a data structure that answers $\sqrt{\log n}$ predecessor queries in time $O(\sqrt{\log n})$, uses space $O(n^{\varepsilon\sqrt{\log n}})$, and can be constructed in time $O(n^{\varepsilon\sqrt{\log n}})$.*

Proof. Set $p(n) = (\varepsilon/2)\sqrt{\log n} - 4$ and apply Theorem 2.

Corollary 2. *For any $\varepsilon > 0$ and $p(n) = O(\sqrt{\log n})$, there exists a data structure that answers $p(n)$ predecessor queries in time $O(\sqrt{\log n})$, uses space $O(n^{\varepsilon p(n)})$, and can be constructed in time $O(n^{\varepsilon p(n)})$.*

If the key size b is such that $\log b = o(\sqrt{\log n})$ (i.e. $\log \log N = o(\sqrt{\log n})$), then a more space efficient data structure can be constructed.

Theorem 3. *For $p(N) = O(\log \log N)$, there exists a data structure that answers $p(N)$ predecessor queries in time $O(\log \log N)$, uses space $O(n^{2p(N)+2})$, and can be constructed in time $O(n^{2p(N)+2})$.*

Proof Sketch. The proof is analogous to the proof of Theorem 1, but query set Q contains $p(n)$ elements. We apply $\log \log N$ rounds of the Stage 1 (range reduction stage) from the proof of Theorem 1. After this, the current key size b' equals to 1 for all elements of the query set, and predecessors can be found in a constant time.

Corollary 3. *For any $\varepsilon > 0$, there exists a data structure that answers $\log \log N$ predecessor queries in time $O(\log \log N)$, uses space $O(n^{\varepsilon \log \log N})$, and can be constructed in time $O(n^{\varepsilon \log \log N})$.*

Corollary 4. *For any $\varepsilon > 0$ and $p(N) = O(\log \log N)$, there exists a data structure that answers $p(N)$ predecessor queries in time $O(\log \log N)$, uses space $O(n^{\varepsilon p(N)})$, and can be constructed in time $O(n^{\varepsilon p(N)})$.*

5 Conclusion

In this paper we constructed new data structures for predecessor queries. These data structures allow us to answer predecessor queries faster than the lower bound of [BF02] at the cost of higher space requirements.

Suppose that n elements are stored in data structure A in sorted order, and query set Q also contains n elements. Using an integer sorting algorithm (e.g. [H02]), we can sort n elements of query set Q in $O(n \log \log n)$ time, then merge them with elements of A , and find predecessors of elements from Q in $O(\log \log n)$ time per query.

An existence of a linear (or polynomial) space data structure, which can answer $p = o(n)$ queries in time $o(\sqrt{\log n / \log \log n})$ per query is an interesting open problem.

References

- [AL62] G. M. Adelson-Velskii, E.M. Landis, *An algorithm for the organization of information*, Dokladi Akademii Nauk SSSR, 146(2):1259-1262, 1962.
- [AFK84] M. Ajtai, M. L. Fredman, J. Komlòs, *Hash Functions for Priority Queues*, Information and Control 63(3): 217-225 (1984).
- [ABR01] S. Alstrup, G. S. Brodal, T. Rauhe, *Optimal static range reporting in one dimension*, STOC 2001, pp. 476-482.
- [A95] A. Andersson, *Sublogarithmic Searching without Multiplications*, FOCS 1995, pp. 655-663.
- [A96] A. Andersson, *Faster Deterministic Sorting and Searching in Linear Space*, FOCS 1996, pp. 135-141
- [AHNR95] A. Andersson, T. Hagerup, S. Nilsson, R. Raman, *Sorting in linear time?* STOC 1995, pp. 427-436.
- [AT00] A. Andersson, M. Thorup, *Tight(er) worst-case bounds on dynamic searching and priority queues*, STOC 2000, pp. 335-342.
- [AT02] A. Andersson, M. Thorup, *Dynamic Ordered Sets with Exponential Search Trees*, The Computing Research Repository (CoRR), cs.DS/0210006: (2002). Available at <http://arxiv.org/abs/cs.DS/0210006>.
- [BF02] P. Beame, F. E. Fich, *Optimal Bounds for the Predecessor Problem and Related Problems*, J. Comput. Syst. Sci. 65(1): 38-72 (2002).
- [BCKM01] A. Brodnik, S. Carlsson, J. Karlsson, J. I. Munro, *Worst case constant time priority queue*, SODA 2001, pp. 523-528.

- [CW79] L. Carter, M. N. Wegman, *Universal Classes of Hash Functions*. J. Comput. Syst. Sci. 18(2): 143-154 (1979).
- [E77] P. van Emde Boas, *Preserving Order in a Forest in Less Than Logarithmic Time and Linear Space*, Inf. Process. Lett. 6(3): 80-82 (1977)
- [EKZ77] P. van Emde Boas, R. Kaas, E. Zijlstra, *Design and Implementation of an Efficient Priority Queue*, Mathematical Systems Theory 10: 99-127 (1977)
- [FW94] M. L. Fredman, D. E. Willard, *Trans-Dichotomous Algorithms for Minimum Spanning Trees and Shortest Paths*, J. Comput. Syst. Sci. 48(3): 533-551 (1994).
- [H98] T. Hagerup, *Sorting and Searching on the Word RAM*, STACS 1998, pp. 366-398.
- [H02] Y. Han, *Deterministic sorting in $O(n \log \log n)$ time and linear space*, STOC 2002, pp. 602-608
- [GL01] B. Gum, R. Lipton, *Cheaper by the Dozen: Batched Algorithms*. 1st SIAM International Conference on Data Mining, 2001
Available at <http://www.math.grin.edu/~gum/papers/batched/>
- [M84] K. Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching*, Springer 1984.
- [PS80] W. J. Paul, S. Simon, *Decision Trees and Random Access Machines*, International Symposium on Logik and Algorithmic, Zürich, pp 331-340, 1980.
- [W83] D. E. Willard, *Log-Logarithmic Worst-Case Range Queries are Possible in Space $\Theta(N)$* , Inf. Process. Lett. 17(2): 81-84 (1983)
- [W84] D. E. Willard, *New Trie Data Structures Which Support Very Fast Search Operations*, J. Comput. Syst. Sci. 28(3): 379-394 (1984).
- [Y81] A. C.-C. Yao *Should Tables Be Sorted?*, J. ACM 28(3): 615-628 (1981).

An Algorithm for Node-Capacitated Ring Routing

András Frank, Zoltán Király, and Balázs Kotnyek

Egerváry Research Group of MTA-ELTE, Department of Operations Research,
Department of Computer Science, Communication Networks Laboratory,
Eötvös University, Pázmány P. s. 1/c. Budapest, Hungary, H-1117*
{frank, kiraly, kotnyekb}@cs.elte.hu

Abstract. A strongly polynomial time algorithm is described to solve the node-capacitated routing problem in an undirected ring network.

1 Introduction

Based on a theorem of Okamura and Seymour [7], the half-integer relaxation of the edge-capacitated routing problem in a ring network can be solved in polynomial time. In [2] it is described a sharpening of the theorem of Okamura and Seymour and this gave rise to a polynomial time algorithm for the routing problem in rings. In [3], the node-capacitated routing problem in ring networks was considered along with its fractional relaxation, the node-capacitated multicommodity flow problem. For the feasibility problem, Farkas' lemma provides a characterization for general undirected graphs asserting, roughly, that there exists such a flow if and only if the so-called distance inequality holds for every choice of distance functions arising from nonnegative node-weights. For ring networks, [3] improved on this (straightforward) result in two ways. First, independent of the integrality of node-capacities and demands, it was shown that it suffices to require the distance inequality only for distances arising from (0-1-2)-valued node-weights, a requirement called the double-cut condition. Second, for integral node-capacities and demands, the double-cut condition was proved to imply the existence of a half-integral multicommodity flow. An algorithm was also developed in [3] to construct a half-integer routing or a violating double-cut. A half-integer routing could then be used to construct a(n integral) routing which, however, may have slightly violated the node-capacity constraint: the violation at each node was proved to be at most one. Such a routing may be completely satisfactory from a practical point of view, especially when the capacities are big. Nevertheless the problem to decide algorithmically whether or not there is a routing remained open in [3].

In the present work we develop a strongly polynomial algorithm to construct a routing in a node-capacitated ring network, if there is one. The algorithm is based

* Research supported by the Hungarian National Foundation for Scientific Research, OTKA T037547 and by European MCRN Adonet, Contract Grant No. 504438. The authors are also supported by Ericsson Hungary.

on a rather tricky reduction to the edge-capacitated routing. The reduction can be carried out algorithmically with the help of the Fourier-Motzkin elimination. Though the FM-algorithm is not polynomial in general, we apply it to a specially structured matrix where it is even strongly polynomial. We include this version of the FM-algorithm in Section 4.

These problems are important from both theoretical and practical point of view. For example, telecommunication network routing problems form the main source of practical demand of this type of problems. In particular, the present work was originally motivated by engineering investigations in the area of so called passive optical networks.

Let $G = (V, E)$ be a simple undirected graph called a **supply** graph, let $H = (V, F)$ be a so-called **demand** graph on the same node set. Suppose that a nonnegative demand value $h(f)$ is assigned to every demand edge $f \in F$ and a nonnegative capacity $g(e)$ is assigned to every supply edge $e \in E$. We will say that an integer-valued function g is **Eulerian** if $d_g(v) := \sum[g(e) : e \text{ incident to } v]$ is even for each node v . By a **path** P we mean an undirected graph $P = (U, A)$ where $U = \{u_1, u_2, \dots, u_n\}$, $A = \{e_1, \dots, e_{n-1}\}$, and $e_i = u_i u_{i+1}$, $i = 1, \dots, n - 1$. The edge-set A of a path P is denoted by $E(P)$ while its node set by $V(P)$. Nodes u_1 and u_n are called the **end-nodes** of P while the other nodes of P are called **internal nodes** and their set is denoted by $I(P)$. We say that a path P **connects** its end-nodes and that P **uses** an edge e if $e \in E(P)$.

For a demand edge $f \in F$, let \mathcal{P}_f denote the set of paths of G connecting the end-nodes of f and let $\mathcal{P} := \cup(\mathcal{P}_f : f \in F)$. By a **path-packing** we mean a function $x : \mathcal{P} \rightarrow \mathbb{R}_+$. A path P for which $x(P) > 0$ is said to **belong to** or **determined by** x . We say that x **fulfills** or **meets** the demand if

$$\sum[x(P) : P \in \mathcal{P}_f] = h(f)$$

holds for every $f \in F$. In this case the path-packing x is also called a **routing** (of the demands). The **occupancy** $o_x(e)$ and $o_x(v)$ of a supply edge $e \in E$ and of a node $v \in V$ is defined, respectively, by

$$o_x(e) := \sum[x(P) : P \in \mathcal{P}, e \in E(P)] \quad \text{and}$$

$$o_x(v) := \sum[x(P) : P \in \mathcal{P}, v \in I(P)].$$

We stress that the paths ending at v are not counted in the occupancy of v . A path-packing x is called **feasible with respect to the edge-capacity**, or, in short, **edge-feasible** if

$$o_x(e) \leq g(e) \tag{1}$$

holds for every supply edge e .

Sometimes we are given a capacity function $c : V \rightarrow \mathbb{R}_+$ on the node set V rather than on E . A path-packing x is called **feasible with respect to node-capacity**, in short, **node-feasible** if

$$o_x(v) \leq c(v) \tag{2}$$

holds for every vertex $v \in V$. Inequality (1) and (2) are called, respectively, the **edge-** and the **node-capacity constraints**.

The **edge-** or **node-capacitated multicommodity flow** problem, respectively, consists of finding an edge- or node-feasible path-packing fulfilling the demand. It is sometimes called the **fractional routing** problem. If x is required to be integer-valued, we speak of an **integer multicommodity flow** problem or a **routing** problem (we also use the notion of **integral routing** for routing if we want to stress integrality). If $2x$ is required to be integer-valued, we speak of a **half-integral multicommodity flow** problem or a **half-integral routing** problem. If each demand and each capacity is one, and x is also required to be (0–1)-valued, then we speak of an **edge-disjoint** or **node-disjoint** paths problem. That is, the edge-disjoint (node-disjoint) paths problem can be formulated as deciding if there is a path in G for each demand edge $f \in F$ connecting its end-nodes so that these $|F|$ paths are edge-disjoint (internally node-disjoint).

By a **ring** (=cycle =circuit) we mean an undirected graph $G = (V, E)$ where $V = \{v_1, v_2, \dots, v_n\}$, $E = \{e_1, \dots, e_n\}$, and $e_i = v_i v_{i+1}$, $i = 1, \dots, n$. [Notation: $v_{n+1} = v_1$.] We will intuitively think that nodes of G are drawn in the plane in a counter-clockwise cyclic order. Note that for rings \mathcal{P}_f has only two paths for any $f \in F$. The edge-capacitated half-integral version were solved earlier by Okamura and Seymour [7], while its integer-valued counter-part by the first named author of the present work [2]. (Actually, both results concerned graphs more general than rings.)

The node-capacitated routing problem for rings is the main concern of the present work. The solvability of the half-integral routing problem was completely characterized in [3]. This gave rise to a sufficient condition for the solvability of the routing problem, while the one of finding a necessary and sufficient condition and of constructing an algorithm to compute an integral routing, if one exists, remained open. In this paper, based on some of the ideas of [3], we develop a fully combinatorial, strongly polynomial solution algorithm. Although a good characterization may be derived with the help of the algorithm, its present form is not particularly attractive and finding an appropriate simplification requires further research. We must note, that the node-capacitated routing problem is more general than the edge-capacitated one. To see this, put a new vertex on each edge, and assign the original capacity of the edge to this new vertex.

For a subset $X \subset V$, the set of edges of G with exactly one end-node in X is called a cut of G . The **capacity** of a cut (with respect to a capacity function g) is the sum of capacities of (supply) edges in the cut and is denoted by $d_g(X)$. The **demand** on a cut (with respect to a demand function h) is the sum of demands of demand edges with exactly one end-node in X and is denoted by $d_h(X)$.

2 The Edge-Cut Criterion for Edge-Capacitated Routing

A conceptually easy necessary condition for the solvability of the edge-capacitated multicommodity flow problem is the so-called **edge-cut criterion** which requires that the demand on every cut cannot exceed its capacity, that is,

$$d_g(X) \geq d_h(X) \tag{3}$$

for every subset $X \subseteq V$. Inequality (3) will be called the **edge-cut inequality**. An important special case when the edge-cut criterion is sufficient is due to H. Okamura and P. D. Seymour [7].

Theorem 1 (Okamura and Seymour). *Suppose that (i) the supply graph G is planar, (ii) the end-nodes of demand edges are all in the outer face of G , (iii) g and h are integer-valued and $g + h$ is Eulerian; then the edge-cut criterion is necessary and sufficient for the solvability of the edge-capacitated routing problem.*

It is known and not difficult to prove anyway that the edge-cut criterion holds if (3) is required only for those subsets X for which both X and $V - X$ induce a connected subgraph of G . Therefore the theorem of Okamura and Seymour specializes to rings as follows.

Corollary 2. *If G is a ring, g and h are integer-valued and $g + h$ is Eulerian, the edge-capacitated routing problem has an integral solution if and only if the edge-cut inequality*

$$g(e_i) + g(e_j) \geq L_h(e_i, e_j) \text{ holds whenever } 1 \leq i < j \leq n \tag{4}$$

where $L_h(e_i, e_j)$ denotes the total demand through the cut determined by e_i and e_j .

Note that there are polynomial algorithms [9][10][8] for this edge-capacitated ring routing problem. Based on algorithm in [9] the second named author developed an algorithm [4] with running time $O(n^2)$. Our method makes use of such an algorithm as a subroutine.

3 Node-Capacitated Ring Routing

Suppose that $G = (V, E)$ is a ring endowed with an integral node-capacity function $c : V \rightarrow \mathbb{Z}_+$. We assume that the demand graph $H = (V, F)$ is complete. Let $h : F \rightarrow \mathbb{Z}_+$ be an integral demand function. Our main goal is to describe an algorithm to construct a node-capacitated routing satisfying the demands, if one exists.

It is well-known that in digraphs the node-disjoint version of the Menger theorem can be derived from its edge-disjoint counterpart by a straightforward node-splitting technique. Therefore it is tempting to try to reduce the node-capacitated ring-routing problem to its well-solved edge-capacitated version. A transformation, however, which is as simple as the one in the Menger theorem, is unlikely to exist since in the edge-capacitated multicommodity flow problem in rings a simple edge-cut criterion is necessary and sufficient for the solvability, while the node-capacitated version was shown in [3] to require a significantly more complex necessary and sufficient condition, the so-called double-cut condition. Still, the approach of [3] showed that such a reduction, though not so

cheaply, is possible. A further refinement of that approach will be the key to the present algorithm.

We start with an easy simplification that has appeared already in [3]. For completeness, we include its proof.

Claim 3. *Given c and h , there is an Eulerian demand function h' so that the routing problem is solvable with respect to c and h if and only if it is solvable with respect to c and h' .*

Proof. Let v_i and v_{i+1} be two subsequent nodes in the ring. As a one-edge path connecting v_i and v_{i+1} has no inner node, increasing the demand between them does not affect the solvability of the routing problem.

If $d_h(v)$ is not even everywhere, then there are two nodes v_i and v_j so that both $d_h(v_i)$ and $d_h(v_j)$ are odd and so that $d_h(v_l)$ is even for each l with $i < l < j$. We can then increase by one the h -values on all demand edges $v_i v_{i+1}, v_{i+1} v_{i+2}, \dots, v_{j-1} v_j$ and this way both $d_h(v_i)$ and $d_h(v_j)$ get even while all other $d_h(v_l)$ -values remain even for $i < l < j$. By repeating this procedure, we obtain the desired Eulerian demand function h' . •

Therefore we will assume throughout that

$$h \text{ is Eulerian.} \tag{5}$$

Note that for an Eulerian demand function the demand on every cut is even.

Theorem 4. *Let h be Eulerian and $g : E \rightarrow \mathbb{Z}_+$ an integral edge-capacity function such that*

$$g \text{ is Eulerian,} \tag{6}$$

$$g \text{ satisfies the edge-cut criterion} \tag{7}$$

and

$$g(e_{i-1}) + g(e_i) \leq d_h(v_i) + 2c(v_i) \text{ for every } v_i \in V. \tag{8}$$

Then there is an edge-feasible integral routing (with respect to g) meeting the demand h , and every such routing is node-feasible. Conversely, if there is a node-feasible integral routing that meets h , then there is an Eulerian function $g : E \rightarrow \mathbb{Z}_+$ satisfying the edge-cut criterion and (8).

Proof. Let us start with the easier second part and let x be a node-feasible routing. For $e \in E$, define $g(e) := \sum[x(P) : e \in P \in \mathcal{P}]$, that is, intuitively, $g(e)$ denotes the number of paths using e . Obviously g satisfies the edge-cut criterion (as x is an edge-feasible routing with respect to g). Let v_i be any node of G and let $\alpha := \sum[x(P) : P \in \mathcal{P}, v_i \text{ is an inner node of } P]$. The paths counted in the sum use both e_i and e_{i-1} while the paths determined by x that end at v_i use exactly one of e_i and e_{i-1} . Therefore $g(e_{i-1}) + g(e_i) = d_h(v_i) + 2\alpha$, and this implies (8) since $\alpha \leq c(v_i)$. Since $d_h(v_i)$ is even it also follows that $g(e_{i-1}) + g(e_i)$ is even.

To see the first part, let g be a function satisfying the hypothesis of the theorem. As both g and h are Eulerian, so is $g + h$, and hence Corollary 2 ensures the existence of an edge-feasible routing x (with respect to edge-capacity function g .) Suppose now that x is an arbitrary edge-feasible routing. We want to show that x is node-feasible as well, with respect to node-capacity function c . To see this, let us consider an arbitrary node v_i .

Let α denote the sum of x -values of those paths ending at v_i and using e_i , and let β denote the sum of x -values of those paths ending at v_i and using e_{i-1} . Finally, let γ be the sum of x -values of those paths using v_i as an inner node.

Then $\alpha + \beta = d_h(v_i)$. We have $\gamma + \alpha \leq g(e_i)$ and $\gamma + \beta \leq g(e_{i-1})$. By combining these with (8), we obtain $\alpha + \beta + 2\gamma \leq g(e_{i-1}) + g(e_i) \leq d_h(v_i) + 2c(v_i) \leq \alpha + \beta + 2c(v_i)$, from which $\gamma \leq c(v_i)$ follows, that is, x is indeed node-feasible with respect to c .

The second part is obvious, if x is a node-feasible integral routing, then take $g(e_i) = o_x(e_i)$ for each i . •

By Theorem 4, the node-capacitated ring-routing problem can be reduced to the edge-capacitated case if one is able to compute the capacity function g described in the theorem. To this end, introduce a nonnegative variable $z(i)$ for each $g(e_i)$. Consider the inequality system

$$z(i - 1) + z(i) \leq d_h(v_i) + 2c(v_i) \text{ for every } v_i \in V \tag{9}$$

and

$$z(i) + z(j) \geq L_h(e_i, e_j) \text{ for every } 1 \leq i < j \leq n. \tag{10}$$

The parity requirement in Theorem 4 may be satisfied in two ways, since every righthand-side is even. Either each $z(i)$ is even or else each $z(i)$ is odd. The algorithm handles these alternatives separately.

The system (9) and (10) has a solution so that z is even-integer-valued if and only if the system

$$z'(i - 1) + z'(i) \leq (d_h(v_i) + 2c(v_i))/2 \text{ for every } v_i \in V \tag{11}$$

and

$$z'(i) + z'(j) \geq L_h(e_i, e_j)/2 \text{ for every } 1 \leq i < j \leq n \tag{12}$$

has a nonnegative integral solution, and $z(i) = 2z'(i)$.

The system (9) and (10) has a solution so that z is odd-integer-valued if and only if the system

$$z''(i - 1) + z''(i) \leq (d_h(v_i) + 2c(v_i) - 2)/2 \text{ for every } v_i \in V \tag{13}$$

and

$$z''(i) + z''(j) \geq (L_h(e_i, e_j) - 2)/2 \text{ for every } 1 \leq i < j \leq n \tag{14}$$

has a nonnegative integral solution, and $z(i) = 2z''(i) + 1$.

Both the inequality system described for z' and the one for z'' have the following form. The right-hand side is integral and each inequality contains at

most two variables. In each inequality the coefficients of the variables has absolute value one. The integral solvability of such an inequality system can be decided in strongly polynomial time by a straightforward modification of the Fourier-Motzkin elimination algorithm. Therefore with two separate applications of the FM-algorithm we are able to decide if any of the systems $\{(11),(12)\}$ and $\{(13),(14)\}$ has a nonnegative integral solution and compute one if it exists. If none of these systems has such a solution then we may conclude that the original node-capacitated ring-routing problem has no integral solution. If one of them has a solution, we can calculate the appropriate (either odd-integer-valued or even-integer-valued) vector z satisfying inequalities (9) and (10), and determine an integral routing with respect to edge-capacities $g(e_i) = z(i)$. We may use the $O(n^2)$ algorithm of [4] for this purpose, so together with the FM algorithm described in the next section the total running time is $O(n^3)$. By Theorem 4 such an edge-feasible routing exists and it is also node-feasible.

For completeness, we include the original FM-algorithm and then we derive its strongly polynomial variation for “simple” matrices.

4 Fourier-Motzkin Elimination

The Fourier-Motzkin elimination is a finite algorithm to find a solution to a linear inequality system $Qx \leq b$, that is, to find an element of the polyhedron $R := \{x \in \mathbb{R}^n : Qx \leq b\}$. It consists of two parts. In the first part, it eliminates the components of x one by one by creating new inequalities. In the second part, it proceeds backward and computes the components of a member of R . Geometrically, one elimination step may be interpreted as determining the polyhedron obtained from R by projecting along a coordinate axis.

Let Q be a $m \times n$ matrix ($m \geq 1, n \geq 2$). For any index set L of the rows of Q let ${}_LQ$ denote the corresponding submatrix of Q . The i 'th row of Q is denoted by iq . In order to find a solution to the system $Qx \leq b$, we may assume that the first column q_1 of Q is $(0, \pm 1)$ -valued since multiplying an inequality by a positive constant does not effect the solution set. Let I, J, K denote the index sets of rows of Q for which the value $q_1(i)$ is $+1, -1$ or 0 , respectively. Define a matrix $Q^{[1]}$ which contains all rows of ${}_KQ$, as follows. For every choice of indices $i \in I$ and $j \in J$, let $iq + jq$ be a row of $Q^{[1]}$ and let this row be denoted by ${}_{[ij]}q$. This means that in case I or J is empty, $Q^{[1]}$ is simply ${}_KQ$. In general $Q^{[1]}$ has $m - (|I| + |J|) + |I||J|$ rows. Note that the first column of $Q^{[1]}$ consists of zeros. The right-hand side vector $b^{[1]}$ is obtained analogously from b . Let $R^{[1]} := \{x : x(1) = 0, Q^{[1]}x \leq b^{[1]}\}$.

Proposition 5. *The projection of R along the first coordinate is $R^{[1]}$, that is, by turning zero the first component of any solution to*

$$Qx \leq b \tag{15}$$

yields a solution to

$$Q^{[1]}x \leq b^{[1]}, \tag{16}$$

and conversely, the first coordinate of any solution to (16) may be suitably changed in order to get a solution to (15).

Proof. The first part follows directly from the construction of $Q^{[1]}$ and $b^{[1]}$ since every row of $(Q^{[1]}, b^{[1]})$ is a nonnegative combination of the rows of (Q, b) .

To see the second part, let z be a solution to (16). For a number α , let z_α denote the vector arising from z by changing its first component to α . If J is empty, that is, the first column of Q has no negative element, then by choosing α small enough, z_α will be an element of R . (Namely, $\alpha := \min_{i \in I} \{b(i) - i q \cdot z\}$ will do.) Analogously, if I is empty, then α may be chosen suitably large. Suppose now that neither I nor J is empty. For any $i \in I, j \in J, i q \cdot z + j q \cdot z = [ij] q \cdot z \leq b(i) + b(j)$ implies that $j q \cdot z - b(j) \leq b(i) - i q \cdot z$ and hence

$$\max_{j \in J} \{j q \cdot z - b(j)\} \leq \min_{i \in I} \{b(i) - i q \cdot z\}. \tag{17}$$

Therefore there is a number α with

$$\max_{j \in J} \{j q \cdot z - b(j)\} \leq \alpha \leq \min_{i \in I} \{b(i) - i q \cdot z\}. \tag{18}$$

We claim that vector z_α is a solution to (15). Indeed, for an index $h \in K$, the first component of ${}_h q$ is zero, so ${}_h q \cdot z_\alpha = {}_h q \cdot z \leq b(h)$. If $h \in I$, that is, ${}_h q(1) = 1$, then the second inequality in (18) implies ${}_h q \cdot z_\alpha = {}_h q \cdot z + \alpha \leq b(h)$. Finally, if $h \in J$, that is, ${}_h q(1) = -1$, then the first inequality of (18) implies ${}_h q \cdot z_\alpha = {}_h q \cdot z - \alpha \leq b(h)$. •

Fourier-Motzkin Elimination for Simple Matrices

Let us call a $(0, \pm 1)$ -valued matrix Q **simple** if each row contains at most two nonzero entries and the rows are distinct. Note that a simple matrix with n columns can have at most $2n + 2 \cdot n(n - 1)/2 + n(n - 1) + 1 = 2n^2 + 1$ rows.

We show how the FM-elimination above may easily be turned to a strongly polynomial algorithm for computing an integral solution of $Qx \leq b$ in case Q is simple and b is integral.

As Q is simple, each row of $Q^{[1]}$ has at most two nonzero entries. If a row of $Q^{[1]}$ has exactly one nonzero element, then this element is ± 2 or ± 1 , while a row with two nonzero entries is $(0, \pm 1)$ -valued. $Q^{[1]}$ is not necessarily simple but it may easily be simplified without changing the integral solution set as follows.

First replace any inequality of type $2x(1) \leq \beta$ or $-2x(1) \leq \beta$ by $x(1) \leq \lfloor \beta/2 \rfloor$ or $-x(1) \leq \lfloor \beta/2 \rfloor$, respectively. Then, if some inequalities with identical lefthand sides remained, then we keep only the one defining the strongest inequality (that is, for which the corresponding righthand side is the smallest). Let $Q^{[2]}$ and $b^{[2]}$ denote the derived lefthand and righthand sides respectively.

Theorem 6. *Any integral solution to*

$$Qx \leq b \tag{19}$$

yields an integral solution to

$$Q^{[2]}x \leq b^{[2]}, \tag{20}$$

and conversely, the first coordinate of any integral solution to (20) may be suitably changed to another integer α in order to get an integral solution to (19).

Calculating $Q^{[2]}$ and $b^{[2]}$ as well as determining the appropriate α value takes $O(n^2)$ steps.

Proof. We follow the lines of the proof of Proposition 5. The first part follows again from the construction. For the second part one must observe that as the righthand and lefthand side of inequality (18) are now integers, we may choose α to be also an integer.

With appropriate organization, one elimination step as well as determining the appropriate α value can be carried out in $O(n^2)$ steps. Instead of storing Q and b we store three matrices PP, PM and MM . Set $PP(k, l) = b(i)$ if there is a row iq in Q , where $Q(i, k) = 1$ and $Q(i, l) = 1$ and $Q(i, j) = 0$ for $j \neq k, j \neq l$; and set $PP(k, l) = \infty$ if no such row exists. Similarly, $PM(k, l) = b(i)$ if there is a row iq in Q , where $Q(i, k) = 1, Q(i, l) = -1$ (and set $PM(k, l) = \infty$ otherwise); and $MM(k, l) = b(i)$ if there is a row iq in Q , where $Q(i, k) = -1, Q(i, l) = -1$ (and set $MM(k, l) = \infty$ otherwise). Observe, for example, that indices in J can be determined examining the first column of MM and of PM . Using these quantities, $PP^{[2]}, PM^{[2]}$ and $MM^{[2]}$ corresponding to $Q^{[2]}$ and $b^{[2]}$ can be easily calculated in $O(n^2)$ time. Determining α in $O(n^2)$ is straightforward (as in $O(n^2)$ time Q and b can be recovered from PP, PM and MM). •

Summarizing, the overall complexity of the FM-elimination for simple matrices can be carried out in $O(n^3)$ steps.

5 Conclusion

The edge-capacitated routing problem (that is, the integral multicommodity flow problem) was solved earlier by Okamura and Seymour in the special case when the supply graph is planar, the terminal nodes are in one face, and $d_g + d_h$ is even integer-valued. The algorithmic answer is particularly simple in the special case when the supply graph is a circuit (ring). The node-capacitated routing problem for a ring was considered in [3] where a characterization and an algorithm was given for half-integer solvability. The present work describes a strongly polynomial time algorithm for the routing problem in ring networks. The algorithm consists of first determining a suitable edge-capacity function g so that the edge-capacitated routings with respect to g are exactly the node-capacitated routings. Such a g could be computed with a variation of the Fourier-Motzkin elimination for simple matrices in $O(n^3)$ time. Next, calculating an edge-feasible routing with respect to function g is a relatively easy task (in $O(n^2)$ time by the algorithm of [4]).

References

1. L.R. Ford and D.R. Fulkerson, *Flows in Networks*, Princeton Univ. Press, Princeton NJ. (1962)
2. A. Frank, *Edge-disjoint paths in planar graphs*, J. of Combinatorial Theory, Ser. B. No. 2 (1985), pp. 164-178.

3. A. Frank, B. Shepherd, V. Tandon, and Z. Végh, *Node-capacitated ring routing*, Mathematics of Operations Research, 27, 2. (2002) pp. 372-383.
4. Z. Király, *An $O(n^2)$ algorithm for ring routing*, Egres Technical Reports, TR-2005-10, see <http://www.cs.elte.hu/egres/>, (2005).
5. R.H. Möhring and D. Wagner, *Combinatorial Topics in VLSI Design*, in: Annotated Bibliographies in Combinatorial Optimization, (M. Dell' Amico, F. M. Maffioli, S. Martello, eds.), John Wiley, (1997) pp. 429-444.
6. K. Onaga and O. Kakusho, *On feasibility conditions of multicommodity flows in Networks*, IEEE Trans. Theory, Vol. 18 (1971) pp. 425-429.
7. H. Okamura and P.D. Seymour, *Multicommodity flows in planar graphs*, J. Combinatorial Theory, Ser. B, 31 (1981) pp. 75-81.
8. H. Ripphausen-Lipa, D. Wagner and K. Weihe, *Survey on Efficient Algorithms for Disjoint Paths Problems in Planar Graphs*, DIMACS-Series in Discrete Mathematics and Theoretical Computer Science, Volume 20 on the "Year of Combinatorial Optimization" (W. Cook and L. Lovász and P.D. Seymour eds.), AMS (1995) pp. 295-354.
9. A. Schrijver, P. Seymour, and P. Winkler, *The ring loading problem*, SIAM J. Discrete Math. Vol. 11, No 1. (1998) pp. 1-14.
10. D. Wagner and K. Weihe, *A linear-time algorithm for edge-disjoint paths in planar graphs*, Combinatorica 15 (1995) pp. 135-150.

On Degree Constrained Shortest Paths

Samir Khuller*, Kwangil Lee, and Mark Shayman**

¹ Department of Computer Science, University of Maryland,
College Park, MD 20742, USA
Tel: (301) 405-6765, Fax: (301) 314-9658
samir@cs.umd.edu

² Institute for Advanced Computer Studies, University of Maryland,
College Park, MD 20742, USA
kilee88@yahoo.com

³ Department of Electrical and Computer Engineering, University of Maryland,
College Park, MD 20742, USA
Tel: 301-405-3667, Fax: 301-314-9281
shayman@glue.umd.edu

Abstract. Traditional shortest path problems play a central role in both the design and use of communication networks and have been studied extensively. In this work, we consider a variant of the shortest path problem. The network has two kinds of edges, “actual” edges and “potential” edges. In addition, each vertex has a degree/interface constraint. We wish to compute a shortest path in the graph that maintains feasibility when we convert the potential edges on the shortest path to actual edges. The central difficulty is when a node has only one free interface, and the unconstrained shortest path chooses two potential edges incident on this node. We first show that this problem can be solved in polynomial time by reducing it to the minimum weighted perfect matching problem. The number of steps taken by this algorithm is $O(|E|^2 \log |E|)$ for the single-source single-destination case. In other words, for each v we compute the shortest path P_v such that converting the potential edges on P_v to actual edges, does not violate any degree constraint. We then develop more efficient algorithms by extending Dijkstra’s shortest path algorithm. The number of steps taken by the latter algorithm is $O(|E||V|)$, even for the single-source all destination case.

1 Introduction

The shortest path problem is a central problem in the context of communication networks, and perhaps the most widely studied of all graph problems. In this paper, we study the *degree constrained shortest path problem* that arises in the context of dynamically reconfigurable networks. The objective is to compute shortest paths in the graph, where the edge set has been partitioned into two classes, such that for a specified subset of vertices, the number of edges on the

* Research supported by NSF grants CCR-0113192 and CCF-0430650.

** Research partially supported by AFOSR under contract F496200210217.

path that are incident to it from one of the classes is constrained to be at most one.

This work is motivated by the following application. Consider a free space optical (FSO) network [14]. Each node has a set of D laser transmitters and D receivers. If nodes i and j are in transmission range of each other, a transmitter from i can be pointed to a receiver at j and a transmitter from j can be pointed to a receiver at i , thereby creating a *bidirectional* optical communication link between i and j . If i and j are within transmission range of each other, we say that a *potential link* exists between them. If there is a potential link between i and j and they each have an unused transmitter/receiver pair, then an *actual link* can be formed between them. We will refer to a transmitter/receiver pair on a node as an *interface*. Thus, a potential link can be converted to an actual link if each of the nodes has an available interface.

We consider a sequential topology control (design) problem for a FSO network. The network initially consists entirely of potential links. Requests arrive for communication between pairs of nodes. Suppose that shortest path routing is used. When a request arrives for communication between nodes v_s and v_t , a current topology exists that consists of the actual links that have thus far been created, along with the remaining potential links. We wish to find a shortest path in this topology consisting of actual and potential links with the property that any potential link on the path can be converted to an actual link. This means that if the path contains a potential link from node i to node j , i and j must each have a free interface. Therefore, when searching for a shortest path, we can delete all potential links that are incident on a node that has no free interfaces. However, deleting these potential links still does not reduce the routing problem to a conventional shortest path problem. This is because if the path contains a pair of consecutive potential links (i, j) , (j, k) , the intermediate node j must have at least two free interfaces[7].

As an example, suppose the current topology is given in Figure 1(a) where the solid lines are actual links and the dotted lines are potential links. Suppose each node has a total of two interfaces. If a request arrives for a shortest path between nodes 1 and 7, the degree constraint rules out the path $1-6-7$ because node 6 has only one free interface. The degree constrained shortest path from 1 to 7 is $1-2-3-4-5-6-7$.

In addition to showing that the degree constrained shortest path problem cannot be reduced to a conventional shortest path problem by deleting potential links incident on nodes without free interfaces, the example illustrates two other features of this problem. Firstly, the union of the set of constrained shortest paths originating at a node need not form a tree rooted at that node. For all shortest paths from node 1, other than to node 7, we can construct a shortest path tree as shown in Figure 1(b). However, node 7 cannot be added to this tree. Secondly, since the constrained shortest path from 1 to 6 is the single hop path $1-6$ and not $1-2-3-4-5-6$, it follows that a sub-path of a constrained shortest path need not be a (constrained) shortest path.

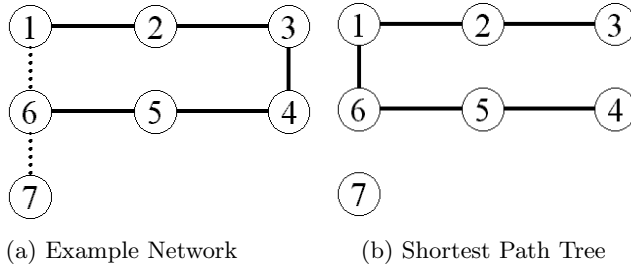


Fig. 1. Degree Constrained Shortest Path Problem

The problem is formally described as follows. We have a network with two kinds of edges (links), ‘actual’ and ‘potential’. We refer to these edges as *green* and *red* edges respectively. We denote green edges by S and red edges by R . Let $E = S \cup R$, where E is the entire edge set of the graph. We are required to find the shortest path from v_s to v_t . Edge e_{ij} denotes the edge connecting v_i and v_j . The weight of edge e_{ij} is w_{ij} . Green edges S represent actual edges, and red edges R represent potential edges. We denote a path from v_s to v_t by $P_t = \{v_s, v_1, \dots, v_t\}$ and its length $|P_t|$. The problem is to find a shortest path P_t^* from a source node v_s to a destination node v_t . However, each vertex has a degree constraint that limits the number of actual edges incident to it by D . If there are D green edges already incident to a vertex, then no red edges incident to it may be chosen, and all such edges can safely be removed from the graph. If the number of green edges is $\leq D - 2$ then we can choose up to two red edges incident to this vertex. In this case, a shortest path is essentially unconstrained by this vertex, as it can choose up to two red edges. The main difficulty arises when a vertex already has $D - 1$ green edges incident to it. In this case, at most one red edge incident to this vertex may be chosen. Hence, if a shortest path were to pass through this vertex, the shortest path must choose at least one green edge incident to this vertex.

In this paper, we study algorithms for the *Degree Constrained Shortest Path problem*. We propose two algorithms for finding a shortest path with degree constraints. First, we show how to compute a shortest paths between a pair of vertices by employing a perfect matching algorithm. However, in some cases we wish to compute single source shortest paths to all vertices. In this case, the matching based algorithm is slow as we have to run the algorithm for every possible destination vertex. The second algorithm is significantly faster and extends Dijkstra’s shortest path algorithm when all edge weights are non-negative. The rest of this paper is organized as follows. In Section 2, we show how to use a perfect matching algorithm to compute the shortest path. The complexity of this algorithm is $O(|E|^2 \log |E|)$ from a source to a single destination. In Section 3 we propose an alternate shortest path algorithm by extending Dijkstra’s algorithm. We introduce the degree constrained shortest path algorithm in Section 4. Section 5 analyzes and compares the complexity of these algorithms. Its complexity is $O(|E||V|)$ from a source not only to one destination but to all destinations.

Finally, we conclude the paper in Section 6. *We would also like to note that even though we describe the results for the version where all the nodes have the same degree constraint of D , it is trivial to extend the algorithm to the case when different nodes have different degree constraints.*

While the shortest path problem with degree constraints has not been studied before, considerable amount of work has been done on the problems of computing bounded degree spanning trees for both weighted and unweighted graphs. In this case, the problems are *NP*-hard and thus the focus of the work has been on the design of approximation algorithms [4,10,3]. In addition, Gabow and Tarjan [6] addressed the question of finding a minimum weight spanning tree with one node having a degree constraint.

2 Perfect Matching Approach

One solution for the degree constrained shortest path problem is based on a reduction to the minimum weight perfect matching problem. We define an instance of a minimum weight perfect matching problem as follows. Each node v has a constraint that at most δ_v red edges can be incident on it from the path. If node v has D green edges incident on it, then $\delta_v = 0$. When node v has $D - 1$ green edges, then $\delta_v = 1$; otherwise, $\delta_v = 2$. When $\delta_v = 0$ we can safely delete all red edges incident to v . We now reduce the problem to the problem of computing a minimum weight perfect matching in a new graph G' . For each vertex $x \in V - \{v_s, v_t\}$ we create two nodes x_1 and x_2 in G' , and add a zero weight edge between them. We retain v_s and v_t as they are.

For each edge $e_{xy} \in E$ we create two new vertices $v_{e_{xy}}$ and $v_{e'_{xy}}$ (called edge nodes) and add a zero weight edge between them. We also add edges from $v_{e_{xy}}$ to x_1 and x_2 , each of these has weight $w_{xy}/2$. When $x = v_s$ or $x = v_t$, we simply add one edge from $v_{e_{xy}}$ to x . We also add edges from $v_{e'_{xy}}$ to y_1 and y_2 , with each such edge having weight $w_{xy}/2$. Finally, for any red edges (u, x) and (x, y) with $\delta_x = 1$ we delete the edges from $v_{e'_{ux}}$ to x_1 and from $v_{e_{xy}}$ to x_1 .

Theorem 1. *A minimum weight perfect matching in G' will yield a shortest path in G connecting v_s and v_t with the property that for any vertex v on the path, no more than δ_v red edges are incident on v .*

The running time of the minimum weight perfect matching algorithm is $O(|V'|(|E'| + |V'| \log |V'|))$ [5][2]. Since $|V'|$ is $O(|V| + |E|)$ and $|E'|$ is $O(|E| + |V|)$ we get a running time of $O(|E|^2 \log |E|)$ (we assume that $|E| \geq |V|$, otherwise the graph is a forest, and the problem is trivial).

3 Shortest Path Algorithm

In this section, we develop an algorithm for finding degree constrained shortest paths using an approach similar to Dijkstra's shortest path algorithm.

3.1 Overview of the Algorithm

In Dijkstra’s algorithm, shortest paths are computed by setting a label at each node. The algorithm divides the nodes into two groups: those which it designates as permanently labeled and those that it designates as temporarily labeled. The distance label d of any permanently labeled node represents the shortest distance from the source to that node. At each iteration, the label of node v is its shortest distance from the source node along a path whose internal nodes are all permanently labeled. The algorithm selects a node v with a minimum temporary label, makes it permanent, and reaches out from that node, i.e., scans all edges of the node v to update the distance labels of adjacent nodes. The algorithm terminates when it has designated all nodes as permanent.

Let Z denote the set of nodes v_x satisfying $\sum_{e_{xy} \in S} e_{xy} = D - 1$. In other words, Z is the set of nodes where there is only one interface available for edges in R . Hence any shortest path passing through $v_x \in Z$ must exit on a green edge if it enters using a red edge. If the shortest path to v_x enters on a green edge, then it can leave on any edge. We start running the algorithm on the input graph with red *and* green edges. However, if a shortest path enters a vertex using a red edge, and the node is in Z then we mark it as a “critical” node. We have to be careful to break ties in favor of using green edges. In other words, when there are multiple shortest paths, we prefer to choose the one that ends with a green edge. So a path terminating with a red edge is used only if it is strictly better than the shortest known path. A shortest path that cuts through this node may not use two red edges in sequence. Hence, we only follow edges in S (green edges) out of this critical vertex. In addition, we create a *shadow* node v'_x for each critical node v_x . The shadow node is a vertex that has directed edges to all the red neighbors of v_x , other than the ones that have been permanently labeled. The shadow node’s distance label records the length of a valid (alternate) shortest path to v_x with the constraint that it enters v_x on a green edge. (Note that the length of this path is strictly greater than the distance label of v_x , due to the tie breaking rule mentioned earlier.)

Let C denote the set of critical nodes. When $v_c \in C$, then only edges in S leaving v_c may be used to get a valid shortest path through v_c . To reach a neighbor v_j such that $e_{cj} \in R$, v_c needs an alternate shortest path p_{sc}^+ , where $\text{parent}(v_c) = v_{p'}$, $v_{p'} \in p_{sc}^+$ and $e_{p'c} \in S$. We re-define the degree constraint shortest path problem as a shortest path problem which computes shortest valid paths for all nodes in V , and alternate shortest valid paths for nodes in C .

In fact, the edges from v'_c to the red neighbors of v_c are *directed* edges so they cannot be used to obtain a shortest path to v'_c . The corresponding critical node v_c retains its neighbors with green edges, and the red edge to its parent.

The set $\text{ancestor}(v_i)$ is the set of all vertices on a shortest valid path from v_s to v_i , including the end-nodes of the path. Let V be the set of vertices in the graph, and V' be the set of shadow vertices that were introduced.

Definition 1. Let $v_i, v_j \in V \cup V'$ and $v_c \in C$ with $(e_{ij} \in S) \vee (e_{ij} \in R \wedge v_i, v_j \notin C)$. A node pair $(v_i, v_j) \in \text{gateway}(c)$ if $v_c \in \text{ancestor}(v_i)$, and $v_c \notin \text{ancestor}(v_j)$.

This means that there always exists a possible alternate path to a critical node through a gateway node pair since P_j^* (shortest path from source to j) does not include the critical node v_c . If v_i 's neighbor v_j is a critical node and it is connected to it with a red edge, then this node pair cannot be a legal gateway node pair. A similar problem occurs when v_i is critical and the edge connecting them is red.

A virtual link $e'_{jc} = (v_j, v'_c)$ is created for inverse sub-path $(p_{ci}^* \cup \{e_{ij}\})^{-1}$, where $(v_i, v_j) \in gateway(c)$ (p_{ci}^* is the portion of the shortest path P_i^*). With shadow nodes and virtual links, we can maintain the data structure uniformly.

When nodes are labeled by the algorithm, it is necessary to check if these nodes form a gateway node pair or not. For this purpose, we define a data structure called CL , the Critical node List. For all nodes, $v_c \in CL(i)$ if $v_c \in ancestor(v_i)$ and $v_c \in C$. So, $CL(i)$ maintains all critical node information along the shortest path from v_s to v_i . By comparing $CL(i)$ and $CL(j)$, we know a node pair (v_i, v_j) is a gateway pair for v_c if $(v_c \in CL(i) \text{ and } v_c \notin CL(j))$.

3.2 Algorithm

The degree constrained shortest path algorithm using critical node sets is developed in this sub-section. The algorithm itself is slightly involved, and the example in the following section will also be useful in understanding the algorithm. In fact, the algorithm modifies the graph as it processes it. We also assume that the input graph is connected. Step 1 is to initialize the data structure. The difference with Dijkstra's algorithm is that it maintains information on two shortest paths for each vertex according to link type. Step I.1 initializes the data structure for each node. We use $pred$ to maintain predecessor (parent node) information in the tree. Step I.2 is for the initialization of the source. The label value is set to 0. Step I.3 is for the initialization of the permanently labeled node set (P) and the queue for the shortest path computation (Q).

Step 2 consists of several sub-steps. First, select a vertex node with minimum label in Q (Step 2.1) and permanently label it (Step 2.2). The label of each node $d[y]$ is chosen as the minimum of the green edge and red edge labels. If the green edge label is less than or equal to the red edge label, we select the path with the green edge and its critical node set (Step 2.3.1). Otherwise, we choose a red edge path (Step 2.4.1). If the path with the red edge is shorter than that with the green edge and the number of green edges incident on v_y is $D - 1$ ($v_y \in Z$), then v_y becomes a critical node (Step 2.4.2.1). In Step 2.5, we define CL ; its initial value is the same as its parent node. After the node is identified as a critical node, then CL will be changed later (Step 2.6.2).

The decision of whether a node is a critical node or not is made when a node is permanently labeled. When a node v_y is identified as a critical node, a shadow node v'_y is created as shown in Steps 2.6.3 through 2.6.6. In Step 2.6 we also choose the neighbors according to the node type. A critical node has neighbors with green edges and its shadow node has directed edges to the neighbors to which the critical node had red edges, if the neighbor has not been permanently labeled as yet (Step 2.6.7). Otherwise, the node can have neighbors with any

type of edges (Step 2.7.1). However, Step 2.7.1 specifies an exception to not add neighbors to which there is a red edge if the neighbor is a critical node. Consider a situation when a non-critical node v_i has a red edge to a critical node v_c . Since v_c is a critical node, v_c is already permanently labeled. Note that v_i cannot be a neighbor of v_c , but a neighbor of v'_c , by definition. Later, when v_i is permanently labeled and is a non-critical node, we do not wish to have v_c in v_i 's list. For this reason, we check if a neighbor node with a red edge is a critical node or not. Since v_i is permanently labeled, there is no shorter path in the graph including path through v'_c . All legal neighbor information is maintained by data structure adj . Note that the graph we construct is actually a *mixed* graph with directed and undirected edges, even though the input graph was undirected.

Step 2.8 examines the neighbors of a permanently labeled node. It consists of two parts. Step 2.8.3 updates labels for all neighbor nodes in adj . This procedure is similar to Dijkstra's algorithm. The only difference is that label update is performed based on link type. Step 2.8.4 is the update for shadow nodes by using procedure *UpdateSNode*. If v_y cannot be a part of the shortest path for its neighbor v_x , then we should check if it can be a gateway node pair or not. If so, we should update labels of all possible shadow nodes. Step P.1 considers only permanently labeled nodes in order to check if it can be a gateway node pair. The reason is that we cannot guarantee that the computed path for shadow nodes through its (temporarily labeled) neighbors would be shortest path since the path for temporarily labeled node could be changed at any time. So, shadow nodes in two different sub-trees are considered at the same time. Steps 2.8.4.1.1 and 2.8.4.1.2 compute the path for shadow nodes for all critical nodes along the path P_x^* and P_y^* . We finally delete v_y from the Q (Step 2.9).

Comments:

$d[x] = \min(d[x][green], d[x][red])$

$d[y'][red] = \infty$ for all shadow nodes $v_{y'}$

Z is the set of nodes with only one free interface

P is the set of permanently labeled nodes

Q is a Priority Queue with vertices and distance labels

C is the set of critical nodes

V' is the set of shadow nodes

I.0 Procedure *Initialize*

I.1 for each $v_x \in V$

I.1.1 $d[x] \leftarrow d[x][green] \leftarrow d[x][red] \leftarrow \infty$

I.1.2 $pred[x] \leftarrow pred[x][green] \leftarrow pred[x][red] \leftarrow null$

I.1.3 $CL[x] \leftarrow \emptyset$

I.1 endfor

I.2 $d[s] \leftarrow d[s][green] \leftarrow d[s][red] \leftarrow 0$

I.3 $P, V', C \leftarrow \emptyset \quad Q \leftarrow V$

I.0 End Procedure

Degree Constrained Shortest Path Algorithm

INPUT : $G = (V, E)$ and source v_s and $E = S \cup R$ OUTPUT: $G' = (V \cup V', E')$ with *pred* giving shortest path information

```

1  call Initialize
2  while NotEmpty(Q)
2.1   $v_y \leftarrow \arg_{v_x \in Q} \min d[x]$  Exit if  $d[y] = \infty$ 
2.2   $P \leftarrow P \cup \{v_y\}$ 
2.3  if  $d[y][green] \leq d[y][red]$  then //  $v_y$ 's shortest path enters on a green edge//
2.3.1   $pred[y] \leftarrow pred[y][green]$ 
2.4  else //  $v_y$ 's shortest path enters using a red edge //
2.4.1   $pred[y] \leftarrow pred[y][red]$ 
2.4.2  if  $v_y \in Z$  then //  $v_y$  has only one free interface //
2.4.2.1   $C \leftarrow C \cup \{v_y\}$ 
2.4.2  endif
2.3  endif
2.5  if  $pred[y]$  is not null then
2.5.1   $CL[y] \leftarrow CL[pred[y]]$  //copy critical list from parent //
2.5  endif
2.6  if  $v_y \in C$  then // processing a critical node //
2.6.1   $adj[y] \leftarrow v_x, \forall v_x, e_{xy} \in S$  //add green neighbors //
2.6.2   $CL[y] \leftarrow CL[y] \cup \{v_y\}$ 
2.6.3   $V' \leftarrow V' \cup \{v'_y\}$  // create a shadow node //
2.6.4   $Q \leftarrow Q \cup \{v'_y\}$ 
2.6.5   $d[y'] \leftarrow d[y][green] \leftarrow d[y][red] \leftarrow \infty$ 
2.6.6   $pred[y'][green] \leftarrow pred[y][red] \leftarrow null$ 
2.6.7   $adj[y'] \leftarrow v_x, \forall v_x, e_{yx} \in R \wedge v_x \notin P$ 
2.7  else if  $v_y \in V$  then // processing a non-critical node //
2.7.1   $adj[y] \leftarrow \{v_x | (e_{yx} \in S) \vee (e_{yx} \in R \wedge v_x \notin C)\}$ 
2.7.2   $adj[y] \leftarrow adj[y] \cup \{v'_x | (e_{yx} \in R \wedge v_x \in C \wedge v'_x \in P)\}$ 
2.6  endif
2.8  for  $\forall v_x \in adj[y]$ 
2.8.1  if  $e_{yx} \in S$  then index  $\leftarrow$  green
2.8.2  else if  $e_{yx} \in R$  then index  $\leftarrow$  red
2.8.1  endif
2.8.3  if  $d[x][index] > d[y] + w_{yx}, v_x \notin P$  then
2.8.3.1   $d[x][index] \leftarrow d[y] + w_{yx}$ 
2.8.3.2   $pred[x][index] \leftarrow \{v_y\}$ 
2.8.3.3   $d[x] \leftarrow \min(d[x][green], d[x][red])$ 
2.8.4  else
2.8.4.1  if  $v_x \in P$  then
2.8.4.1.1  call UpdateSNode( $v_y, v_x$ )
2.8.4.1.2  call UpdateSNode( $v_x, v_y$ )
2.8.4.1  endif
2.8.1  endif
2.8  endfor
2.9  Delete  $[Q, v_y]$ 
2  endwhile

```

```

P.0 Procedure UpdateSNode( $v_m, v_n$ )
P.1   for each  $v_i \in CL[m] - CL[n]$  and  $v'_i \notin P$ 
P.1.1   if  $d[i'][\text{green}] > d[n] + w_{nm} + d[m] - d[i]$  then
P.1.1.1      $d[i'][\text{green}] \leftarrow d[n] + w_{nm} + d[m] - d[i]$  //encodes a path from  $v_n$  to  $v_i$ //
P.1.1.2      $pred[i'][\text{green}] \leftarrow \{v_n\}$ 
P.1.1.3      $CL[i'] \leftarrow CL[n]$ 
P.1.1.4      $d[i'] \leftarrow d[i'][\text{green}]$ 
P.1.1.5      $d[i'][\text{red}] \leftarrow \infty$ 
P.1.1.6      $E' \leftarrow E' \cup \{e'_{ni'}\}$ 
P.1.1   endif
P.1   endfor
P.0 End Procedure
    
```

4 Detailed Example

Consider the graph shown in Figure 2. The source vertex is v_s . Our goal is to compute shortest valid paths from the source to all vertices in the graph. We now illustrate how the algorithm computes shortest valid paths and shortest alternate paths (for critical nodes). Suppose that the nodes in $Z = \{v_c, v_f, v_b, v_a\}$, and we can pick at most one red edge incident to any of these nodes in a shortest path.

Initially, we have $Q = \{v_s, v_c, v_b, v_d, v_f, v_a, v_e\}$. The first row of the table shows the distance labels of each vertex in V when we start the while loop in Step 2. In fact, we show the status of the queue and the distance labels each time we start a new iteration of the while loop. In the table, for each node we denote the shortest path lengths ending with a green/red edge as x/y .

Iteration	v_s	v_c	v_b	v_d	v_f	v_a	v_e	v'_c	v'_b	v'_f
1 ($v_y = v_s$)	0/0	∞/∞	∞/∞	∞/∞	∞/∞	∞/∞	∞/∞			
2 ($v_y = v_c$)	0/0	$\infty/5$	$\infty/9$	$7/\infty$	∞/∞	∞/∞	∞/∞			
3 ($v_y = v_d$)	0/0	$\infty/5$	$55/9$	$7/\infty$	∞/∞	$15/\infty$	∞/∞	∞/∞		
4 ($v_y = v_e$)	0/0	$\infty/5$	$55/9$	$7/\infty$	∞/∞	$15/\infty$	$8/\infty$	∞/∞		
5 ($v_y = v_b$)	0/0	$\infty/5$	$11/9$	$7/\infty$	∞/∞	$15/\infty$	$8/\infty$	∞/∞		
6 ($v_y = v'_b$)	0/0	$\infty/5$	$11/9$	$7/\infty$	∞/∞	$15/\infty$	$8/\infty$	$59/\infty$	$11/\infty$	
7 ($v_y = v_a$)	0/0	$\infty/5$	$11/9$	$7/\infty$	∞/∞	$15/16$	$8/\infty$	$59/\infty$	$11/\infty$	
8 ($v_y = v'_c$)	0/0	$\infty/5$	$11/9$	$7/\infty$	∞/∞	$15/16$	$8/\infty$	$26/\infty$	$11/\infty$	
9 ($v_y = v_f$)	0/0	$\infty/5$	$11/9$	$7/\infty$	$\infty/31$	$15/16$	$8/\infty$	$26/\infty$	$11/\infty$	
10 ($v_y = v'_f$)	0/0	$\infty/5$	$11/9$	$7/\infty$	$\infty/31$	$15/16$	$8/\infty$	$26/\infty$	$11/\infty$	∞/∞

- (Iteration 1): $Q = \{v_s, v_c, v_b, v_d, v_f, v_a, v_e\}$
 $v_y = v_s$. Add v_s to P . Since v_s is not critical, we set $pred[s] = null$ in Step 2.3.1. $CL[v_s] = \emptyset$. We define $adj[s] = \{v_d, v_c, v_b\}$ in Step 2.7.1. In Step 2.8 we now update the distance labels of v_d, v_c, v_b . Since none of these nodes are in P , we do not call procedure *UpdateSNode*. The updated distance labels are shown in the second row.
- (Iteration 2): $Q = \{v_c, v_b, v_d, v_f, v_a, v_e\}$
 $v_y = v_c$. Add v_c to P . Since v_c is critical (shortest path enters using a red edge, and $v_c \in Z$) we add v_c to C . We define $pred[c] = v_s$. We also define $CL[c] = \{v_c\}$. Since v_c is critical, in Step 2.6 we define $adj[c] = \{v_a, v_b\}$. Note that we do not add v_f to $adj[c]$ since the edge (v_c, v_f) is in R . We also create a shadow node v'_c and

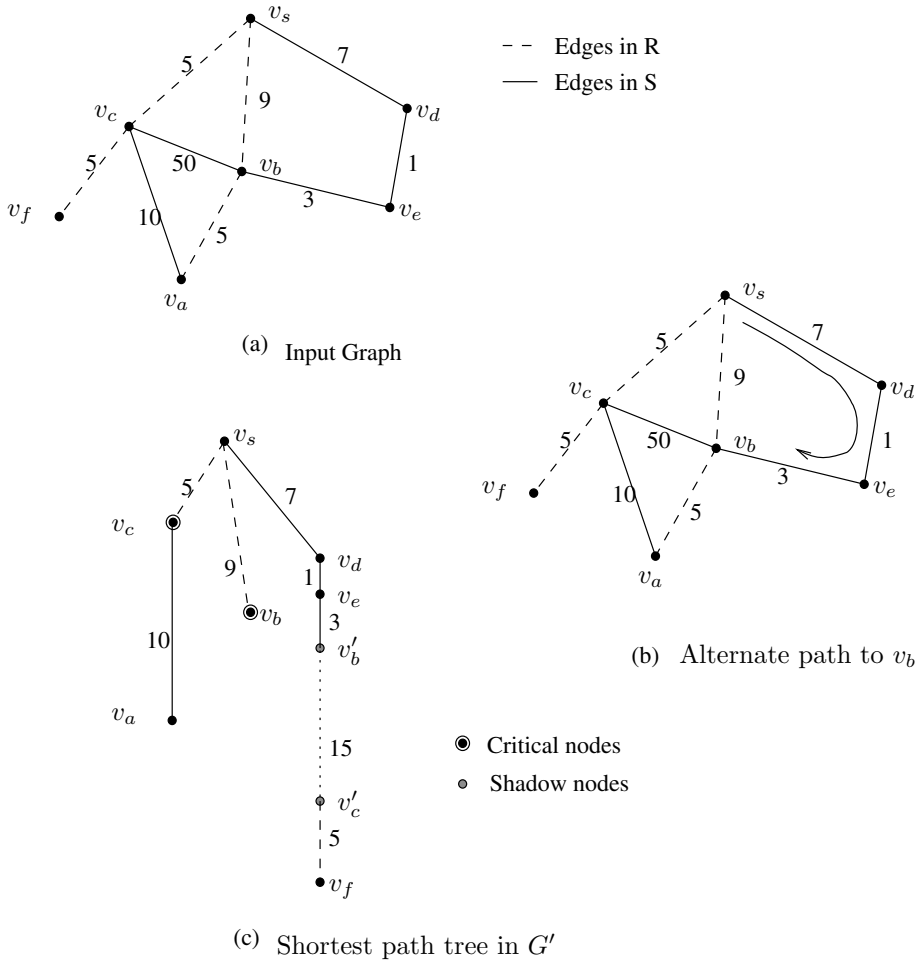


Fig. 2. Example to illustrate algorithm

add it to Q and V' . We define $adj[c'] = \{v_f\}$. We now update the distance labels of v_a and v_b in Step 2.8. The updated distance labels are shown in the third row.

3. (Iteration 3): $Q = \{v_b, v_d, v_f, v_a, v_e, v'_c\}$
 $v_y = v_d$. Add v_d to P . Define $pred[d] = v_s$. $CL[d] = \emptyset$. In Step 2.7.1 we define $adj[d] = \{v_e\}$. We update the distance label of v_e in Step 2.8 The updated distance labels are shown in the fourth row.
4. (Iteration 4): $Q = \{v_b, v_f, v_a, v_e, v'_c\}$
 $v_y = v_e$. Add v_e to P . We define $pred[e] = v_d$. $CL[e] = \emptyset$. In Step 2.7.1 we define $adj[e] = \{v_b\}$. We update the distance label of v_b in Step 2.8. The updated distance labels are shown in the fifth row.
5. (Iteration 5): $Q = \{v_b, v_f, v_a, v'_c\}$
 $v_y = v_b$. Add v_b to P . Since v_b is critical (shortest path enters using a red edge, and $v_b \in Z$) we add v_b to C . We define $pred[b] = v_s$. $CL[b] = \{v_b\}$. Since v_b is critical, in Step 2.6 we define $adj[b] = \{v_c, v_e\}$. We also create a shadow node v'_b and add

it to Q and V' . We define $adj[b'] = \{v_a\}$. In Step 2.8, since both v_c and v_e are in P we call $UpdateSNode(v_b, v_c)$, $UpdateSNode(v_c, v_b)$ and $UpdateSNode(v_b, v_e)$, $UpdateSNode(v_e, v_b)$. Consider what happens when we call $UpdateSNode(v_b, v_c)$, $UpdateSNode(v_c, v_b)$. We update the distance label of v'_b to 55. We also define $pred[b'] = v_c$. At the same time we update the distance label of v'_c to 59 and define $pred[c'] = v_b$. Consider what happens when we call $UpdateSNode(v_b, v_e)$, $UpdateSNode(v_e, v_b)$.

We update the distance label of v'_b to 11 and re-define $pred[b'] = v_e$. The updated distance labels are shown in the sixth row.

6. (Iteration 6): $Q = \{v_f, v_a, v'_c, v'_b\}$
 $v_y = v'_b$. Add v'_b to P . We define $pred[b'] = v_e$. $CL[b'] = \emptyset$. Recall that $adj[b'] = \{v_a\}$. (Since this is a shadow node, note that it is not processed in Step 2.6 or Step 2.7). In Step 2.8 we let $v_x = v_a$. However, this does not affect $d[a]$ which has value 15, even though it updates $d[a][red]$. The updated distance labels are shown in the seventh row.
7. (Iteration 7): $Q = \{v_f, v_a, v'_c\}$
 $v_y = v_a$. Add v_a to P . Define $pred[a] = v_c$ and $CL[a] = \{v_c\}$. Since $v_a \in V$ and is not critical, in Step 2.7.1 we define $adj[a] = \{v_c\}$. In Step 2.7.2, we add v'_b to $adj[a]$. In Step 2.8, we make calls to $UpdateSNode(v_a, v_c)$, $UpdateSNode(v_c, v_a)$ and $UpdateSNode(v_a, v'_b)$, $UpdateSNode(v'_b, v_a)$. The first two calls do not do anything. The second two calls update $d[c']$ to be $11 + 5 + 15 - 5 = 26$ (Step P.1.1.1). We also define $pred[c'][green] = v'_b$. The updated distance labels are shown in the eighth row.
8. (Iteration 8): $Q = \{v_f, v'_c\}$
 $v_y = v'_c$. We add v'_c to P and this node is not critical. We define $pred[c'] = v'_b$. We also define $CL[c'] = \emptyset$. Recall that $adj[c'] = \{v_f\}$. (Since this is a shadow node, note that it is not processed in Step 2.6 or Step 2.7). In Step 2.8 we update $d[f] = 31$. The updated distance labels are shown in the ninth row.
9. (Iteration 9): $Q = \{v_f\}$
 $v_y = v_f$. We add v_f to P . This node is identified as critical since it is in Z . We also define $CL[f] = \emptyset$. We create a shadow node v'_f . However, v_f has no green neighbors so $adj[f] = \emptyset$. We add v'_f to V' and $adj[f'] = \emptyset$. The updated distance labels are shown in the tenth row.
10. (Iteration 10): $Q = \{v'_f\}$
 We exit the loop since $d[f'] = \infty$.

Due to space limitations, we omit the proof of the algorithm and the complexity analysis completely.

Acknowledgments

We thank Julian Mestre and Azarakhsh Malekian for useful comments on an earlier draft of the paper.

References

1. Ravindra K. Ahuja , Thomas L. Magnanti, James B. Orlin, “Network Flows: Theory, Algorithms and Applications”, *Prentice Hall*, 1993.
2. W. Cook, A. Rohe, “Computing Minimum Weight Perfect Matchings”, *INFORMS Journal of Computing*, 1998.

3. S. Fekete, S. Khuller, M. Klemmstein, B. Raghavachari and N. Young, "A Network-Flow technique for finding low-weight bounded-degree spanning trees", *Journal of Algorithms*, Vol 24, pp 310–324 (1997).
4. M. Fürer and B. Raghavachari, "Approximating the minimum degree Steiner tree to within one of optimal", *Journal of Algorithms*, Vol 17, pp 409–423 (1994).
5. H. N. Gabow, "Data structures for weighted matching and nearest common ancestors with linking", *Proc. of the ACM-SIAM Symp. on Discrete Algorithms*, pages 434–443, 1990.
6. H. N. Gabow and R. E. Tarjan, "Efficient algorithms for a family of matroid intersection problems", *Journal of Algorithms*, Vol 5, pp 80-131 (1984).
7. P. Gurumohan, J. Hui "Topology Design for Free Space Optical Network", *ICCCN '2003*, Oct. 2003
8. Z. Huang, C-C. Shen, C. Srisathapornphat and C. Jaikaeo, "Topology Control for Ad Hoc Networks with Directional Antennas", *ICCCN '2002*, Miami, Florida, October 2002.
9. A. Kashyap, K. Lee, M. Shayman "Rollout Algorithms for Integrated Topology Control and Routing in Wireless Optical Backbone Networks" *Technical Report*, Institute for System Research, University of Maryland, 2003.
10. J. Könemann and R. Ravi, "Primal-dual algorithms come of age: approximating MST's with non-uniform degree bounds", *Proc. of the 35th Annual Symp. on Theory of Computing*, pages 389–395, 2003.
11. S. Koo, G. Sahin, S. Subramaniam, "Dynamic LSP Provisioning in Overlay, Augmented, and Peer Architectures for IP/MPLS over WDM Networks", *IEEE INFOCOM*, Mar. 2004.
12. K. Lee, M. Shayman, "Optical Network Design with Optical Constraints in Multi-hop WDM Mesh Networks", *ICCCN'04*, Oct. 2004.
13. E. Leonardi, M. Mellia, M. A. Marsan, "Algorithms for the Logical Topology Design in WDM All-Optical Networks", *Optical Networks Magazine*, Jan. 2000, pp. 35-46.
14. N.A.Riza, "Reconfigurable Optical Wireless", *LEOS '99*, Vol.1, Nov. 1999, pp. 8-11.

A New Template for Solving p -Median Problems for Trees in Sub-quadratic Time (Extended Abstract)

Robert Benkoczi¹ and Binay Bhattacharya²

¹ School of Computing Queen's University Kingston, ON, Canada K7L 3N6

² School of Computing Science, Simon Fraser University,
Burnaby, BC, Canada V5A 1S6

Abstract. We propose an $O(n \log^{p+2} n)$ algorithm for solving the well-known p -Median problem for trees. Our analysis relies on the fact that p is considered constant (in practice, very often $p \ll n$). This is the first result in almost 25 years that proposes a new algorithm for solving this problem, opening up several new avenues for research.

1 Introduction

The p -Median problem is one of several problems considered fundamental for the field of combinatorial optimization. It has a rich history dating back to the 17-th century when it was posed as a mathematical puzzle most probably by French mathematician Pierre de Fermat [12]. Today, there is a huge amount of literature dedicated to the p -Median problem (see the survey of Hale *et al.* [19]).

For a graph $G = (V, E)$ with vertex weights and edge lengths, the p -Median problem seeks to identify a subset of p vertices called *facilities* or *medians*, so that the sum of the weighted distances from each vertex in G to the closest median is minimized. It was shown by Kariv and Hakimi in [23] that the p -Median problem is NP-complete even for planar graphs with unit edge length and with maximum vertex degree 3. Moreover, Lin and Vitter [24,25] showed that approximating p -Median is as hard as approximating dominating set and set cover, and therefore, it is unlikely that there exist constant factor approximation algorithms for these problems. However, their result applies to general instances of the problem where the distance (cost) between pairs of locations does not satisfy the triangle inequality. For the planar or graph p -Median, constant factor approximations are possible and many algorithms have been proposed [22,10,31,3,2].

When the input graph is a tree, an interval, or a circular arc graph, the p -Median problem is solved in polynomial time and the best algorithms known to date have a running time of $O(pn^2)$ (Tamir [29]), $O(pn \log n)$ (Bespamyatnikh *et al.* [7]), and $O(pn^2 \log n)$ [7] respectively. When $p \in \{1, 2, 3\}$, it is possible to design algorithms that are even more efficient. For trees, the 1-Median can be solved easily in $O(n)$ time (Goldman *et al.* [15,16]), the 2-Median in $O(n \log n)$ time (Gavish *et al.* [14] and Breton [9]), and the 3-Median in $O(n \log^3 n)$ time (Benkoczi *et al.* [6,5]). Tamir's p -Median algorithm for trees [29] is in fact a slight modification of a straightforward dynamic programming approach proposed by

Kariv and Hakimi [23] more than twenty years ago. With his modification, Tamir was able to prove a tighter upper bound on the running time of the dynamic programming method improving Kariv and Hakimi's result from $O(p^2n^2)$ to $O(pn^2)$. Surprisingly, no other results are known for the p -Median problem on trees.

Our Results. In this paper, we describe a new algorithm to solve the p -Median problem on trees. As long as p is considered fixed, our algorithm has a running time that is sub-quadratic in n , where n represents the number of vertices of the tree. This is the first improvement on the solution to the p -Median problem in a context where several similar optimization problems on trees with previous quadratic complexity have been solved more efficiently. Shah, Langerman, and Lodha [28] considered the problem of placing filters in multicast trees and proposed an $O(n \log n)$ algorithm based on a modified dynamic programming model. Shah and Farach-Colton [27] took the same dynamic programming paradigm and solved the uncapacitated facility and minimum cost coverage problems in trees again in $O(n \log n)$ time. However, the authors were unable to extend their result on the p -Median problem. Our method is still based on a dynamic programming formulation, but we use a totally different approach of working with the cost functions which gives us a time complexity of $O(n \log^{p+2} n)$. It is still not known whether a p -Median algorithm for trees with a running time sub-quadratic in n exists when p is not fixed.

Solving optimization problems defined on trees is important for several reasons. First, algorithms on trees can be used in deriving approximation algorithms for other more general problem instances [30]. For this reason, there is a growing interest now in studying the relationship between tree metrics and arbitrary metrics [4,13]. Second, our result advances the research towards finding ways to exploit the special structure present in other types of graphs such as the partial k -trees [26]. These are graphs which have a configuration resembling that of a tree. There are few published results concerning facility location problems in partial k -trees and many issues are still open. Among the articles that considered the structure of special graphs we mention the work of Gurevich and Stockmeyer [18] on the continuous minimum cover problem, Hassin and Tamir [20] on uncapacitated facility location (UFL) and p -center problems in partial 2-trees, Granot and Skorin-Kapov [17] on UFL.

Our paper is organized as follows. In Section 2 we describe the general idea behind dynamic programming algorithms for the p -Median problem. The time complexity of this approach is quadratic in n . We explain the modification of Shah *et al.* [28,27] that leads to sub-quadratic performance for solving problems like UFL. We then introduce our own approach which leads to sub-quadratic running time for the p -Median problem on balanced binary trees. With the help of a tree decomposition structure which is presented in Section 3, we show in Section 4 how we can extend our results to arbitrary trees.

2 The General Approach

We consider a rooted binary tree $T = (V, E)$ with vertex weights and edge lengths. If the given tree is not rooted or binary, we can root it at an arbitrary

vertex or make it binary by adding a linear number of vertices and edges of weight zero as in [29]. For vertex $v \in V$ let $w(v)$ be its weight; similarly, for edge $e \in E$, let $l(e)$ be its length. We denote the tree distance between vertices u and v by $d(u, v)$. Let F_p be an arbitrary set of p medians of T . The cost of having F_p as median set is,

$$C(F_p) = \sum_{z \in F_p} c(z) + \sum_{v \in V} w(v) d(v, F_p),$$

where $c(z)$ is an additional cost incurred for considering vertex z as one of the medians and $d(v, F_p) = \min_{z \in F_p} d(v, z)$ is the tree distance from v to the closest median in F_p . We say that v is served (covered) by the closest median in F_p . This cost model is identical with that used by Tamir [29] and generalizes the usual p -Median cost function for which $c(z) = 0, \forall z \in V$. The p -Median problem is to compute

$$\min_{\substack{F_p \subseteq V \\ |F_p|=p}} C(F_p).$$

$C(F_p)$ is also called *the objective function*.

The classical dynamic programming algorithms to solve the p -Median problem in trees [23,29] compute recursively bottom-up a set of cost functions associated with the subtrees of T . The cost functions represent the contribution to the objective function of only the vertices in the given subtree. Starting from the leaves of T for which the cost functions have trivial values, a set of cost functions for the entire tree is computed and the optimal p -Median solution is given by the minimum cost function from this set. The correctness of the dynamic programming algorithm is insured by the following condition.

Optimality Condition. Consider F_p to be an optimal median set for the objective function $C(F_p)$. Out of the p medians from F_p , assume that q are located in subtree T_v . Denote by G_q these q vertices ($G_q = F_p \cap T_v$), and denote by z the median covering v , the root of T_v . Then G_q is an optimal q -Median for T_v under the constraint that vertex z is a median covering v .

This condition suggests a definition for the cost functions of the dynamic programming algorithm. Every cost function has three parameters. (1) $v \in V$ is the root of the subtree (T_v) for which the cost function is computed, (2) $q \in \mathbb{Z}_+$ is the number of medians to be located optimally inside T_v , and (3) $z \in V$ is the median constrained to cover v ; if $z \in T_v$, then z is one of the optimal q medians of T_v . The value returned by the cost function is the minimum cost of the q -Median problem on subtree T_v with the restriction that vertex z given as parameter is one of the medians and it covers root v . To simplify our exposition later on, we use two notations for the cost functions, depending on where parameter z is chosen from. We use $IN(v, q, z)$ if z is a vertex from T_v and $OUT(v, q, z)$ if $z \in T \setminus T_v$. Clearly then, the optimal solution of the p -Median problem is obtained from

$$\min_{z \in T} IN(r, p, z), \quad \text{where } r \text{ is the root of } T.$$

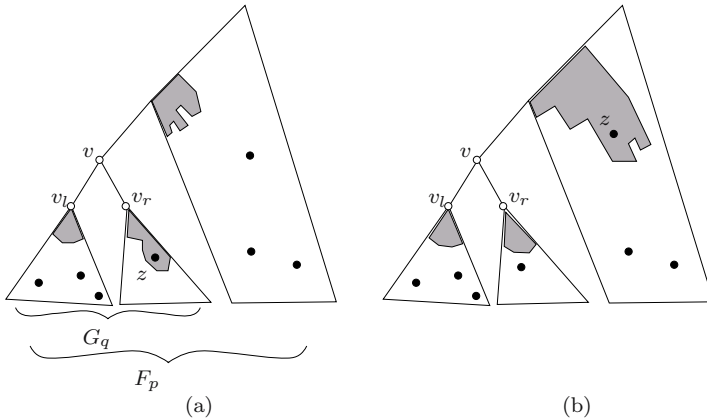


Fig. 1. Cost functions defined for subtree T_v . The shaded area represents the vertices in T covered by z . Case (a) $z \in T_v$; case (b) $z \in T \setminus T_v$.

Since the optimal choice for z is not known beforehand, we need to compute cost functions for all $z \in V$ at every subtree T_v , which amounts to $O(pn^2)$ cost functions in total. To reduce the time complexity of the algorithm to a function sub-quadratic in n , we use a simple observation made by Shah *et al.* [28]. They observed that the value of function $OUT(v, q, z)$ does not depend on z but on the distance from z to v . Recall that $z \in T \setminus T_v$ in this case and the additive cost $c(z)$ is not accounted for by function OUT . Therefore, we can work with continuous cost function $OUT(v, q, \alpha)$, where α is the distance from v to the external median and belongs to the range $[0, \infty)$. Dynamic programming with this kind of continuous cost functions was named *undiscretized dynamic programming* by Shah *et al.* [28].

The algorithm remains essentially the same. Function $OUT(v, q, \alpha)$ is obtained recursively by adding and taking the minimum of several functions computed for the children of v . It can be shown that function $OUT(v, q, \alpha)$ is piecewise linear and concave, and thus addition and minimum operations take time proportional to the total number of linear pieces of the functions involved. The number of linear pieces is called the complexity of the cost function. For uncapacitated facility location, Shah *et al.* [27] proved that the complexity is linear in the size of the subtree for all cost functions. They used a data structure to store cost functions that allows all operations to be performed in time linear in the complexity of the smaller function and logarithmic in the complexity of the larger one. As consequence, the running time of their algorithms is $O(n \log n)$. However, they could not apply their method to the p -Median problem because the cost functions have too many parameters.

Our approach is different. We use no special data structure to store our cost functions and all operations on them are proportional to the sum of the complexities of the functions involved. If the complexity of cost function $OUT(v, q, \alpha)$ is bounded by some increasing function $f(|T_v|, q)$ with two parameters, then the

total running time of our procedure becomes at least $O(h f(n, p))$ where h is the height of the tree. In Section 4 we sketch the proof for an upper bound of $O(n \log^{p-2} n)$ on $f(n, p)$. For balanced binary trees, h is $O(\log n)$. Thus, we can show a running time of $O(n \log^{p-1} n)$ for our algorithm on balanced binary trees.

To accommodate arbitrary trees, we preprocess the given tree in a data structure called *spine decomposition*. This is a new data structure that allows us to treat any set of trees as if they were of logarithmic height. In the following section we give a brief description of the spine decomposition and in Section 4 we show how to define the cost functions in order to utilize the properties of our decomposition. Finally, we review the main steps we took in proving the time complexity of $O(n \log^{p+2} n)$ and space complexity of $O(n \log^p n)$ for our general algorithm.

3 The Spine Decomposition (SD) of Trees

In a decomposition of a tree we partition the input tree T into two or more subtrees called the components. Each component is then partitioned recursively. This recursive process can be traced by the recursion tree. The depth of the recursion tree is logarithmic. In the literature, there exist several tree decompositions, perhaps the best known being the centroid decomposition [11]. In the centroid decomposition, the partition in two components is determined by the existence of a special vertex in a tree called the centroid of the tree.

Our decomposition, the spine decomposition (SD) [6,5], uses a path to direct the partition of the input tree. This makes it suitable for computation tasks involving components of the tree that interact with one another. A very similar decomposition was proposed independently by Boland [8]. There also exist other decompositions that are suitable for our purposes, for example the top-trees proposed by Holm *et al.* ([21] and [1]). However, we feel that SD is more intuitive for the type of problems we are solving here.

We now describe the SD structure (see Fig. 2). Consider a binary tree T rooted at a vertex r_T . We select a path from r_T to a leaf in T such that the next vertex in this path always follows the child with the most number of leaves hanging from it. Formally, if $v_0 = r_T, v_1, \dots, v_k$ are the vertices on the path, if $p(v)$ denotes the parent of v , and if $N_l(v)$ denotes the number of leaves that have v as ancestor, then we always have $N_l(v_{i+1}) \geq N_l(u_i)$ where u_i is the other child of node v_i . We call path v_0, v_1, \dots, v_k a spine and use notation $\pi(v_0, v_k)$. If we remove the spine from T we obtain a set of at most k disconnected components which are each recursively decomposed. Let $T(u_i)$ be one of these components rooted at u_i where u_i is adjacent to spine vertex v_i but is not itself a spine vertex.

Let $\lambda(v_i)$ denote the number of leaves of $T(u_i)$. We construct a binary search tree with vertices v_i as leaves and we associate it with the spine. Thus the leaves of the search tree and the spine vertices of $\pi(v_0, v_k)$ are the same (or one can consider a one to one mapping between them). The root of the search tree is linked to the spine vertex of the parent spine, *i.e.* the root of the search tree for

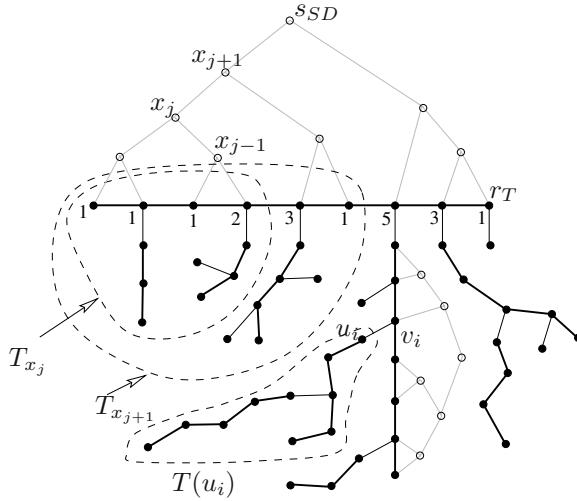


Fig. 2. A typical spine decomposition; spines are shown in thick lines, search trees as thin lines and components are outlined by dashed lines; the numbers beside spine vertices at the top-most spine give the number of leaves of T for the corresponding SD component

the spine of $T(u_i)$ is linked to v_i . The search tree is balanced by weight $\lambda(v_i)$ associated with leaf v_i such that components with many leaves (and thus many recursive spines) are closer to the root of the search tree. Once all the search trees for all spines are constructed, any tree traversal is performed through the search trees and not through the links of the original tree T . A search tree traversal can begin at root s_{SD} of the search tree of the top-most spine; then the search tree is traversed until leaf v_i on the spine is reached; the next search tree node visited is the root of the search tree constructed for $T(u_i)$. In this way, the paths in several search trees are concatenated, and we can talk about one super-path through search trees connecting any two search tree nodes¹. We denote the super-path between search tree nodes x and y by $\sigma(x, y)$. Two important properties of the SD are mentioned here without proof because of space constraints. Full proofs are provided in [5].

Theorem 1. *The length (number of edges) of any super-path $\sigma(x, y)$ in the SD is $O(\log n)$ where n is the number of vertices of the input tree T .*

Theorem 2. *The construction algorithm for the SD has time complexity $O(n)$. The storage space complexity of the data structure for the SD is also $O(n)$.*

It is interesting to point out here that although both our spine decomposition and the centroid decomposition have linear time construction algorithms, the algorithm for SD is simpler than the one for the centroid decomposition.

¹ Note that any vertex of the input tree T is also a leaf of some search tree.

4 The p -Median Template

We now describe a dynamic programming algorithm for the p -Median problem on trees that runs in time sub-quadratic in n . Our algorithm follows the steps outlined in Section 2 but associates the cost functions with the nodes of search trees in the SD. Each such node corresponds to a subtree of T . The algorithm can be sketched as follows:

- Transform input tree T into binary tree as in [29].
- Construct the spine decomposition of T .
- Traverse the decomposition and compute the cost functions *bottom-up*, as sketched in the following paragraphs.
- Return the cost function with minimum value computed for the root of the spine decomposition.

To exploit the balanced structure of the SD we define the cost functions differently than in Section 2. In this section we will describe each cost function and we will illustrate the recursive computation of one of them (for lack of space). But first, we introduce the notation for the subtrees of T that are associated with the nodes of the SD.

For a given search tree node x (Fig. 3 - a), let (i) T_x – small subtree; it is the subtree of T induced by all vertices v that contain x in the super-path from v to the root s_{SD} of the decomposition and (ii) $T(x_R)$ – big subtree; it is the subtree of T rooted at x_R in the usual sense. The cost functions for node x are obtained from the cost functions computed for its two children y and t in the search tree (Fig. 3 - a). To insure that the dynamic programming algorithm runs in time sub-quadratic in n , the cost functions should have a complexity linear – ignoring logarithmic factors – in the size of the small subtrees. Only then we can hope to obtain a sub-quadratic algorithm if we implement addition and minimum of cost functions in time linear in the sum of their complexities.

Before we explain how we satisfy this requirement, we introduce the notion of split edge [14,9,6]. A solution to the p -Median problem is a set of p vertices of T , but it can also be viewed as a collection of $p - 1$ edges of T . If these edges are removed (split), the tree gets disconnected into p components and the optimal 1-Medians of each component form together the p -Median set that is solution to the original p -Median problem. We can thus seek to compute optimal split edges instead of optimal medians. We are going to define our cost functions in terms of split edges because there are special cases that we need to consider during the computation of cost functions which occur when one or more split edges fall on the spine. If we work with medians only, these cases are difficult to model.

As presented in Section 2, we have two types of cost functions: discrete (a given median is forced to cover a special vertex of the given subtree) and continuous (an external median is forced to cover a special vertex of the given subtree). We use the following notation.

1. $IBU =$ Inside Big Unconstrained (similar to function IN)
 $IBU_R(x, j, z)$ returns the optimal cost of $T(x_R)$ if j split edges (and thus $j + 1$ medians) are selected in $T(x_R)$. The facility covering x_R is vertex z

chosen only from T_x . In this way, the number of values for cost function $IBU_R(x, j, z)$ is proportional to the size of T_x and not $T(x_R)$. The cost of the optimal p -Median solution for T can be retrieved from

$$\min_{z \in T} IBU_R(s_{SD}, p - 1, z).$$

Hence, our goal is to evaluate $IBU_R()$ at the SD root node s_{SD} .

2. $OBU =$ Outside Big Unconstrained (similar to function OUT).

Function $OBU_R(x, j, \alpha)$ returns the cost of subtree $T(x_R)$ when j spine edges are chosen from T_x but some may be chosen from the spine. If a split edge is on the spine (see Fig. 3 (b)) then the outside median cannot cover the portion towards the leaf of the spine and cost functions $OBU_R()$ cannot be used recursively. In this case, we need to compute value C_{opt} illustrated in the figure which is the optimal q -Median of the subtree separated by the split edge for $q \in \{1, \dots, p - 1\}$. This is in fact the requirement most difficult to satisfy in our algorithm. Note that we force the split edges to belong to T_x and thus constrain the complexity of the function to depend on the size of the small subtree.

3. $OSC =$ Outside Small Constrained.

$OSC_R(x, j, \alpha)$ returns the optimal cost of T_x if j split edges are chosen from T_x , none of them on the spine, and the closest external median is at distance α from x_R and covers x_R . Function $OSC_L(x, j, \alpha)$ is the same except that the external median is at distance α from x_L , and covers x_L .

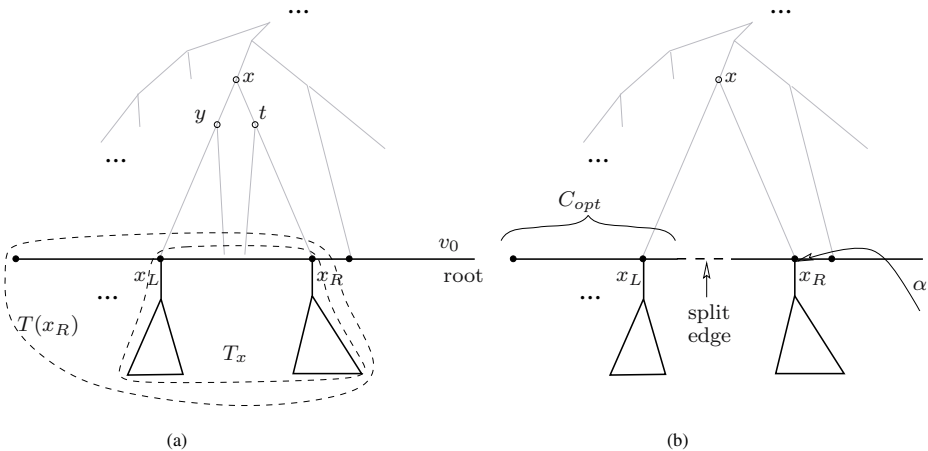


Fig. 3. Defining the cost functions for SD node x ; Case (b) for $OBU_R()$ when the rightmost split edge is on the spine

We illustrate now the recursive formula used in calculating $IBU_R(x, j, z)$ for a node x of the SD that is not a spine vertex (see Fig. 3 (a)). If median z comes

from the right subtree (T_t) then the value of the cost function remains unchanged from what was computed at node t according to the definition of $IBU_R()$. If z comes from the left subtree (T_y) then we need to add the contribution of vertices in T_t which is returned by function $OSC_L()$.

$$IBU_R(x, j, z) = \begin{cases} IBU_R(t, j, z), & \text{if } z \in T_t \\ \min_{0 \leq q \leq j} \left\{ IBU_R(y, q, z) + OSC_L(t, j - q, d(z, t_L)) \right\}, & \text{if } z \in T_y \end{cases} \tag{1}$$

When node x is on spine, the formula is more involved and requires evaluating $OBUR(x, j, \alpha)$ at different values of α . A complete presentation of all calculations is beyond the scope of this paper.

From (1) and the rest of the formulae used to compute cost functions, we obtain a recurrence relation for a function denoted $f : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{R}$ that represents an upper bound on the complexity of the continuous cost functions $OBUR(x, j, \alpha)$ and $OSCR(x, j, \alpha)$. The upper bound $f(|T_x|, j)$ is in terms of the size of the small subtree (T_x) and the number of split edges (j) given as parameter in the cost function.

Lemma 1. *For any node x of the SD of T ,*

$$f(|T_x|, j) \leq \begin{cases} 1, & \text{if } j = 0 \\ |T_x| - 1, & \text{if } j = 1 \\ c|T_x| \log |T_x| \left(1 + c \log |T_x|\right)^{j-2}, & \text{if } j \geq 2, \end{cases} \tag{2}$$

where c and j are constant.

The recurrence relation and the proof of Lemma 1 are not included due to space constraints.

Using Lemma 1, Theorem 1, and the fact that subtrees T_x for different nodes x situated at the same distance from the SD root s_{SD} are disjoint, the following can be proved.

Theorem 3. *Our algorithm has space complexity $O(n \log^{p-2} n + S_{C_{opt}}(n))$ and running time complexity $O(n \log^{p-1} n + n \log^3 n + nT_{C_{opt}}(n))$, where $p \geq 3$ is a constant, $T_{C_{opt}}(n)$ is the time complexity for computing C_{opt} in Fig. 3 (b) and $S_{C_{opt}}(n)$ is the extra space required by the method that computes C_{opt} .*

The Challenge. The challenge in our algorithm is to compute value C_{opt} (Fig. 3 (b)) for every edge of T being split and for every value $q \in \{1, \dots, p - 1\}$ in sub-linear time in the size of the subtree being split. We sketch here our approach.

Consider Fig. 4. We want to compute value C_{opt} (an optimal q -Median solution on tree $T(v_i)$). Vertex v_i is the spine vertex incident to the split edge towards the leaf. In the optimal solution v_i is served by some median $z \in T(v_i)$. Assume that z falls in subtree T_y for some SD node y . Notice that there are only $O(\log n)$ such nodes y which are hanging on the left of of the path from v_i to

the root of the search tree. Then, we can represent cost function $IBU_R(y, j, z)$ for all $z \in T_y$ as a point in distance-cost space (see Fig. 4). We can compute in a preprocessing step the lower convex hull of this set of points. It can be shown that the convex hull can be used in a binary search procedure to return the optimal $z \in T_y$ that covers v_i . The only requirement is to efficiently maintain the convex hull as a new split edge is selected to the right of the current v_i . This maintenance amounts to adding a continuous cost function $OSC_L()$ to the set of points in the convex hull. Since the function is piecewise linear and concave, we need to consider eliminating some points on the hull that have become reflex points. The procedure can be done in time proportional to the number of points eliminated and takes a total time polylogarithmic in n . The details are omitted from the conference version of this paper (see also [5]).

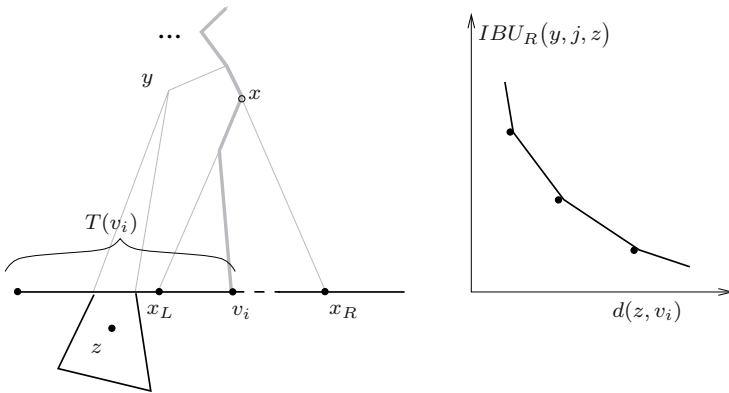


Fig. 4. Computing the optimal q -Median on subtree $T(v_i)$ in polylogarithmic time

Theorem 4. *The p -Median problem in trees can be solved, for any constant p in time $O(n \log^{p+2} n)$ and space $O(n \log^p n)$.*

5 Conclusion

In this paper we give the first algorithm for solving the p -Median problem in trees, for constant p , that is significantly different than the almost 25 years old dynamic programming algorithm of Kariv and Hakimi which was improved by Tamir in 1996. For constant values of p , it is asymptotically better than the best known $O(pn^2)$ procedure of Tamir. However, if p is not constant, we do not know whether sub-quadratic algorithms still exist. We conjecture that it is possible to solve the p -Median problem in less than $O(pn^2)$ even when p is not fixed.

References

1. S. Alstrup, J. Holm, and M. Thorup. Maintaining center and median in dynamic trees. In *In Proc. 7-th SWAT*, volume 1851 of *LNCS*, pages 46–56, 2000.
2. S. Arora, P. Raghavan, and S. Rao. Approximation schemes for euclidean k -medians and related problems. In *Proc. 30th Annual ACM Symposium on Theory of Computing (STOC'98)*, pages 106–113, 1998.
3. V. Arya, N. Garg, R. Khandekar, A. Meyerson, K. Mungala, and V. Pandit. Local search heuristic for k -median and facility location problems. In *Proc. 33rd Annual ACM Symposium on Theory of Computing*, pages 21–29, 2001.
4. Y. Bartal. On approximating arbitrary metrics by tree metrics. In *Proc. STOC*, pages 183–193, 1998.
5. Robert Benkoczi. *Cardinality constrained facility location problems in trees*. PhD thesis, School of Computing Science, Simon Fraser University, Burnaby, BC, Canada, May 2004.
6. R.R. Benkoczi, B.K. Bhattacharya, M. Chrobak, L. Larmore, and W. Rytter. Faster algorithms for k -median problems in trees. In B. Rován and P. Vojtáš, editors, *Proc. 28th International Symposium on Mathematical Foundations of Computer Science*, volume LNCS 2747, pages 218–227, 2003.
7. Sergei Bespamyatnikh, Binay Bhattacharya, M. Keil, David Kirkpatrick, and M. Segal. Efficient algorithms for centers and medians in interval and circular-arc graphs. *NETWORKS*, 39(3):144–152, 2002.
8. R.P. Boland. *Polygon visibility decompositions with applications*. PhD thesis, University of Ottawa, Ottawa, Canada, 2002.
9. D. Breton. Facility location optimization problems in trees. Master's thesis, School of Computing Science, Simon Fraser University, Canada, 2002.
10. M. Charikar and S. Guha. Improved combinatorial algorithms for facility location and k -median problems. In *Proc. 40th Symposium on Foundations of Computer Science (FOCS'99)*, pages 378–388, 1999.
11. R. Cole and U. Vishkin. The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time. *Algorithmica*, 3:329–346, 1988.
12. Z. Drezner, K. Klamroth, A. Schobel, and G.O. Wesolowsky. *Facility Location: Applications and Theory*, chapter The Weber Problem, pages 1–36. Springer-Verlag, 2002.
13. J. Fakcharoenphol, S. Rao, and K. Talwar. A tight bound on approximating arbitrary metrics by tree metrics. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing*, 2003.
14. R. Gavish and S. Sridhar. Computing the 2-median on tree networks in $O(n \log n)$ time. *Networks*, 26:305–317, 1995.
15. A.J. Goldman. Optimal center location in simple networks. *Trans. Sci.*, 5:212–221, 1971.
16. A.J. Goldman and C.J. Witzgall. A localization theorem for optimal facility placement. *Trans. Sci.*, 1:106–109, 1970.
17. Daniel Granot and Darko Skorin-Kapov. On some optimization problems on k -trees and partial k -trees. *Discrete Applied Mathematics*, 48(2):129–145, 1994.
18. Y. Gurevich, L. Stockmeyer, and U. Vishkin. Solving NP-hard problems on graphs that are almost trees and an application to facility location problems. *Journal of the ACM*, 31(3):459–473, 1984.
19. Trevor S. Hale and Christopher R. Moberg. Location science research: A review. *Annals of Operations Research*, 123:21–35, 2003.

20. R. Hassin and A. Tamir. Efficient algorithms for optimization and selection on series-parallel graphs. *SIAM Journal of Algebraic Discrete Methods*, 7:379–389, 1986.
21. Jacob Holm and Kristian de Lichtenberg. Top-trees and dynamic graph algorithms. Technical Report 17, Univ. of Copenhagen, Dept. of Computer Science, 1998.
22. K. Jain and V. Vazirani. Primal-dual approximation algorithms for metric facility location and k-median problems. Manuscript, March 1999.
23. O. Kariv and S.L. Hakimi. An algorithmic approach to network location problems II: The p-medians. *SIAM Journal on Applied Mathematics*, 37:539–560, 1979.
24. J.-H. Lin and J.S. Vitter. Approximation algorithms for geometric median problems. *Information Processing Letters*, 44:245–249, 1992.
25. J.-H. Lin and J.S. Vitter. ϵ -approximations with minimum packing constraint violation. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 771–782, 1992.
26. N. Robertson and P.D. Seymour. Graph minors. I. excluding a forest. *J. Combin. Theory Ser. B*, 35:39–61, 1983.
27. R. Shah and M. Farach-Colton. Undiscretized dynamic programming: faster algorithms for facility location and related problems on trees. In *Proc. 13th Annual Symposium on Discrete Algorithms (SODA)*, pages 108–115, 2002.
28. R. Shah, S. Langerman, and S. Lodha. Algorithms for efficient filtering in content-based multicast. In *Proc. 9th Annual European Symposium on Algorithms (ESA)*, pages 428–439, 2001.
29. A. Tamir. An $O(pn^2)$ algorithm for the p -median and related problems on tree graphs. *Operations Research Letters*, 19:59–64, 1996.
30. A. Tamir, D. Pérez-Brito, and J.A. Moreno-Pérez. A polynomial algorithm for the p-centdian problem on a tree. *Networks*, 32:255–262, 1998.
31. M. Thorup. Quick k-median, k-center, and facility location for sparse graphs. In *28th International Colloquium on Automata, Languages and Programming*, volume 2076 of *Lecture Notes in Computer Science*, pages 249–260, Crete, Greece, 2001.

Roll Cutting in the Curtain Industry

(Extended Abstract)

Arianna Alfieri¹, Steef L. van de Velde², and Gerhard J. Woeginger³

¹ Dipartimento dei Sistemi di Produzione ed Economia dell'Azienda,
Polytechnic University of Torino, Torino, Italy

² Rotterdam School of Management, Erasmus University,
Rotterdam, The Netherlands

³ Department of Mathematics and Computer Science,
Eindhoven University of Technology, Eindhoven, The Netherlands

Abstract. We study the problem of cutting a number of pieces of the same length from n rolls of different lengths so that the remaining part of each utilized roll is either sufficiently short or sufficiently long. A piece is sufficiently short, if it is shorter than a pre-specified threshold value δ_{\min} , so that it can be thrown away as it cannot be used again for cutting future orders. And a piece is sufficiently long, if it is longer than a pre-specified threshold value δ_{\max} (with $\delta_{\max} > \delta_{\min}$), so that it can reasonably be expected to be usable for cutting future orders of almost any length. We show that this problem, faced by a curtaining wholesaler, is solvable in $O(n \log n)$ time by analyzing a non-trivial class of allocation problems.

1 Introduction

The one-dimensional stock cutting problem to minimize trim loss continues to be a challenging day-to-day optimization problem for many companies in a variety of industries, with the first contributions tracing back to the seminal papers by Gilmore and Gomory (1961, 1963); see for instance the editorial of the recent featured issue of *European Journal Operational Research* on cutting and packing (Wang and Wäscher, 2002).

In this paper, we study the optimization of roll cutting faced by a curtaining wholesaler and distributor. An important subproblem is to cut a number of pieces of the *same* length from n rolls of different lengths so that the remaining part of each utilized roll is either sufficiently short so that it can be thrown away (as it cannot be used again for cutting future orders), or sufficiently long so that it can reasonably be expected to be usable for cutting future orders of almost any length. Mathematically, ‘sufficiently short’ means shorter than a pre-specified threshold value δ_{\min} , and ‘sufficiently long’ means longer than a pre-specified threshold value δ_{\max} , with $\delta_{\max} > \delta_{\min}$.

This problem comes close to a roll cutting problem in the *clothing* industry introduced by Gradišar *et al.* (1997). Two main differences play a role (and some smaller ones, also). First, in their problem, an order consists of pieces of *different* lengths; and second, no maximum threshold value applies—effectively, this

means that in their case $\delta_{\max} = \delta_{\min}$. Gradišar *et al.* (1997) present a heuristic for this problem with a compelling computational performance; Gradišar *et al.* (1999) presented an improved heuristic for the same problem. No proof was given, but we claim that this problem can be easily shown to be NP-hard by a reduction from KNAPSACK (Garey and Johnson, 1979).

In contrast, we will prove that our problem with equal-length pieces is solvable in $O(n \log n)$ time. We begin, however, by giving the practical context and motivation, along with a detailed description, of our roll cutting problem.

2 Problem Description: The Practical Context

The curtaining wholesaler is carrying a wide assortment of curtain material, folded on fabric rolls (we refer to them as *folded rolls*), of different design, color, and fabric quality. The width of a fabric roll is standardized. Of each curtain material, the wholesaler has a number of rolls in stock. Usually, parts of each roll are unusable due to weaving faults. The location of each weaving fault is determined and recorded by scanning the rolls upon delivery by the manufacturers. Hence, each roll effectively can be seen as a number of (virtual) sub-rolls of different lengths.

The retailers in turn usually have a more limited assortment of curtain material on display; they only carry samples and catalogues to show to their customers (usually households). The customer typically orders a number of pieces of a specific curtain material, either all of the same length, or a number or pieces of two, maybe three different lengths. The retailers consolidate these customer orders and order periodically, usually weekly, from the wholesaler, to minimize delivery costs. The wholesaler has split up its service area into five regions and delivers each region once per week. Accordingly, the activities performed by the wholesaler that add value in this supply chain include bulk breaking, stock keeping, cutting, and distributing.

All in all, the cutting problem that the wholesaler is struggling with is a wide variety of (end-customer) orders, with each order consisting of a number of pieces to be cut from the same curtain material. From the wholesaler's perspective, there are few opportunities for consolidating customer orders; for a specific curtain material, there is usually no more than one customer order per day.

With respect to trim loss, the wholesaler seeks a cutting pattern for each order such that the remaining part of each utilized (virtual) roll after cutting is either longer than a given threshold value δ_{\max} , or smaller than a given threshold value δ_{\min} , smaller than the smallest order length. In the first case, the remaining part is still usable for cutting future orders of almost any length; in the second case, the remaining part is so small that it can best be thrown away as it can never be used again. This company procedure precludes the endless stocking of obsolete or almost obsolete rolls. Note that trim loss is not the only concern for the wholesaler with respect to the cutting pattern for each order. In fact, due to the unusable parts of rolls, using a folded roll may result in two or more remaining parts that each need to be folded and stocked.

Two types of problem instances can be distinguished: One in which the pieces to be cut need all be of the same length; and one in which this is not so. In the latter case, usually only a very small number of varying lengths need to be cut.

The second type of problem, the one with different lengths, can easily be shown to be NP hard by a reduction from BIN PACKING (Garey and Johnson, 1979). To see this, note that the question whether it is possible to pack m items of size l_1, \dots, l_m into n bins of length 1 is equivalent to the question whether it is possible to cut m pieces of lengths l_1, \dots, l_m from n rolls R_1, \dots, R_n all of length 1. The values of d_{\max} and d_{\min} are then immaterial. We also claim that this second type of problem (with general d_{\max} and d_{\min}) is solvable by a straightforward pseudopolynomial-time dynamic programming algorithm, just like the bin packing problem is. The development of the algorithm is completely standard and also quite tedious, however, and for this reason we have not included it in our paper.

The remainder of this paper is devoted to the analysis of the first type of problem: the roll cutting problem for pieces of equal length. This problem is much more intriguing than it may seem on first sight. The complexity of this problem is not only of interest of its own, it is also a highly relevant roll cutting problem in this industry. Moreover, the problem itself can represent a variety of other real-life applications, such as, for example, capacity allocation on a batch processing multi-machine station. We prove that this problem is solvable in $O(n \log n)$ time for any number of pieces to be cut, where n is the number of fabric rolls. To arrive at this result, we analyze a certain allocation problem in Sections 3 and 4. In particular, we prove that a special non-trivial case is solvable in polynomial time. Then, in Section 5 we show that our roll cutting problem with pieces of equal length is in turn a special case of this special case.

3 A Taxonomy of Allocation Problems

In this section we introduce and discuss a number of resource allocation problems that fit into the framework of Ibaraki & Katoh (1988). Three of these allocation problems are NP-hard, whereas the remaining two problems are polynomially solvable. In Section 5, we will show that the cutting problem described in Section 2 forms a special case of one of the two polynomially solvable allocation problems. Our most general problem is the following integer allocation problem:

Problem: Integer Allocation Problem (IAP)

Instance: An integer S . Feasible regions $\mathcal{R}_1, \dots, \mathcal{R}_n$, where every region \mathcal{R}_k is specified as a collection of closed intervals over the non-negative real numbers. The left and right endpoints of all intervals are integers, and the left-most interval in \mathcal{R}_k starts at 0.

Question: Do there exist n integers x_1, \dots, x_n with $x_k \in \mathcal{R}_k$ ($k = 1, \dots, n$) that add up to S ?

Note that the stated constraints on the intervals of \mathcal{R}_k are not restrictive at all: As the values x_k must be integers, we may as well restrict ourselves to intervals

whose endpoints are integers. And by appropriately shifting a region \mathcal{R}_k and the value S , we can enforce that the left-most interval in \mathcal{R}_k indeed starts at 0.

The special case of the IAP where every feasible region consists of exactly t closed intervals is denoted by t -IAP. It is straightforward to see that the 1-IAP is solvable in polynomial time. In this paper, we will mainly concentrate on the following formulation of the 2-IAP.

Problem: 2-IAP

Instance: An integer S . Non-negative integers $\alpha_1, \dots, \alpha_n, \beta_1, \dots, \beta_n$, and $\gamma_1, \dots, \gamma_n$ that satisfy $\beta_k \leq \gamma_k$ for $k = 1, \dots, n$.

Question: Do there exist n integers x_1, \dots, x_n that add up to S and that satisfy $x_k \in \mathcal{R}_k$ with $\mathcal{R}_k = [0; \alpha_k] \cup [\alpha_k + \beta_k; \alpha_k + \gamma_k]$ for $k = 1, \dots, n$?

By setting $\alpha_k \equiv 0$ and $\beta_k \equiv \gamma_k$ for $k = 1, \dots, n$, the question simplifies to whether there exist integers $x_k \in \{0, \beta_k\}$ that add up to the goal sum S . This simplification is precisely the classical Subset-Sum problem (Garey & Johnson, 1979). Therefore, problems 2-IAP and IAP both are NP-hard in the ordinary sense. Furthermore, they can be solved in pseudo-polynomial time by dynamic programming.

We introduce some more notations around the 2-IAP. For $k = 0, \dots, n$, we define the values $A_k = \sum_{j=1}^k \alpha_j$ and $B_k = \sum_{j=1}^k \beta_j$. For $0 \leq k \leq \ell \leq n$, we define $C_{k,\ell}$ to be the sum of the k largest values among the ℓ numbers $\gamma_1, \dots, \gamma_\ell$. Note that for $1 \leq k \leq \ell \leq n - 1$ we have

$$C_{k,\ell+1} = \max \{C_{k,\ell}, C_{k-1,\ell} + \gamma_{\ell+1}\}. \tag{1}$$

Definition 1. An instance of 2-IAP is called a **SMALL GAPS instance**, if it satisfies the following two conditions:

$$\alpha_k + \beta_k \leq \alpha_{k+1} + \beta_{k+1} \quad \text{for } 1 \leq k \leq n - 1 \tag{2}$$

and

$$\beta_{\ell+1} \leq A_\ell - A_{k-1} - B_{k-1} + C_{k,\ell} + 1 \quad \text{for } 1 \leq k \leq \ell \leq n - 1. \tag{3}$$

Note that in the 2-IAP the feasible range \mathcal{R}_k for variable x_k consists of the two intervals $[0; \alpha_k]$ and $[\alpha_k + \beta_k; \alpha_k + \gamma_k]$ that are separated by a gap of length β_k . The name “*small gaps instance*” results from the upper bounds on these gap lengths imposed by (3). The conditions in (2) are rather soft, and can be reached by a simple renumbering step. The conditions in (3) put hard upper bounds on the values β_k , and are actually quite restrictive.

The main result of our paper is the following theorem on Small Gaps instances that will be proved in Section 4.

Theorem 2. *Small Gaps instances of 2-IAP can be solved in $O(n^2)$ time.*

Figure 1 summarizes the five optimization problems of this section: A directed arc represents that the lower problem is a special case of the upper problem. Problems with a solid frame are NP-complete, and problems with a dashed frame are solvable in polynomial time.

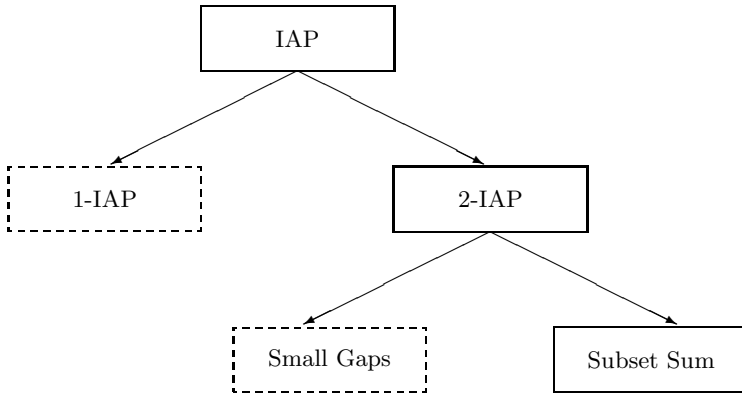


Fig. 1. Five optimization problems. NP-complete problems have a solid frame, polynomially solvable problems have a dashed frame.

4 A Polynomial Time Algorithm for Small Gaps Instances

This section is entirely devoted to a proof of Theorem 2, and to the combinatorics of small gaps instances.

We start with some more notation. For two intervals $U = [u'; u'']$ and $V = [v'; v'']$, we denote by $U + V$ the interval $[u' + v'; u'' + v'']$, that is, the interval consisting of all values $u + v$ with $u \in U$ and $v \in V$. For $0 \leq k \leq \ell \leq n$ we introduce the interval

$$\mathcal{I}_{k,\ell} = [A_k + B_k; A_\ell + C_{k,\ell}]. \tag{4}$$

The following two Lemmas 3 and 4 investigate the combinatorics of small gaps instances. Their statements will lead us to a polynomial time algorithm.

Lemma 3. *Let I be a small gaps instance of 2-IAP, let T be an integer, and let $1 \leq k \leq \ell \leq n - 1$. Then $T \in \mathcal{I}_{k,\ell+1}$ if and only if*

- (a) *there exist integers $T' \in \mathcal{I}_{k,\ell}$ and $x' \in [0; \alpha_{\ell+1}]$ with $T = T' + x'$, or*
- (b) *there exist integers $T'' \in \mathcal{I}_{k-1,\ell}$ and $x'' \in [\alpha_{\ell+1} + \beta_{\ell+1}; \alpha_{\ell+1} + \gamma_{\ell+1}]$ with $T = T'' + x''$.*

Proof. Omitted in this extended abstract. □

Lemma 4. *Let I be a small gaps instance of 2-IAP, and let $0 \leq \ell \leq n$. Then there exist ℓ integers $x_k \in \mathcal{R}_k$ ($k = 1, \dots, \ell$) that add up to the goal sum S , if and only if S is contained in one of the $\ell + 1$ intervals $\mathcal{I}_{k,\ell}$ with $0 \leq k \leq \ell$.*

Proof. The proof is done by induction on ℓ . For $\ell = 0$, there is nothing to prove. For the inductive step from ℓ to $\ell + 1$, the inductive assumption yields that a goal sum S can be written as the sum of $\ell + 1$ integers $x_k \in \mathcal{R}_k$, if and only if S can be written as the sum $T + x_{\ell+1}$ where T is contained in one of the $\ell + 1$ intervals $\mathcal{I}_{k,\ell}$ with $0 \leq k \leq \ell$ and where $x_{\ell+1} \in \mathcal{R}_{\ell+1}$. Hence, the goal sum S must be contained in one of the $\ell + 1$ intervals $\mathcal{I}_{k,\ell} + [0; \alpha_{\ell+1}]$ with $0 \leq k \leq \ell$, or in one of $\ell + 1$ intervals $\mathcal{I}_{k,\ell} + [\alpha_{\ell+1} + \beta_{\ell+1}; \alpha_{\ell+1} + \gamma_{\ell+1}]$ with $0 \leq k \leq \ell$. We will show that the union of these $2(\ell + 1)$ intervals coincides with the union of the $\ell + 2$ intervals $\mathcal{I}_{k,\ell+1}$ with $0 \leq k \leq \ell + 1$.

First, we note that the interval $\mathcal{I}_{0,\ell} + [0; \alpha_{\ell+1}]$ equals the interval $\mathcal{I}_{0,\ell+1}$. Secondly, the interval $\mathcal{I}_{\ell,\ell} + [\alpha_{\ell+1} + \beta_{\ell+1}; \alpha_{\ell+1} + \gamma_{\ell+1}]$ equals the interval $\mathcal{I}_{\ell+1,\ell+1}$. Finally, Lemma 3 yields for $1 \leq k \leq \ell$ that the union of the two intervals $\mathcal{I}_{k,\ell} + [0; \alpha_{\ell+1}]$ and $\mathcal{I}_{k-1,\ell} + [\alpha_{\ell+1} + \beta_{\ell+1}; \alpha_{\ell+1} + \gamma_{\ell+1}]$ coincides with the interval $\mathcal{I}_{k,\ell+1}$. This completes our proof. \square

By setting $\ell := n$ in Lemma 4, we see that in a small gaps instance of 2-IAP the set of representable goal sums can be written as the union of at most $n + 1$ intervals. This linear bound is in strong contrast to the combinatorics of arbitrary (not necessarily small gaps) instances of 2-IAP, where we usually face the union of an *exponential* number of intervals: For example, the instance with $\alpha_k \equiv 0$ and $\beta_k \equiv \gamma_k \equiv 3^k$ for $k = 1, \dots, n$ is not a small gaps instance, and the set of representable goal sums can not be written as the union of less than 2^n intervals.

For small gaps instances, the bound of $n + 1$ on the number of intervals is in fact best possible, as illustrated by the following example: Consider an instance I^* of 2-IAP where $\alpha_k \equiv 1$ and $\beta_k \equiv \gamma_k \equiv n + 2$ for $k = 1, \dots, n$. Then condition (2) is satisfied, since $\alpha_k + \beta_k = n + 3 = \alpha_{k+1} + \beta_{k+1}$ for $1 \leq k \leq n - 1$. Moreover, condition (3) is satisfied since for $1 \leq k \leq \ell < n$

$$\beta_{\ell+1} = n + 2 \leq n + \ell - k + 3 = A_\ell - A_{k-1} - B_{k-1} + C_{k,\ell}.$$

Therefore, instance I^* is a small gaps instance. The set of representable goal sums can be written as the union of the $n + 1$ intervals $\mathcal{I}_{k,n} = [k(n + 1) + k; k(n + 1) + n]$ where $k = 0, \dots, n$, but obviously cannot be written as the union of fewer intervals.

Lemma 5. *For a small gaps instance of 2-IAP, we can decide in $O(n)$ time whether its answer is YES or NO.*

Proof. By Lemma 4, the problem boils down to deciding whether the value S is contained in one of the intervals $\mathcal{I}_{k,n} = [A_k + B_k; A_n + C_{k,n}]$ with $0 \leq k \leq n$. Note that some of these intervals may overlap each other. However, their left endpoints form a non-decreasing sequence, and also their right endpoints form a non-decreasing sequence.

We determine the maximum index m for which $S \geq A_m + B_m$ holds. This can be done in $O(n)$ time by repeatedly computing $A_{j+1} + B_{j+1}$ in $O(1)$ time from $A_j + B_j$. Then S is contained in the union of all intervals $\mathcal{I}_{k,n}$ with $0 \leq k \leq n$,

if and only if S is contained in the interval $\mathcal{I}_{m,n}$, if and only if $S \leq A_n + C_{m,n}$ holds. It is straightforward to determine A_n in $O(n)$ time. The value $C_{m,n}$ can be determined in $O(n)$ time by computing the m -largest number among $\gamma_1, \dots, \gamma_n$ according to the $O(n)$ time algorithm of Blum, Floyd, Pratt, Rivest & Tarjan (1972). \square

Lemma 6. *For a small gaps instance of 2-IAP with answer YES, we can compute the corresponding x_k values in $O(n^2)$ time.*

Proof. In a preprocessing phase, we compute and store all the values A_k for $k = 0, \dots, n$. Since $A_{k+1} = A_k + \alpha_{k+1}$, this can be done in $O(1)$ time per value and in $O(n)$ overall time for all values. Analogously, we determine and store all the values B_k in $O(n)$ overall time. With the help of (1), all values $C_{k,\ell}$ with $0 \leq k \leq \ell \leq n$ can be determined and stored in $O(n^2)$ overall time.

In the main phase, we first determine the interval $\mathcal{I}_{j(n),n}$ that contains the goal sum S . This can be done in linear time according to Lemma 5. Then we distinguish three cases:

1. $j(n) = 0$ holds.

Since $\mathcal{I}_{0,n}$ equals $\mathcal{I}_{0,n-1} + [0; \alpha_n]$, we may fix variable x_n at a value $x_n^* \in [0; \alpha_n]$ such that $S_{n-1} = S - x_n^* \in \mathcal{I}_{0,n-1}$.

2. $j(n) = n$ holds.

Since $\mathcal{I}_{n,n}$ equals $\mathcal{I}_{n-1,n-1} + [\alpha_n + \beta_n; \alpha_n + \gamma_n]$, we may fix variable x_n at a value $x_n^* \in [\alpha_n + \beta_n; \alpha_n + \gamma_n]$ such that $S_{n-1} = S - x_n^* \in \mathcal{I}_{n-1,n-1}$.

3. $1 \leq j(n) \leq n - 1$ holds.

By Lemma 3 interval $\mathcal{I}_{j(n),n}$ equals the union of the two intervals $\mathcal{I}_{j(n),n-1} + [0; \alpha_n]$ and $\mathcal{I}_{j(n)-1,n-1} + [\alpha_n + \beta_n; \alpha_n + \gamma_n]$. We can either fix variable x_n at a value $x_n^* \in [0; \alpha_n]$ such that $S_{n-1} = S - x_n^* \in \mathcal{I}_{j(n),n-1}$, or we can fix variable x_n at $x_n^* \in [\alpha_n + \beta_n; \alpha_n + \gamma_n]$ such that $S_{n-1} = S - x_n^* \in \mathcal{I}_{j(n)-1,n-1}$.

In either case, we define the index $j(n - 1)$ so that $S_{n-1} \in \mathcal{I}_{j(n-1),n-1}$.

In all three cases, we compute the left and right endpoints of the corresponding intervals $\mathcal{I}_{k,n-1}$. Since the endpoints of these intervals only depend on the values A_k, B_k , and $C_{k,n}$, this can be done in $O(1)$ time by using the data stored in the preprocessing phase.

All in all, we have reduced the problem from an n -dimensional instance with variables x_1, \dots, x_n and goal sum $S \in \mathcal{I}_{j(n),n}$ to an $(n - 1)$ -dimensional instance with variables x_1, \dots, x_{n-1} and goal sum $S_{n-1} \in \mathcal{I}_{j(n-1),n}$. We repeat this procedure step by step, where the k th step reduces the dimension to $n - k$ by fixing variable x_{n-k+1} at some value in \mathcal{R}_{n-k+1} . Since the computation time in every step is $O(1)$, the overall running time of the main phase is $O(n)$. \square

This completes the proof of Theorem 2. Let us briefly sketch (without proofs) how the running time in Lemma 6 can be improved to $O(n \log n)$: The main issues are the computations around the values $C_{k,\ell}$. These computations contribute $O(n^2)$ to the preprocessing phase, whereas the rest of the preprocessing phase is $O(n)$. In the main phase, there are $O(n)$ steps that cost $O(1)$ time per step.

Every step in the main phase uses one or two of these values $C_{k,\ell}$. The values $C_{k,\ell}$ used in consecutive steps are closely related; if some step uses $C_{j^{(k)},k}$, then its successor step can only use $C_{j^{(k)}-1,k-1}$ and/or $C_{j^{(k)},k-1}$. The speed-up results from using a balanced binary search tree as an underlying data structure (see for instance Cormen, Leiserson, & Rivest, 1990). In the preprocessing phase, all numbers $\gamma_1, \dots, \gamma_n$ are inserted into this binary search tree; this costs $O(n \log n)$ preprocessing time. In every step of the main phase, we delete one number γ_k from the search tree, and appropriately update the information stored in the internal vertices of the tree. This costs $O(\log n)$ time per update, and also allows us to determine the relevant values $C_{j^{(k)}-1,k-1}$ and/or $C_{j^{(k)},k-1}$ in $O(\log n)$ time. All in all, this yields a time complexity of $O(n \log n)$ for the main phase.

5 The Roll Cutting Problem with Equal-Length Pieces

We return to the roll cutting problem described in Section 2. An instance of our roll cutting problem consists of n rolls R_1, \dots, R_n of (real, non-trivial, not necessarily integer) lengths p_1, \dots, p_n from which S pieces of length 1 need to be cut. Note that we have normalized the piece length at 1. Furthermore, there are two real thresholds δ_{\min} and δ_{\max} with $\delta_{\min} < \delta_{\max}$. From each roll R_k , we may cut a number x_k of pieces of length 1, provided that one of the following two situations occurs:

- The remaining length of roll R_k satisfies $p_k - x_k \leq \delta_{\min}$. Then the remaining length is sufficiently small, and the roll can be thrown away without incurring a major trim loss.
- The remaining length of roll R_k satisfies $p_k - x_k \geq \delta_{\max}$. Then the remaining length is sufficiently large, and the roll can be reused for future cuttings.

Hence, the only forbidden values for x_k are those with $\delta_{\min} < p_k - x_k < \delta_{\max}$. The roll cutting problem is to decide whether it is possible to find integral values x_1, \dots, x_n such that:

$$\sum_{k=1}^n x_k = S, \quad \text{and} \tag{5}$$

$$p_k - x_k \leq \delta_{\min} \quad \text{or} \quad p_k - x_k \geq \delta_{\max}, \quad \text{for all } k = 1, \dots, n, \tag{6}$$

$$p_k - x_k \geq 0, \quad \text{for all } k = 1, \dots, n. \tag{7}$$

Next, we will show that the roll cutting problem specified in (5)–(7) can be considered as a special case of the 2-IAP. To see this, we define for $k = 1, \dots, n$ the integer values

$$\begin{aligned} \alpha_k &= \lfloor p_k - \delta_{\max} \rfloor \\ \beta_k &= \lceil p_k - \delta_{\min} \rceil - \lfloor p_k - \delta_{\max} \rfloor \\ \gamma_k &= \lfloor p_k \rfloor - \lfloor p_k - \delta_{\max} \rfloor. \end{aligned}$$

Here we use $\lfloor z \rfloor$ to denote the largest integer less or equal to z , and $\lceil z \rceil$ to denote the smallest integer greater or equal to z . Hence, the number x_k of pieces to be cut from roll R_k should either be smaller than or equal to α_k (to ensure that the remaining roll has length at least δ_{\max}), or it should be larger than or equal to $\alpha_k + \beta_k$ (to ensure that the remaining has length at most δ_{\min}). The value $\alpha_k + \gamma_k$ refers to the maximum number of pieces that can be cut from roll R_k .

Note that for some rolls R_k it might happen that $\gamma_k < \beta_k$. For instance, if $\delta_{\min} = 0.4$ and $\delta_{\max} = 5$ then a roll R_k with length $p_k = 10.5$ will yield the values $\alpha_k = 5$, $\beta_k = 6$, $\gamma_k = 5$, and then $\gamma_k < \beta_k$ holds. This is an exceptional case that can only occur for $\delta_{\min} < 1$. In such a case the interval $[\alpha_k + \beta_k; \alpha_k + \gamma_k]$ becomes empty, and the feasible region for x_k boils down to the single interval $[0; \alpha_k]$. Rolls R_k with $\gamma_k < \beta_k$ are called *exceptional* roles, and the remaining rolls R_k with $\gamma_k \geq \beta_k \geq 0$ are called *standard* roles. The standard rolls form the so-called *core* instance of the roll cutting instance. We assume that the rolls in the core instance are indexed in order of non-decreasing lengths, so that they satisfy:

$$\delta_{\max} \leq p_1 \leq p_2 \leq \dots \leq p_n. \tag{8}$$

Lemma 7. *A roll cutting instance with goal value S possesses a solution, if and only if S can be written as $S = S_1 + S_2$, where*

- S_1 is an integer with $0 \leq S_1 \leq E$, where E is the sum of the α_k values of all the exceptional rolls;
- S_2 is an integer, such that the core instance with goal value S_2 possesses a solution. □

In other words, solving a roll cutting instance essentially boils down to solving the corresponding core instance. The following two lemmas show that every core instance in fact is a small gaps instance.

Lemma 8. *For every core instance, there exist two non-negative integers β^* and γ^* with $\beta^* \leq \gamma^*$ such that $\beta_k \in \{\beta^*, \beta^* + 1\}$ and $\gamma_k \in \{\gamma^*, \gamma^* + 1\}$ holds for all $1 \leq k \leq n$.*

Proof. Omitted in this extended abstract. □

Lemma 9. *Every core instance is a small gaps instance.*

Proof. Omitted in this extended abstract. □

Once we know that core instances are small gap instances, the theory developed in Section 4 together with Lemma 7 easily yields the following central result.

Theorem 10. *Roll cutting instances can be solved in $O(n \log n)$ time.*

Proof. First of all, we determine the corresponding core instance in $O(n)$ time. Then a simple $O(n \log n)$ sorting step ensures the chain of inequalities in (8) for the core instance.

Our next goal is to speed up the preprocessing phase in Lemma 6: Most steps in this preprocessing phase take $O(n)$ time, except for the expensive computation of the values $C_{k,\ell}$ which takes $O(n^2)$ time. We replace this expensive computation step by the following faster computation step: By Lemma 8, the γ_k can only take two values γ^* and $\gamma^* + 1$. We define $F[k]$ ($k = 1, \dots, n$) to be the number of occurrences of the value $\gamma^* + 1$ among the integers $\gamma_1, \dots, \gamma_k$. It is straightforward to compute all values $F[k]$ in $O(n)$ overall time (if $\gamma_{k+1} = \gamma^*$, then $F[k+1] = F[k]$; and if $\gamma_{k+1} = \gamma^* + 1$, then $F[k+1] = F[k] + 1$). Note that $C_{k,\ell} = k \cdot \gamma^* + \min\{k, F[k]\}$ holds for all $0 \leq k \leq \ell \leq n$. Consequently, whenever the main phase needs to access a value $C_{k,\ell}$, we can compute it in $O(1)$ time from the precomputed values $F[k]$.

Lemma 4 and Lemma 7 imply that the roll cutting instance with goal value S possesses a solution, if and only if there exists an integer S_2 with $S - E \leq S_2 \leq S$, that is contained in one of the intervals $\mathcal{I}_{k,n} = [A_k + B_k; A_n + C_{k,n}]$ with $0 \leq k \leq n$. As a consequence of the above preprocessing phase, we can compute all intervals $\mathcal{I}_{k,n}$ in $O(n)$ time and intersect them with the interval $[S - E; S]$. If all intersections are empty, then there is no solution. If one of these intersections is non-empty, then there exists a solution. In that case, the corresponding x_k values can be computed in $O(n)$ overall time as described in Lemma 6. \square

References

1. M. BLUM, R.W. FLOYD, V.R. PRATT, R.L. RIVEST, AND R.E. TARJAN (1972). Time bounds for selection. *Journal of Computer and System Sciences* 7, 448–461.
2. T.H. CORMEN, C.E. LEISERSON, AND R.L. RIVEST (1990). *Introduction to Algorithms*. MIT Press.
3. M.R. GAREY AND D.S. JOHNSON (1979). *Computers and Intractability*. W.H. Freeman and Co., New York.
4. P.C. GILMORE AND R.E. GOMORY (1961). A linear programming approach to the cutting stock problem. *Operations Research* 9, 849–859.
5. P.C. GILMORE AND R.E. GOMORY (1961). A linear programming approach to the cutting stock problem. Part II. *Operations Research* 11, 863–888.
6. M. GRADIŠAR, J. JESENKO AND G. RESINOVIČ (1997). Optimization of roll cutting in clothing industry. *Computers and Operations Research* 24, 10, 945–953.
7. M. GRADIŠAR, M. KLJAJIĆ, G. RESINOVIČ, AND J. JESENKO (1999). A sequential heuristic procedure for one-dimensional cutting. *European Journal of Operational Research* 114, 3, 557–568.
8. T. IBARAKI AND N. KATOH (1988). *Resource allocation problems: Algorithmic approaches*. MIT Press, Cambridge, USA.
9. P.Y. WANG AND G. WÄSCHER (2002). Editorial – cutting and packing. *European Journal of Operational Research* 141, 2, 239–240.

Space Efficient Algorithms for the Burrows-Wheeler Backtransformation

Ulrich Lauther and Tamás Lukovszki

Siemens AG, Corporate Technology,
81730 Munich, Germany
{Ulrich.Lauther, Tamas.Lukovszki}@siemens.com

Abstract. The Burrows-Wheeler transformation is used for effective data compression, e.g., in the well known program bzip2. Compression and decompression are done in a block-wise fashion; larger blocks usually result in better compression rates. With the currently used algorithms for decompression, $4n$ bytes of auxiliary memory for processing a block of n bytes are needed, $0 < n < 2^{32}$. This may pose a problem in embedded systems (e.g., mobile phones), where RAM is a scarce resource. In this paper we present algorithms that reduce the memory need without sacrificing speed too much.

The main results are: Assuming an input string of n characters, $0 < n < 2^{32}$, the reverse Burrows-Wheeler transformation can be done with $1.625 n$ bytes of auxiliary memory and $O(n)$ runtime, using just a few operations per input character. Alternatively, we can use n/t bytes and $256 t n$ operations. The theoretical results are backed up by experimental data showing the space-time tradeoff.

1 Introduction

The Burrows-Wheeler transformation (BWT) [6] is at the heart of modern, very effective data compression algorithms and programs, e.g., bzip2 [13]. BWT-based compressors usually work in a block-wise manner, i.e., the input is divided into blocks and compressed block by block. Larger block sizes tend to result in better compression results, thus bzip2 uses by default a block size of 900,000 bytes and in its low memory mode still 100,000 bytes. The standard algorithm for decompression (reverse BWT) needs auxiliary memory of 4 bytes per input character, assuming 4-byte computer words and thus $n < 2^{32}$. This may pose a problem in embedded systems (say, a mobile phone receiving a software patch over the air interface) where RAM is a scarce resource. In such a scenario, space requirements for compression ($8n$ bytes when a suffix array [10] is used to calculate the forward BWT) is not an issue, as compression is done on a full fledged host. In the target system, however, cutting down memory requirements may be essential.

1.1 The BWT Backtransformation

We will not go into details of the BW-transformation here, as it has been described in a number of papers [2,4,6,7,8,11] and tutorials [1,12] nor do we give a

proof of the reverse BWT algorithm. Instead, we give the bare essentials needed to understand the problem we solve in the following sections. The BWT (conceptually) builds a matrix whose rows contain n copies of the n character input string, row i rotated i steps. The n strings are then sorted lexicographically and the last column is saved as the result, together with the "primary index", i.e., the index of the row that contains - after sorting - the original string. The first column of the sorted matrix is also needed for the backtransformation, but it needs not to be saved, as it can be reconstructed by sorting the elements of the last column. (Actually, as we will see, the first column is also needed only conceptually.)

Figure 1 shows the first and last columns resulting from the input string "CARINA". The arrow indicates the primary index. Note that we have numbered the occurrences of each character in both columns, e.g., row 2 contains the occurrence 0 of character "A" in L , row 5 contains occurrence 1. We call these numbers the *rank* of the character within column L .

	F	L rank	base
	0 A 0	N 0	A : 0
	1 A 1	C 0	C : 2
⇒	2 C 0	A 0	I : 3
	3 I 0	R 0	N : 4
	4 N 0	I 0	R : 5
	5 R 0	A 1	

Fig. 1. First (F) and last (L) column for the input string "CARINA"

To reconstruct the input string, we start at the primary index in L and output the corresponding character, "A", whose rank is 0. We look for A_0 in column F , find it at position 0 and output "N". Proceeding in the same way, we get "I", "R", "A", and eventually "C", i.e., the input string in reverse order. The position in F for a character/rank pair can easily be found if we store for each character of the alphabet the position of its first occurrence in F ; these values are called *base* in Figure 1.

This gives us a simple algorithm when the vectors *rank* and *base* are available:

```
int h = primary_index;
for (int i = 0; i < n; i++) {
    char c = L[h];
    output(c);
    h = base[c] + rank[h];
}
```

The *base*-vector and *rank* can easily be calculated with one pass over L and another pass over all characters of the alphabet. (We assume an alphabet of 256 symbols throughout this paper.)

```
for (int i = 0; i < 256; i++) base[i] = 0;
for (int i = 0; i < n; i++) {
    char c = L[i];
    rank[i] = base[c];
    base[c]++;
}
```

```

int total = 0;
for (int i = 0; i < 256; i++) {
    int h = base[i];
    base[i] = total;
    total += h;
}

```

These algorithms need $O(n)$ space (n words for the *rank*-vector) and $O(n)$ time. Alternatively, we could do without precalculation of rank-values and calculate $rank[h]$ whenever we need it, by scanning L and counting occurrences of $L[h]$. This would give us $O(1)$ space and $O(n^2)$ time.

The question, now, is: is there a data structure that needs significantly less than n words without increasing run time excessively?

In this paper we present efficient data structures and algorithms solving following problems:

Rank Searching: The input must be preprocessed into a data structure, such that for a given index i , it supports a query for $rank(i)$. This query is referred to as rank-query.

Rank-Position Searching: The input must be preprocessed into a data structure, such that for a given character c and rank r , it supports a query for index i , such that $rank(i) = r$. This query is referred to as rank-position-query. (This allows traversing L and F in the direction opposite to that discussed so far, producing the input string in forward order).

1.2 Computation Model

As computation model we use a random access machine (RAM) (see e.g., in [3]). The RAM allows indirect addressing, i.e., accessing the value at a relative address, given by an integer number, in constant time. In this model it is also assumed that the length of the input n can be stored in a computer word. Additionally, we assume that the size $|A|$ of the alphabet A is a constant, and particularly, $|A| - 1$ can be stored in a byte. Furthermore, we assume that a bit shift operation in a computer word, word-wise *and* and *or* operations, converting a bit string stored in a computer word into an integer number and vice-versa and algebraic operations on integer numbers ('+', '-', '*', '/', 'mod', where '/' denotes the integer division with remainder) are possible in constant time.

1.3 Previous Results

In [14] Seward describes a slightly different method for the reverse BWT by handling the so-called transformation vector in a more explicit way. He presents several algorithms and experimental results for the reverse BWT and answering rank-queries (more precisely, queries "how many symbols x occur in column L up to position i ?", without the requirement $L[i] = x$). A rigorous analysis of the algorithms is omitted in [14]. The algorithms described in [14], **basis** and **bw94** need $5n$ bytes of memory storage and support a constant query time; algorithm

MergedTL needs $4n$ bytes if n is limited to 2^{24} and supports a constant query time. The algorithm **indexF** needs $2.5n$ bytes if $n < 2^{20}$ and $O(\log |A|)$ query time. The algorithms **tree** and **treeopt** build 256 trees (one for each symbol) on sections of the vector L . They need $2n$ and $1.5n$ bytes, respectively, if $n < 2^{20}$ and support $O(\log(n/\Delta) + c_x \Delta)$ query time, where Δ is a given parameter depending on the allowed storage and c_x is a relatively big multiplier which can depend on the queried symbol x .

1.4 Our Contributions

We present a data structure which supports answering a rank-query $Q(i)$ in $O(1)$ time using $n(\frac{\ell-1}{8} + \frac{w|A|}{2^\ell})$ bytes, where w denotes the length of a computer word in bytes, and $|A|$ is the size of the alphabet. If $|A| \leq 256$ and $w = 4$ (32 bit words), by setting $\ell \in \{12, 13\}$, we obtain a data structure of $\frac{13}{8}n$ or 1.625 bytes. For $w = 2$ we get a data structure of $\frac{25}{16}n$ or 1.5625 bytes. Thus, the space requirement is strictly less than that of the trivial data structure, which stores the rank for each position as an integer in a computer word and that of the methods in [14] with constant query time. The preprocessing needs $O(n)$ time and $O(|A|)$ working storage.

We also present data structures of n bytes, where we allow at most $L = 2^9$ sequential accesses to a data block of L bytes. Because of caching and hardware prefetching mechanism of todays processors, with this data structure we obtain a reasonable query time.

Furthermore, we present a data structure, which supports answering a rank-query $Q(i)$ in $O(t)$ time using t random accesses and $c \cdot t$ sequential accesses to the memory storage, where c is a constant, which can be chosen, such that the speed difference between non-local (random) accesses and sequential accesses is utilized optimally. The data structure needs $\frac{n(8+|A|\log ct)}{8ct} + \frac{n|A|w}{ct^2}$ bytes. For $t = \omega(1)$, this results in a sub-linear space data structure, e.g., for $t = \Theta(n^{1/d})$ we obtain a data structure of $\frac{1}{d}n^{1-1/d}|A|(1 + o(1))$ bytes. The preprocessing needs $O(n)$ time and $O(|A|)$ working storage.

After this, we turn to the inverse problem, the problem of answering rank-position-queries. We present a data structure of $n(\frac{|A|(\ell+8w)}{2^\ell} + \ell)$ bits, which supports answering rank-position-queries in $O(\log(n/2^\ell))$ time. The preprocessing needs $O(n)$ time and $O(|A| + 2^\ell)$ working storage. For $\ell = 13$, we obtain a data structure of $14\frac{3}{8} \cdot n$ bits.

Finally, we present experimental results, that show that our algorithms perform quite well in practice. Thus, they give significant improvement for decompression in embedded devices for the mentioned scenarios.

1.5 Outline of the Paper

In Section 2 we describe various data structures supporting rank-queries. Section 3 is dedicated to data structures for rank-position-queries. In Section 4 we present experimental results for the reverse BWT. Finally, in Section 5 we give conclusions and discuss open problems.

2 Algorithms for Rank-Queries

Before we describe the algorithm we need some definitions. For a string S of length n (i.e. of n symbols) and integer number i , $0 \leq i < n$ we denote by $S[i]$ the symbol of S at the position (or index) i , i.e., the index counts from 0. For a string S and integers $0 \leq i, j < n$, we denote by $S[i..j]$ the substring of S starting at index i and ending at j , if $i \leq j$, and the empty string if $i > j$. $S[i..i] = S[i]$ is the symbol at index i . The rank of symbol $S[i]$ at the position i is defined as $rank(i) := |\{j : 0 \leq j < i, S[j] = S[i]\}|$, i.e., the rank of the k th occurrence of a symbol is $k - 1$.

2.1 A Data Structure of $\frac{13}{8} \cdot n$ Bytes

We divide the string S into $n' = \lceil n/L \rceil$ blocks, each of $L = 2^\ell$ consecutive symbols (bytes). L will be determined later. (The last block may contain less than L symbols.) The j th block $B[j]$ of the string S , $0 \leq j < n'$, starts at the position $j \cdot L$, i.e. $B[j] = S[j \cdot L .. \min\{n, (j + 1)L\} - 1]$.

In our data structure, for each block $B[j]$, $0 \leq j < n'$ and each symbol $x \in A$, we store an integer value $b[j, x]$, which contains the number of occurrences of symbol x in blocks $B[0], \dots, B[j]$, i.e. in $S[0..L(j + 1) - 1]$. In the following we assume $b[-1, x] = 0$, $x \in A$, but there is no need to store these values explicitly. For storing the values $b[j, x]$, $0 \leq j < n'$, $x \in A$, we need $n'|A| = \lceil n/L \rceil |A|$ computer words, i.e. $\lceil n/L \rceil 8w|A|$ bits.

Additionally, for each index i , $0 \leq i < n$, we store the *reduced* rank $r[i]$ of the symbol $x = S[i]$, $r[i] = rank(i) - b[i/L - 1, x]$. Then a rank query $Q(i)$ obviously can be answered by reporting the value $b[i/L - 1, x] + r[i]$. Note that $0 \leq r[i] < L$, and thus, each $r[i]$ can be stored in ℓ bits. We can save an additional bit (or equivalently, double the block size), if we define the reduced rank $r[i]$ in a slightly different way:

$$r[i] = \begin{cases} rank(i) - b[i/L - 1, x] & \text{if } i \bmod L < L/2, \\ b[i/L, x] - rank(i) - 1 & \text{otherwise.} \end{cases}$$

For storing the whole vector r , we need $n \cdot (\ell - 1)$ bits.

Storage Requirement: The storage requirement for storing $r[i]$ and $b[j, x]$, $0 \leq i < n$, $0 \leq j < n'$, $x \in A$ is $n(\ell - 1 + \frac{8w|A|}{L}) = n(\ell - 1 + 8w|A|2^{-\ell})$ bits. We obtain the continuous minimum of this expression at the point, in which the derivative is 0. Thus, we need $1 + 8w|A|2^{-\ell}(-\ln 2) = 0$. After reorganizing this equality, we obtain

$$2^{-\ell} = \frac{1}{8w|A| \ln 2} \quad \text{and thus} \\ \ell = \log(8w|A|) + \log \ln 2.$$

Since $|A| \leq 256$, for $w = 4$ (32 bit words), the continuous minimum is reached in $\ell \approx 3 + 2 + 8 - 0.159$. Setting $\ell = 12$ or $\ell = 13$ (and the block size $L = 4096$ or $L = 8192$, respectively), the size of the data structure becomes $\frac{13}{8}n$ bytes.

Computing the Values $b[j, x]$ and $r[i]$: The values $b[j, x]$, $0 \leq j < n'$, $x \in A$, can be computed in $O(n)$ time using $O(|A|)$ space by scanning the input as follows. We maintain an array b_0 of $|A|$ integers, such that after processing the i th symbol of the input string S , $b_0[x]$ will contain the number of occurrences of symbol x in $S[0..i]$. At the beginning of the algorithm we initialize each element of b_0 to be 0. We can maintain the above invariant by incrementing the value of $b_0[x]$, when we read the symbol x . After processing the symbol at a position i with $i \equiv -1 \pmod L$, i.e. after processing the last symbol in a block, we copy the array b_0 into $b[i/L]$.

The values of $r[i]$, $0 \leq i < n$, are computed in a second pass, when all the b -values are known. Here, we maintain an array r_0 of $|A|$ integers, such that just before processing the i th symbol of the input string S , $r_0[x]$ will contain the number of occurrences of symbol x in $S[0..i - 1]$. Thus, we can set $r[i] = r_0[x] - b[i/L - 1, x]$ or $r[i] = b[i/L] - r_0[x] - 1$, respectively. At the beginning of the algorithm we initialize each element of r_0 to be 0. We can maintain the above invariant by incrementing the value of $r_0[x]$, after reading the symbol x .

Clearly, the above algorithm needs $O(n)$ time and $O(|A|)$ working storage (for the arrays b_0 and r_0).

Answering a Query $Q(i)$: If we have the correct values for $r[i]$ and $b[j, x]$, $0 \leq i < n$, $0 \leq j < n'$, $x \in A$, then a query $Q(i)$, $0 \leq i < n$ can be answered easily by determining the symbol $x = S[i]$ in the string S and combining the reduced rank $r[i]$ with the appropriate b -value:

$$rank(i) = \begin{cases} r[i] + b[i/L, x] & \text{if } i \pmod L < L/2, \\ b[i/L + 1, x] - r[i] - 1 & \text{otherwise.} \end{cases}$$

This sum can be computed using at most 2 memory accesses and a constant number of unit time operations on computer words. Summarizing the results of this section we obtain the following.

Theorem 1. *Let S be a string of length n . S can be preprocessed into a data structure which supports answering a rank query $Q(i)$ in $O(1)$ time. The data structure uses $n(\ell - 1 + 8w|A|/2^\ell)$ bits, where w is the number of bytes in a computer word. For $|A| \leq 256$, $w = 4$, and $\ell \in \{12, 13\}$, the size of the data structure is $\frac{13}{8}n$ bytes. The preprocessing needs $O(n)$ time and $O(|A|)$ working storage.*

Remark: If the maximum number of occurrences of any symbol in the input is smaller than the largest integer that can be stored in a computer word, i.e. $n < 2^p$ and $p < 8w$, then we can store the values of $b[j, x]$ using p bits instead of a complete word. Then the size of the data structure is $n(\ell - 1 + \frac{p|A|}{2^\ell})$ bits. For instance, for $p = 16$ we obtain a data structure of $n\frac{25}{16}$ bytes, and for $p = 24$ one of $n\frac{51}{32}$ bytes.

Utilizing Processor Caching – Data Structure of $\leq n$ Bytes: The processors in modern computers use caching, hardware prefetching, and pipelining techniques, which results in significantly higher processing speed, if the algorithm

accesses consecutive computer words than in the case of non-local accesses (referred to as random accesses). In case of processor caching, when we access a computer word, a complete block of the memory will be moved into the processor cache. For instance, in Intel Pentium 4 processors, the size of such a block (the so-called L2 cache line size) is 128 bytes (see, e.g. [9]). Using this feature, we also obtain a fast answering time, if we get rid of storing the values of $r[i]$, but instead of this, we compute $r[i]$ during the query by scanning the block (of L bytes) containing the string index i . More precisely, it is enough to scan the half of the block: the lower half in increasing order of indices, if $i \bmod L < L/2$, and the upper half in decreasing order of indices, otherwise. In that way we obtain a data structure of size $n|A|w/L$ bytes.

Theorem 2. *Let S be a string of length n . S can be preprocessed into a data structure $D(S)$, which supports answering a rank query $Q(i)$ by performing 1 random access to $D(S)$ (and to S) and at most $L/2$ sequential accesses to S . The data structure uses $n|A|w/L$ bytes, where w is the length of a computer word in bytes. The preprocessing needs $O(n)$ time and $O(|A|)$ working storage.*

For $|A| \leq 256$, $w = 4$ and $L = 2^{10}$, the size of the data structure is n bytes. If $n < 2^p$, $p < 8w$, then we get a data structure of n bytes by using a block size of $L = p|A|/8$ bytes.

2.2 A Sub-linear Space Data Structure

In this Section we describe a sub-linear space data structure for supporting rank-queries in strings in $O(t)$ time for $t = \omega(1)$.

Similarly to the data structure described in Section 2.1, we divide the string S into $n^* = \lceil \frac{n}{c \cdot t} \rceil$ blocks, each of size $L = c \cdot t$ bytes, where c is a constant. (The constant c can be determined for instance, such that the effect of processor caching and pipelining is being exploited optimally).

We store the values $b^*[j, x]$, where $b^*[j, x]$ is the number of occurrence of symbol $x \in A$ in the j th block. The value of $b^*[j, x]$ is stored as a bit-string of $\lfloor \log ct \rfloor + 1$ bits. Note that $0 \leq b^*[j, x] \leq L$. Let $b'[j, x] = b^*[j, x] \bmod L$. Then $0 \leq b'[j, x] \leq L$, and thus, each $b'[j, x]$ can be stored in $\log ct + 1$ bits. Furthermore, the value of $b^*[j, x]$ can be reconstructed from the value of $b'[j, x]$ in constant time: if $b'[j, x] = 0$ and an arbitrary symbol (say the first symbol) of block $B[j]$ is equal to x , then $b^*[j, x] = L$, otherwise $b^*[j, x] = b'[j, x]$. For this test, we store in $c[j]$ the first symbol in $B[j]$, for each block $B[j]$. Storing $b^*[j, x]$ and $c[j]$, for $0 \leq j < \lceil \frac{n}{c \cdot t} \rceil$ and $x \in A$ needs $\frac{n \cdot (8+|A| \log cn)}{c \cdot t}$ bits, i.e. $\frac{n \cdot (8+|A| \log cn)}{8 \cdot c \cdot t}$ bytes.

Additionally to the linear space data structure, the blocks are organized in $\hat{n} = \lceil \frac{n}{c \cdot t^2} \rceil$ super-blocks, such that each super-block contains t consecutive blocks. We compute the values of $\hat{b}[k, x]$, for $0 \leq k < \lceil \frac{n}{c \cdot t^2} \rceil$ and $x \in A$, such that $\hat{b}[k, x]$ contains the number of occurrences of symbol x in the super-blocks $0, \dots, k$. These values are stored as integers. Storing all values needs $\frac{n \cdot |A|}{c \cdot t^2}$ computer words.

The values of $b^*[j, x]$ and $\hat{b}[k, x]$, $0 \leq j < n^*$, $0 \leq k < \hat{n}$, $x \in A$ can be computed in $O(n)$ time using $O(|A|)$ space by scanning the input string S in a similar way as in Section 2.1.

Answering a Query: Using this data structure, a query $Q(i)$ can be answered as follows. Let \hat{B} be the super-block containing the query position i , i.e. \hat{B} is the super-block with index $k = i/(c \cdot t^2)$. Let B be the block containing the position i , i.e. B is the block with index $j^* = i/(c \cdot t)$. Let $x = S[i]$. The query $Q(i)$ can be answered by summing $\hat{b}[k - 1, x]$ and the values of $b^*[j, x]$, for each index j of a block in the super-block \hat{B} , such that $j < j^*$, i.e. $(i/(c \cdot t^2)) \cdot t \leq j < j^*$. Then we scan the block B and compute the rank $r[i]$ of $S[i]$ in block B during the query time (by scanning the half of the block, as described in the previous section). Then

$$\text{rank}(i) = r[i] + \hat{b}[i/(c \cdot t^2) - 1, x] + \sum_{j=(i/(c \cdot t^2)) \cdot t}^{i/(c \cdot t) - 1} b^*[j, x].$$

Since we have one random access to the value $\hat{b}[k - 1, x]$, at most t random accesses to the values of $b^*[j, x]$, and $c \cdot t/2$ sequential accesses to the input string, we can perform a rank-query in $O(t)$ time.

Summarizing the description, we obtain:

Theorem 3. *Let S be a string of length n . S can be preprocessed into a data structure which supports answering a rank-query $Q(i)$ in $O(t)$ time. The data structure uses $n \left(\frac{|A|(8 + \log ct)}{ct} + \frac{8w|A|}{ct^2} \right)$ bits. For $t = \omega(1)$, it uses $\left(\frac{n \cdot (8 + |A| \log ct)}{ct} \right) (1 + o(1))$ bits. The preprocessing needs $O(n)$ time and $O(|A|)$ working storage.*

Note, that for a block size of $L = ct = 2^9$ bytes and $t \geq \frac{16}{7}w$, we obtain a data structure of n bytes. With other words, we can guarantee a data structure of n bytes using smaller blocks than in Section 2.1 to the cost of t random storage accesses.

Corollary 1. *Let S be a string of length n . S can be preprocessed into a data structure of n bytes, which for $t \geq \frac{16}{7}w$, supports answering a rank-query $Q(i)$ in $O(t)$ time using t random accesses and $ct/2$ sequential accesses. The preprocessing needs $O(n)$ time and $O(|A|)$ working storage.*

If we allow in Theorem 3, for instance, a query time $O(n^{1/d})$, then we can store the value of $b^*[j, x]$ in $\frac{\log n}{d}$ bits and the whole matrix b^* in $\frac{1}{d}n^{1-1/d}|A|$ computer words.

Corollary 2. *Let S be a string of length n . S can be preprocessed into a data structure which supports answering a rank-query $Q(i)$ in $O(n^{1/d})$ time. The data structure uses $\frac{1}{d}n^{1-1/d}|A|(1 + o(1))$ computer words. The preprocessing needs $O(n)$ time and $O(|A|)$ working storage.*

3 An Algorithm for Rank-Position-Queries

In this Section we consider the inverse problem of answering rank-queries, the problem of answering rank-position-queries. A rank-position-query $Q^*(x, k)$, $x \in$

A , $r \in \mathbb{N}_0$, reports the index $0 \leq i < n$ in the string S , such that $S[i] = x$ and $\text{rank}(i) = k$, if such an index exist, and "no index" otherwise. We show, how to preprocess S into a data structure of $n(\frac{|A|w}{L} + \frac{\ell}{8})$ bytes, which supports answering rank-position-queries in $O(\log(n/L))$ time.

We divide the string S into $n' = \lceil n/L \rceil$ blocks, each containing $L = 2^\ell$ consecutive symbols. The rank-position-query $Q^*(x, k)$ will work in two steps:

1. Find the block $B[j]$, which contains the index of the k th occurrence of x in S , and determine k_0 the overall number of occurrences of x in the blocks $B[0], \dots, B[j - 1]$.
2. Find the relative index i' of the k' ($:= k - k_0$)th occurrence of x within $B[j]$, if i' exists, and return with index $i = i' + jL$, and return with "no index" otherwise.

Data Structure for Step 1: For each block $B[j]$, $0 \leq j < n'$ and each symbol $x \in A$, we store an integer value $b[j, x]$, which contains the overall number of occurrences of symbol x in blocks $B[0], \dots, B[j - 1]$, i.e. in $S[0 .. jL - 1]$. For storing the values $b[j, x]$, $0 \leq j < n'$, $x \in A$, we need $n'|A| = \lceil n/L \rceil |A|$ computer words, i.e. $\lceil n/L \rceil 8w|A|$ bits. The values of all $b[j, x]$ can be computed in $O(n)$ time using $O(|A|)$ working storage.

Let j be the largest number, such that $b[j, x] < k$. Then $B[j]$ is the block, which contains the index of the k th occurrence of x , if S contains at least k occurrences of x , and $B[j]$ is the last block otherwise. We set $k_0 = b[j, x]$. Using this data structure, Step 1 can be performed in $O(\log(n/L))$ time by logarithmic search for determining j .

Data Structure for Step 2: For each block $B[j]$, $0 \leq j < n'$ and each symbol $x \in A$, we store a sorted list $p[j, x]$ of relative positions of the occurrences of symbol x in $B[j]$, i.e. if $B[j][i'] = x$, $0 \leq i' < L$, then $p[j, x]$ contains an element for i' . The relative index i' of the k' th occurrence of x in $B[j]$ is the k' th element of the list $p[j, x]$. Note, that the overall number of list elements for a block $B[j]$ is L and each relative position can be stored in ℓ bits. Therefore, we can store all lists for $B[j]$ in an array $a[j]$ of L elements, where each element of $a[j]$ consists of ℓ bits. Additionally, for each $0 \leq j < n'$ and $x \in A$, we store in $s[j, x]$ the start index of $p[j, x]$ in $a[j]$. Since $0 \leq s[j, x] < L$, $s[j, x]$ can be stored in ℓ bits. Therefore, the storage requirement of storing $a[j]$ and $s[j, x]$, $0 \leq j < n'$, $x \in A$, is $n\ell + n\ell|A|/L$ bits. These values can be computed in $O(n)$ time using $O(L + |A|)$ working storage. (First we scan $B[j]$ and build linked lists for $p[j, x]$: for $0 \leq i' < n'$, if $B[j][i'] = x$, then we append a list element for i' to $p[j, x]$. Then we compute $a[j]$ and $s[j, x]$ for each $x \in A$ from the linked lists.) Let $k' = k - k_0$. Then the index i' of the k' th occurrence of symbol x in $B[j]$ can be computed in $O(1)$ time: $i' = a[j][s[j, x] + k' - 1]$, if $s[j, x] + k' < s[j, x + 1]$, where $x + 1$ is the symbol following x in the alphabet A . Otherwise, we return "no index".

Summarizing the description of this Section we obtain the following.

Theorem 4. *Let S be a string of length n and $L = 2^\ell$. S can be preprocessed into a data structure which supports answering a rank-position-query $Q^*(x, k)$*

in $O(\log(n/L))$ time. The data structure uses $n(\frac{sw|A|}{L} + \ell + \frac{\ell|A|}{L})$ bits, where w is the number of bytes in a computer word. For $|A| \leq 256$, $w = 4$, and $\ell = 12$, the size of the data structure is $14\frac{3}{4} \cdot n$ bits, and for $\ell = 13$ it is $14\frac{3}{8} \cdot n$ bits. The preprocessing needs $O(n)$ time and $O(|A| + L)$ working storage.

Remark: If we do not store the values of $p[j, x]$, but instead of this, we compute the relative index of the k' th occurrence of x for $Q^*(x, k)$ during the query time, we can obtain a sub-linear space data structure at the cost of longer query times.

4 Experimental Results

As each rank value is used exactly once in the reverse BWT, the space and runtime requirements depend solely on the size of the input, not on its content - as long as we ignore caching effects. Therefore, we give a first comparison of algorithms based on only one file. We used a binary file with 2542412 characters as input. Experiments were carried out on a 1 GHz Pentium III running Linux kernel 2.4.18 and using g++-3.4.1 for compilations. When implementing the $\frac{13}{8}n$ data structure we choose the variant with $L = 2^{13}$, where the rank values are stored in 12 bits (1.5 bytes). That allows reasonably fast access without too much bit fiddling. Our BWT-based compressor does no division into chunks for compression; the whole file is handled in one piece. The following table shows space and runtime requirements for various algorithms. The reported values refer just to the reverse BWT step, not to complete decompression. The row "4 byte rank value" contains the results for the straightforward data structure, where the rank value is maintained as a 4 byte integer for each position. The other rows show results for the algorithms discussed.

Table 1. Space and time requirement of the reverse BWT algorithms

Algorithm	Space [byte/input char]	Runtime [sec/Mbyte]
4 byte rank values	4	0.371
12 bit rank fields, 8192 fields per block	1.625	0.561
no rank fields, 1024 characters per block	1	3.477
no rank fields, 2048 characters per block	0.5	6.504

We can see that the increase in run time when we avoid fully precalculated rank-values is moderate (about 50%). Here we remark that in our implementation the reverse BWT with 4 byte rank values takes about 60% of the of the total decompression time. Thus, the increase in total decompression time is about 30%. Even without any rank-fields and one block of counts for every 1024 input characters (resulting in 256 search steps on average for each rank computation) the increase in time for the reverse BWT is less than ten fold. In an embedded system where decompressed data are written to slow flash memory, writing to flash might dominate the decompression time.

To further consolidate results, we give run times for the three methods for the files of the Calgary Corpus [15], which is a standard test suite and collection of reference results for compression algorithms (see e.g., in [5]). As runtimes were too low to be reliably measured for some of the files, each file was run ten times. Table 2 summarizes the running times.

Table 2. Runtime of the reverse BWT algorithms in [sec/Mbyte] with the files of the Calgary Corpus

File	size [bytes]	$4n$ byte data structure	$\frac{13}{8}n$ byte data structure	n byte data structure
paper5	11954	0.175	0.351	2.632
paper4	13286	0.158	0.316	2.683
obj1	21504	0.195	0.341	2.536
paper6	38105	0.196	0.358	2.642
progc	39611	0.185	0.371	2.594
paper3	46526	0.180	0.361	2.682
progp	49379	0.191	0.361	2.718
paper1	53161	0.178	0.355	2.702
progl	71646	0.205	0.381	2.795
paper2	82199	0.255	0.344	2.768
trans	93695	0.201	0.392	2.652
geo	102400	0.287	0.410	2.601
bib	111261	0.283	0.415	2.790
obj2	246814	0.259	0.442	2.885
news	377109	0.350	0.551	3.128
pic	513216	0.259	0.323	3.735
book2	610856	0.388	0.580	3.459
book1	768771	0.430	0.649	3.796

Table 2 shows that the normalized running times increase with increasing file sizes. This effect can be explained as the result of caching. Since the order of indices in consecutive rank-queries can be an arbitrary permutation of $[0, \dots, n - 1]$, the number of page faults in the L1- and L2-caches becomes higher for bigger inputs.

5 Conclusions

We showed in this paper how the memory requirement of the reverse Burrows-Wheeler transformation can be reduced without decreasing the speed too much. This transformation is used e.g. in the well known program bzip2. Decreasing the memory requirement for decompression may be essential in some embedded devices (e.g., mobile phones), where RAM is a scarce resource.

We showed that the reverse BWT can be done with $1.625n$ bytes of auxiliary memory and $O(n)$ runtime. Alternatively, we can use n/t bytes and $256tn$

operations. We also presented several time-space tradeoffs for the variants of our solution. These results are based on our new data structures for answering rank-queries and rank-position-queries. The theoretical results are backed up by experimental data showing that our algorithms work quite well in practice.

The question, if the space requirement of the data structures for rank-queries and rank-position-queries can be further reduced in our computational model, is still open. Improvements on the presented upper bounds have a practical impact. The problems of establishing lower bounds and improved time-space tradeoffs are open, as well.

References

1. J. Abel. Grundlagen des Burrows-Wheeler-Kompressionsalgorithmus (in german). *Informatik - Forschung und Entwicklung*, 2003. http://www.data-compression.info/JuergenAbel/Preprints/Preprint_Grundlagen_BWCA.pdf.
2. Z. Arnavut. Generalization of the BWT transformation and inversion ranks. In *Proc. IEEE Data Compression Conference (DCC '02)*, page 447, 2002.
3. M.J. Atallah, editor. *Algorithms and Theory of Computation Handbook*. CRC Press, 1999.
4. B. Balkenhol and S. Kurtz. Universal data compression based on the Burrows-Wheeler transformation: Theory and practice. *IEEE Trans. on Computers*, 23(10):1043–1053, 2000.
5. T.C. Bell, J.G. Cleary, and I.H. Witten. *Text Compression*. Prentice Hall, Englewood Cliffs, NJ, 1990.
6. M. Burrows and D.J. Wheeler. A block-sorting lossless data compression algorithm. *Tech. report 124, Digital Equipment Corp.*, 1994. <http://gatekeeper.research.compaq.com/pub/DEC/SRC/research-reports/abstracts/src-rr-124.html>.
7. P. Fenwick. Block sorting text compression – final report. Technical report, Department of Computer Science, The University of Auckland, 1996. <ftp://ftp.cs.auckland.ac.nz/pub/staff/peter-f/TechRep130.ps>.
8. P. Ferragina and G. Manzini. Compression boosting in optimal linear time using the Burrows-Wheeler transform. In *Proc. 15th ACM-SIAM Symposium on Discrete Algorithms (SODA '04)*, pages 655–663, 2004.
9. Tom's hardware guide. <http://www.tomshardware.com/cpu/20001120/p4-01.html>.
10. U. Manber and E. Meyers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22:935–948, 1993.
11. G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.
12. M. Nelson. Data compression with the Burrows-Wheeler transform. *Dr. Dobb's Journal*, 9, 1996.
13. J. Seward. Bzip2 manual, <http://www.bzip.org/1.0.3/bzip2-manual-1.0.3.html>.
14. J. Seward. Space-time tradeoffs in the inverse B-W transform. In *Proc. IEEE Data Compression Conference (DCC '01)*, pages 439–448, 2001.
15. I.H. Witten and T.C. Bell. The Calgary Text Compression Corpus. available via anonymous ftp at: <ftp.cpcs.ucalgary.ca/pub/projects/text.compression.corpus>.

Cache-Oblivious Comparison-Based Algorithms on Multisets

Arash Farzan¹, Paolo Ferragina², Gianni Franceschini², and J. Ian Munro¹

¹ School of Computer Science, University of Waterloo
{afarzan, imunro}@uwaterloo.ca

² Department of Computer Science, University of Pisa
{ferragin, francesc}@di.unipi.it

Abstract. We study three comparison-based problems related to multisets in the cache-oblivious model: Duplicate elimination, multisorting and finding the most frequent element (the mode). We are interested in minimizing the cache complexity (or number of cache misses) of algorithms for these problems in the context under which cache size and block size are unknown. We give algorithms with cache complexities within a constant factor of the optimal for all the problems. In the case of determining the mode, the optimal algorithm is randomized as the deterministic algorithm differs from the lower bound by a sublogarithmic factor. We can achieve optimality either with a randomized method or if given, along with the input, $\lg \lg$ of relative frequency of the mode with a constant additive error.

1 Introduction

The memory in modern computers consists of multiple levels. Traditionally, algorithms were analyzed in a flat random-access memory model (RAM) in which access times are uniform. However, the ever growing difference between access times of different levels of a memory hierarchy makes the RAM model ineffective. Hierarchical memory models have been introduced to tackle this problem. These models usually suffer from the complexity of having too many parameters which result in algorithms that are often too complicated and hardware dependant.

The *cache-aware (DAM) model* [1] is the simplest of hierarchical memory models, taking into account only two memory levels. On the first level, there is a cache memory of size M which is divided into blocks of size B ; on the second level there is an arbitrarily large memory with the same block size. A word must be present in the cache to be accessed. If it is not in the cache, we say a *cache miss/fault* has occurred, and in this case the block containing the requested word must be brought in from the main memory. In case, all blocks in the cache are occupied, a block must be chosen for eviction and replacement. Thus, a *block replacement policy* is necessary for any algorithm in the model.

The *cache-oblivious model*, which was introduced by Frigo et al. [2], is a simple hierarchical memory model and differs from the DAM model in that it avoids any hardware configuration parametrization, and in particular it is

not aware of M, B . This makes cache-oblivious algorithms independent of any hardware configuration. The block replacement policy is assumed to be the off-line optimal one. However, using a more realistic replacement policy such as the least recently used policy (LRU) increases the number of cache misses by only a factor of two if the cache size is also doubled [3]. It is known that if an algorithm in this model performs optimally on this two-level hierarchy, it will perform optimally on any level of a multiple-level hierarchy.

Cache complexity of an algorithm is the number of cache misses the algorithm causes or equivalently the number of block transfers it incurs between these two levels. In this paper, our concern is with the order of magnitude of the cache complexities of algorithms and we ignore constant factors. Following a usual practice in studying sorting problems, we make the so called *tall cache assumption*, that is, we assume the cache has size $M = \Omega(B^2)$. *Work complexity* of an algorithm is the complexity of the number of operations performed and, since here we focus on comparison-based problems, the work complexity is the number of comparisons the algorithm performs.

We will study three problems regarding the searching and sorting in multisets, namely duplicate elimination, sorting and determining the mode. A multiset is a generalization of a set in which repetition is allowed, so we can have several elements with the same key value. Suppose we are given a multiset of size N with k distinct elements whose multiplicities are N_1, \dots, N_k and set $f = \max_i N_i$. The problem of reducing the original multiset to the set of *distinct* elements is called duplicate elimination. The problem of sorting the elements of the multiset is called multisorting. The problem of finding the most frequent element (the mode) in a multiset, is called determining the mode.

Sorting can be studied in two models: In the first model, elements are such that one of the two equal elements can be removed and then later on, it can be copied back from the remained one. In this model, we can keep only one copy of an element and throw away duplicates as encountered. However, in the second model, elements cannot be deleted and regenerated as elements are viewed as complex objects with some satellite data in addition to their key values. By multisorting, then, we infer the second, more difficult problem.

Munro and Spira [4] studied these problems in the comparison model and showed tight lower and upper bounds for them. These results are summarized in Table 1. Later, Arge et al. [5] proved how the lower bounds in the comparison model can be turned into lower bounds in the cache-aware model. They used the method to obtain lower bounds for the problems of determining the mode and duplicate elimination. For the multisorting problem, we use the lower bound on duplicate elimination as duplicate elimination can be reduced to multisorting in linear complexity. The theorem in [5] also yields the same lower bound for multisorting as for duplicate elimination. A lower bound in the cache-aware model is certainly a lower bound in cache oblivious model as well. The lower bounds are summarized in Table 1. Note that there is an obvious lower bound of $\frac{N}{B}$, necessary to read the input, associated with all the problems which contributes

to the max terms in the entries of the table. From now on, in this paper, we will assume the other term is always dominant.

Arge et al.[5] give optimal cache-aware algorithms for the problems of duplicate elimination and determining the mode. Their techniques are heavily dependant on the knowledge of values of M, B . Their algorithm for duplicate elimination and determining the mode are adaptations of the mergesort and the distribution sort and at each run merges M/B sublists or breaks a lists into M/B sublists respectively. In the following sections, we give optimal *cache-oblivious* algorithms that all match the lower bounds. The optimal algorithm for determining the mode is randomized. The deterministic algorithm differs from the complexity lower bound of a sublogarithmic factor or requires “a little hint”.

Table 1. The lower bounds in the comparison model and the cache-oblivious model

	Comparisons	I/Os
Multisorting	$N \log N - \sum_{i=1}^k N_i \log N_i$	$\max \left\{ \frac{N}{B} \log \frac{M}{B} \frac{N}{B} - \sum_{i=1}^k \frac{N_i}{B} \log \frac{M}{B} N_i, \frac{N}{B} \right\}$
Duplicate Elimination	$N \log N - \sum_{i=1}^k N_i \log N_i$	$\max \left\{ \frac{N}{B} \log \frac{M}{B} \frac{N}{B} - \sum_{i=1}^k \frac{N_i}{B} \log \frac{M}{B} N_i, \frac{N}{B} \right\}$
Determining the Mode	$N \log \frac{N}{f}$	$\max \left\{ \frac{N}{B} \log \frac{M}{B} \frac{N}{f}, \frac{N}{B} \right\}$

2 Duplicate Elimination

Our duplicate removal technique is an adaptation of the lazy funnelsort [6] which sorts a set of (distinct) values. Hence, we first give a quick overview of the method in Section 2.1 and then present our modifications in Section 2.2.

2.1 Funnelsort

To sort a set of (distinct) values, Frigo et al. [2] proposed funnelsort which can be described as a cache-oblivious version of the mergesort. In funnelsort, the set of N input values are first divided into $N^{1/3}$ subsequences each of size $N^{2/3}$. Each subsequence is sorted recursively and then the sorted subsequences are merged by a data structure which is referred to as a $N^{1/3}$ -*funnel*.

A k -funnel is indeed a k -way merger that takes k sorted subsequences, merges them, and outputs the sorted sequence. A k -funnel has an output buffer of size k^3 and k input buffers. As Figure 1 illustrates, a k -funnel is built recursively out of \sqrt{k} \sqrt{k} -funnels (at the bottom) and one \sqrt{k} -funnel (on the top). There are \sqrt{k} intermediate buffers of size $2k^{3/2}$ each. They are FIFO queues that form the output buffers of the lower funnels and the input buffers of the top funnel. It can be proved by induction that a k -funnel occupies $O(k^2)$ space.

A k -funnel works recursively as follows. The k -funnel must output k^3 elements at any invocation. To do this, the top funnel is invoked many times outputting $k^{3/2}$ values at each invocation. Therefore the top funnel is eventually executed $k^{3/2}$ times to get k^3 elements. Before each invocation though, the k -buffer checks

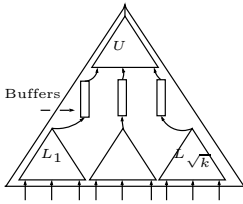


Fig. 1. ([2]) Recursive structure of a k -funnel

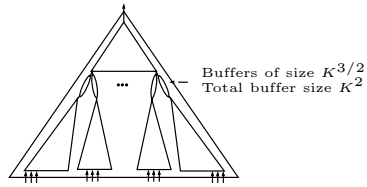


Fig. 2. ([7]) Structure of a lazy k -funnel

all its input buffers to see if they are more than half full. If any buffer is less than half full, the associated bottom funnel is invoked to fill-up its output buffer.

Later, Brodal et al.[6] simplified funnels by introducing the notion of *lazy funnelsort*. Their modification consists of relaxing the requirement that all input buffers must be checked before each invocation. Rather, a buffer is filled up on demand, when it runs empty. This simplifies the description of a funnel and as we will see in next sections, it also makes the analysis easier.

In the lazy funnelsort, a k -funnel can be thought of as a complete binary tree with k leaves in which each node is a binary merger and edges are buffers. A tree of size S is laid out recursively in memory according to the van Emde Boas layout ([8]): The tree is cut along the edges at half height. The top tree of size \sqrt{S} is recursively laid out first and it is followed by \sqrt{S} bottom trees in order that are also laid out recursively. The sizes of the buffers can be computed by the recursive structure of a funnel. As it is illustrated in Figure 2, in a k -funnel the middle edges (at half height) have size $k^{3/2}$. The sizes of the buffers in the top and bottom trees (which are all \sqrt{k} -funnels) are recursively smaller. The buffers are stored along with the trees in the recursive layout of a funnel. For the sake of consistency, we suppose there is an imaginary buffer at the top of the root node to which the root merger outputs. The size of this buffer is k^3 (this buffer is not actually stored with the funnel). The objective is to fill up this buffer. At any node, we perform merging until either of the two input buffers run empty. In such a case, the merger suspends and control is given to the associated child to fill up the buffer recursively (the child is the root of its subtree).

The cache complexity analysis of a lazy k -funnel follows an amortized argument. Consider the largest value s such that an s -funnel along with one block from each of its input buffers fits in the cache memory. It is easy to see that under the tall cache assumption $s = \Omega(M^{1/4})$ and $s = O(M^{1/2})$. The whole k -funnel can be considered as a number of s -funnels that are connected by some large buffers among them. The sizes of these buffers are at least s^3 . We denote these buffers as large buffers (see Figure 3).

An s -funnel fits entirely in cache memory and once completely loaded it does not cause any extra cache misses during its working. An s -funnel has size s^2 and thus it takes only $O(s^2/B + s)$ cache faults to load it in memory (that is if we exclude the cache misses that may occur when an s -funnel make use of its input and output buffers). Though it outputs s^3 elements, thus the amortized

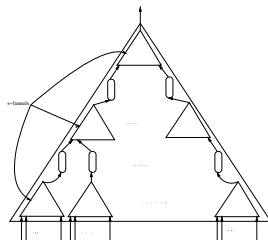


Fig. 3. The funnel can be considered as a number of s -funnels connected by buffers

cost per element that enters in the funnel is $O(1/B)$. However, in the event that its input buffers run empty, an s -funnel will have to stop and give control to its lower subfunnels. In that case, the funnel might get evicted from the memory and when its input buffers are filled up again, it must be reloaded into memory. We have to account for these cache faults as well. Every time a buffer runs empty, it is filled up with at least s^3 elements. The cost of reloading the s -funnel can be charged to these s^3 elements. In the very last invocation a funnel might not be able to output sufficiently many elements, but we can simply charge the cost to the previous invocation (there exists at least one). Each element is thus charged $O(1/B)$ in each of the large buffers. Since there are $\Theta(\log_M N)$ such buffers, the total charge is $O(\frac{N}{B} \log_M N)$ which is optimal. The work complexity can be calculated similarly to be $O(N \log N)$.

2.2 Duplicate Removal by Funnels

As it was mentioned previously, our duplicate elimination is an adaptation of the lazy funnelsort. We introduce a constraint on a binary merger: When the two elements on top of the two input buffers of a binary merger are equal, one of them is appended to a memory zone devoted to contain the duplicates. In the end we obtain the set of distinct elements with all the duplicate elements laying in a separate zone in no particular order.

We first show that the work complexity of the above mentioned algorithm for duplicate elimination is optimal, i.e. $O(N \log N - \sum_i N_i \log N_i)$. Being a binary merging algorithm, we would spend $N \log N$ comparisons on finding the total ordering of the multiset, if elements were all pairwise distinct[4]. Therefore, we need only to show that by removal of duplicates, we save $\sum_i N_i \log N_i$ number of comparisons. Consider the set E of duplicates of an element i_j ($|E| = N_j$). At any comparison of two elements in E , one of them is removed immediately, thus there can be at most N_j comparisons among elements of E . As it was shown by Munro et al.[4], this implies a saving of $N_j \log N_j - N_j$ in the number of comparisons. Hence, in total we have a saving of $\sum_i N_i \log N_i - \sum_i N_i$. So the total comparisons is $N \log N - \sum_i N_i \log N_i + \sum_i N_i = O(N \log N - \sum_i N_i \log N_i)$.

Now we show that the cache complexity of the algorithm also matches the lower bound shown in Section 1. The analysis is essentially the same as that of the cache complexity of funnelsort in Section 2.1, with the difficulty now that it

is no longer true that it does exist at least one invocation of an s -funnel, with $s = \Omega(M^{1/4})$ and $s = O(M^{1/2})$, having in input at least s^3 elements.

As depicted in Fig. 2, a k -funnel can be seen as a complete binary tree with k leaves, where every internal node is a binary merger and every edge is a buffer. The whole recursive algorithm can be then abstracted as a complete binary tree T with buffers on the edges. We know that every recursive call corresponds to a t -funnel, for some t , having t input buffers of size t^2 each (possibly containing less than t^2 elements because of duplicate removal) and an output buffer of size t^3 . Given this view, the topmost $(1/3)\log N$ levels of T correspond to the $N^{1/3}$ -funnel which is the last one executed in the algorithm. The leaves of this funnel have input buffers of size $N^{2/3}$, which are filled up by the $N^{2/9}$ -funnels corresponding to the next $(2/9)\log N$ levels of T . These funnels are executed before the top one. The other levels of T are defined recursively.

To evaluate the cost of our algorithm's recursion, we observe that when the funnels constituting T are fully confined within main memory no extra I/O occurs. These *small* funnels lie in the bottommost $l = \Theta(\log M)$ levels of T , and they have sizes between $M^{2/3}$ and M . The *larger* funnels above the small ones in T thus contain $O(N/M^{2/3})$ overall binary mergers. Since we know that an s -funnel takes $O(s + s^2/B)$ extra I/Os to perform its duplicate elimination task, every binary merger in this funnel pays $O(1 + s/B)$ extra I/Os. Summing over all the binary mergers of the larger funnels we have that the cache complexity of loading and using these s -funnels is $O(N/M^{2/3})O(1 + s/B) = O(N/B)$. This shows that the cache complexity of an element entering in a s -funnel is $O(1/B)$.

By considering the path an element takes from a leaf of T to its root, we have that the element enters a number of s -funnels which is proportional to the number of comparisons it participates in divided by $\Omega(\log M)$. Since the overall number of comparisons in the algorithm is $N \log N - \sum_i N_i \log N_i$, the number of element entrances into s -funnels is $O((1/\log M))$ of that. But, as we argued, every element entrance in an s -funnel costs $O(1/B)$ and hence:

Theorem 1. *The cache complexity of the duplicate removal algorithm matches the lower bound and is $O\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B} - \sum_{i=1}^k \frac{N_i}{B} \log_{\frac{M}{B}} N_i\right)$.*

3 Multisorting

In this section, we will show how to sort a multiset within a constant factor of the optimal number of I/Os. We will match the lower bound presented in Section 1 which is the same as the lower bound for duplicate elimination.

The algorithm consists of two phases. The first phase is duplicate elimination, the second phase is based on a “reverse” application of a funnel now oriented to *elements distribution* rather than to elements merging. The distribution of elements does not preserve the original ordering of the duplicates and thus the resulting algorithm is an unstable sort. We run the duplicate elimination algorithm to discover all the distinct elements in the multiset and to count their multiplicities as follows. We associate a counter to each element which is initialized to one in the beginning. When two duplicates are compared, one of them

is appended to a memory zone devoted to contain the duplicates and we add its counter value to the counter of the element remained. Knowing the sorted order of the distinct elements and their multiplicities, we can easily figure out the boundaries of duplicates in the sorted multiset: The sorted multiset will be composed of subsequences $A_1 A_2 \dots A_m$ where A_r consists entirely of duplicates of element i_r . By a scan over the sorted distinct elements, we find out the beginning and ending positions of each duplicate sequence A_i in the sorted multiset.

For the second phase, we need the concept of *k-splitter*. A *k-splitter* is a binary tree with k leaves. Each internal node v of a splitter is composed by an input buffer, a *pivot* $p(v)$ and a pointer to a contiguous memory portion $Z(v)$ *outside* the *k-splitter* where the duplicates of $p(v)$ will be moved as soon as they meet with the pivot. The root of the splitter, and leaves are, respectively, the input and the output buffers of the whole *k-splitter*. The buffers associated with the root and the leaves have sizes $O(k^3)$ and $O(k^2)$, respectively, and are considered external entities, just like the $Z(v)$, and they will not be accounted as space occupied by the whole *k-splitter*. As for the internal nodes, the choice of the size of their buffers, and the layout in memory of a *k-splitter*, is done as in the case of a *k-funnel* and therefore the space occupied is $O(k^2)$ (only the buffers and pivots associated with internal nodes are accounted). When an internal node v is invoked, the elements of its buffer are partitioned and sent to its two children according to the results of comparisons with $p(v)$. The duplicates of the pivot are immediately moved into $Z(v)$. The process involving v stops when its buffer is empty, or if the buffer of at least one of its children is full: in the first case the control is returned to the parent of v , in the second case the child (children) corresponding to the full buffer(s) is (are) invoked. The root is marked as “done” when there are no more elements in its buffer. Any node other than the root is marked as “done” when there are no more elements in its buffer and its parent has been marked as done. The splitters we use are not necessarily complete splitters. Thus we have to process each output buffer of a splitter recursively by other splitters (this corresponds to the top-level recursion from funnelsort).

We are ready now to sketch the second phase of our multisorting algorithm. Recall that the first phase produced two sequences, one containing the sorted set of distinct elements and the other one with the remaining duplicates in no particular order. Let us denote respectively with A and B these sequences. Moreover, for each element e we have counted its multiplicity $n(e)$. The second phase consists of three main steps.

1. With a simple scan of A a sequence $C = A_1 A_2 \dots A_m$ is produced. For any i , A_i has $n(A[i])$ positions, its first position is occupied by $A[i]$ and any other position contains a pointer to the first location of subsequence A_i . Let $D = E_1 E_2 \dots E_m$ be a contiguous memory portion divided into subarrays E_i of size $n(A[i])$. In the end, D will contain the sorted multiset. Finally, we let $F = G_1 G_2 \dots G_m$ be a contiguous memory portion divided into subarrays G_i of size $n(A[i])$.
2. Using C we construct a splitter S of height at most $(1/3) \log N$ (not necessarily a complete splitter) and we invoke it. Let A_j be the subsequence of

C that includes the $N/2$ -th location (we need $O(1)$ I/Os to find j). If $j > 1$ or $j < m$ then we have the element $A_j[1]$ as the pivot of the root of the splitter. This step is applied recursively to the sequences $C' = A_1 \dots A_{j-1}$ and $C'' = A_{j+1} \dots A_m$ (at least one of them exists). That recursive process stops when the maximal tree S of pivots of height $(1/3) \log N$ is built (not necessarily a complete binary tree) or when there are no more sequences A_j . Note that each internal node v of S is associated to a subsequence A_j (the one from which $p(v)$ has been taken). Let t be the number of internal nodes of S , let $j_1, j_2 \dots j_t$ be the indices (in increasing order) of the sub-sequences A_j of C chosen in the process above, and let v_{j_r} be the internal node of S that has $A_{j_r}[1]$ as pivot. The memory zone $Z(v_{j_r})$ devoted to contain the duplicates of v_{j_r} , is E_{j_r} . The input buffer of the root of S is B (the one that contains the duplicates after the first phase). The buffer of the leaf u that is the right (resp. left) child of a node v_{j_r} of S is $B_u = G_{j_r+1}G_{j_r+2} \dots G_{j_r+1-1}$ (resp. $B_u = G_{j_r-1+1}G_{j_r-1+2} \dots G_{j_r-1}$). Finally, the sizes of the buffers of all the internal nodes and their layout in memory are chosen as we already discussed above when we gave the definition of k -splitter. Now, the splitter is ready and can be invoked.

3. Recursive steps. For any output buffer of the splitter S we apply Step 2 recursively. Let u be a leaf of S and let us suppose that is the right child of a node v_{j_r} of S (the left-child case is analogous). In the recursive call for u we have that $B_u = G_{j_r+1}G_{j_r+2} \dots G_{j_r+1-1}$ plays the role of B , $E_{j_r+1}E_{j_r+2} \dots E_{j_r+1-1}$ plays the role of D , and $A_{j_r+1}A_{j_r+2} \dots A_{j_r+1-1}$ plays the role of C . After the recursive steps, D contains the sorted multiset.

The analysis is similar to the one for the duplicate elimination problem, and it can be shown that we get the same optimal I/O-bound. We omit the details of the proof here. Though we mention the observations necessary to obtain the complexity. The first step is a simple scan and requires $O(N/B)$ I/Os. In the second step, the splitter S can be divided in s -splitters pretty analogously to the case of s -mergers. The third step, as mentioned earlier, corresponds to the top-level recursion of funnelsort by Frigo et al.[2]. Similar to a funnel, a recursive call operates in a contiguous region of memory and all accesses are confined to this region. As soon as the size of this region drops below M , the region is entirely present in the cache memory and we incur no more cache faults thereon. It can also be shown that all the “spurious” I/O’s— like the $O(1)$ accesses for the construction of any node in the splitter or the $O(s + s^2/B)$ accesses needed to load a s -splitter— give again $O(N/B)$ I/Os.

Theorem 2. *The cache complexity of the multisorting algorithm matches the lower bound and is $O\left(\frac{N}{B} \log \frac{M}{B} \frac{N}{B} - \sum_{i=1}^k \frac{N_i}{B} \log \frac{M}{B} N_i\right)$.*

4 Determining the Mode

In this section, we study the problem of determining the most occurring element (mode) in a multiset. The cache-aware algorithm [5] for the problem is funda-

mentally dependant on knowledge of values of M, B and seems impossible to make cache-oblivious. We will take two approaches: Deterministic and Randomized. The upper bound we achieve in the randomized approach matches the lower bound for finding the mode as was mentioned in Section 1, essentially because we can use samples to get a “good enough” estimate of the *relative frequency* of the mode (i.e. f/N). However, the deterministic approach can be in the worst case an additive term of $O\left(\frac{N}{B} \log \log M\right)$ away from the lower bound.

4.1 Deterministic Approach

The key idea is to use as a basic block a cache-efficient algorithm for finding “frequent” elements that occur more often than a certain threshold in the multiset. We then repeatedly run the algorithm for a *spectrum of thresholds* to hunt the most frequent element. Let us first precisely define what we mean by a “frequent” element.

Definition 1. We call an element C -frequent if and only if it occurs more than $\frac{N}{C}$ times in a multiset of size N (i.e. if its relative frequency is at least $1/C$).

The algorithm works in two phases. In the first phase, we try to find a set of at most C candidates that contains all the C -frequent elements. There may also be some other arbitrarily infrequent elements in our list of candidates. In the second phase, we check the C candidates to determine their exact frequencies. Note that, by definition, the number of C -frequent elements cannot exceed C .

Phase 1. The key idea in this phase is essentially what Misra [9] used. We find and remove a set of t ($t \geq C$) distinct elements from the multiset. The resulting multiset has the property that those elements that were C -frequent in the original multiset are still C -frequent in the reduced multiset. Thus, we keep removing sets of at least C distinct elements from the multiset, one at a time, until the multiset has no longer more than C distinct elements. The C -frequent elements in the original multiset must be also present in the final multiset.

We scan the multiset in groups of C elements (there are N/C such groups) by maintaining an array of C “candidates” which is initially empty and eventually will hold as many as C distinct values. Each element in this array also has a counter associated with it which shows the number of occurrences of the element seen so far. As soon as the number of elements in this array goes over C , we downsize the array by removing C distinct elements. More precisely, for each group G of C elements from the multiset, we first sort the elements in G and also sort the elements in the candidates array. This can be done by using any method of cache-oblivious sorting by pretending that the elements are all distinct. Then, we merge the two sorted arrays into another array T ensuring that we keep only the distinct elements. T may contain up to $2C$ elements. At this time we remove groups of at least C distinct elements to downsize T to at most C elements. This is done by sorting T according to the value of the counters (not the value of elements), and by finding the $C + 1$ largest counter m_{C+1} . All elements with a counter value less than m_{C+1} are thrown away, and the counter of the other

elements is decreased by m_{C+1} . One can easily see that this is equivalent to repeatedly throwing away groups of at least C distinct elements from T one at a time. The candidates array is then set to T , and the process continues on the next group of C elements from the multiset. At the end, the candidates array contains all possible C -frequent elements in the multiset; however, as mentioned, it may also contain some other arbitrary elements.

Phase 2. This phase is similar to the first phase, except that the candidates array remains intact throughout this phase. We first zero out all the counters and sort the array using any method of cache-oblivious sorting for sets. We then consider N/C groups of C elements from the multiset one at a time; we first sort the C elements for each group, and by doing a scan of the lists as in the merge sort, we can count how many times each of the candidates occur in the group. We accumulate these counts so that after considering the final group, we know the multiplicities of the candidates in the whole multiset. We finally keep all elements whose counters are more than N/C and discard the rest. Cache complexity of both phases are the same and one can see the major task is N/C executions of sorting C elements. Therefore:

Lemma 1. *In a multiset of size N , C -frequent elements and their actual multiplicities can be determined with cache complexity $O\left(\frac{N}{B} \max\{1, \log_{\frac{M}{B}} C\}\right)$.*

Now we show how the frequent finding algorithm can be used in our hunt for the mode. We repeatedly apply Lemma 1 for a series of increasing values of C to determine whether there is any C -frequent element in the multiset. The first time some C -frequent elements are found, we halt the algorithm and declare the most frequent among them as the mode. The algorithm in Lemma 1 is executed in rounds as C goes doubly exponentially for the following values of C in order: $C = 2^{2^1}, 2^{2^2}, \dots, 2^{2^i}, \dots, 2^{2^{\lceil \lg \lg n \rceil}}$. At the end of each round, we either end up empty-handed or we find some C -frequent elements. In the former case, the algorithm continues with the next value of C . In the latter case, we declare the most frequent of the C -frequent elements to be the mode, and the algorithm halts. Note that the algorithm of Lemma 1 also produces the actual multiplicities of the C -frequent elements, thus finding the most frequent element among the C -frequent ones requires only a pass of the at most C elements to select the element with the maximum multiplicity.

The cache complexity of the algorithm can be analyzed as follows. Let us denote by f the frequency of the mode. The cache complexity of the algorithm is the sum of cache complexity of the algorithm in Lemma 1 over different values of C up to $2^{2^{\lg(N/f)+1}}$, where we find the mode. Hence, the cache complexity is

$$\sum_{j=1}^{\lg \lg(N/f)+1} O\left(\frac{N}{B} \max\left\{1, \log_{\frac{M}{B}} 2^{2^j}\right\}\right) = O\left(\max\left\{\frac{N}{B} \log \log \frac{N}{f}, \frac{N}{B} \log_{\frac{M}{B}} \frac{N}{f}\right\}\right).$$

It is clear to see that the following cache oblivious upper bound can be in the worst case an additive term of $O\left(\frac{N}{B} \log \log M\right)$ larger than the lower bound:

Theorem 3. *The cache complexity of the deterministic algorithm for determining the mode is $O\left(\max\left\{\frac{N}{B} \log \frac{M}{B}, \frac{N}{f}, \frac{N}{B} \log \log \frac{N}{f}\right\}\right)$. \square*

It is worthy to note that the slightest hint on the size of memory or the relative frequency of the mode would result in an optimal algorithm. Given the value of M , we can tailor the program so it skips from values of C smaller than M and starts from $C = M$. Thus, as a by-product, we have an optimal *cache-aware algorithm* which is simpler than the existing one in Arge et al. [5]. Also, knowing the value of $\lg \lg \frac{N}{f}$ with a constant additive error helps us to jump start from the right value for C . Furthermore, given an element, we can confirm with an optimal cache complexity whether it is indeed the mode. Let us define *having a hint* on a value v as knowing $\lg \lg v$ within a constant additive error:

Theorem 4. *Given a hint on the value of memory size or relative frequency of the mode (i.e. M or N/f), the cache complexity of the deterministic algorithm for determining the mode matches the lower bound and is $O\left(\frac{N}{B} \log \frac{M}{B}, \frac{N}{f}\right)$.*

4.2 Randomized Approach

We still use the C -frequent finder algorithm, but instead of starting from small values of C and squaring it at each step, we will estimate a good value for C using randomized sampling techniques. The sample must be large enough to produce a good estimate, with high confidence. It must also be small enough so that working with the sample does not dominate our cost. We have a high degree of latitude in choosing the sample size. Something around $N^{1/3}$ is reasonable.

Sample generation takes $N^{1/3}$ random I/Os which is less than N/B . We can also afford to sort these sampled elements as sorting $N^{1/3}$ elements takes $O\left(\frac{N}{B}\right)$ cache misses. After sorting the sample, we scan and find the mode in the sample with frequency p . The estimate of the frequency of the mode in the multiset is $f' = pN^{2/3}$. Consequently we start by finding C -frequent elements for $C = \frac{N}{f'}$. If there is no C -frequent element, we square C and re-run the algorithm for the new value of C and so on.

Let us now sketch the analysis the cache complexity of the randomized algorithm. Clearly, if our estimate for the mode is precise, then the cache complexity of the algorithm matches the lower bound. However undershoot or overshoot are possible: We may underestimate the value of f (i.e. $f' < f$), or we may overestimate the value of f (i.e. $f' > f$). In the former case, we find the mode on the first run of the frequent element finder algorithm, but as the value of C is greater than what it should be, the cache complexity of the algorithm can be potentially larger. In the latter case, multiple runs of the frequent finder algorithm is likely; Therefore, potentially we can have the problem of too many runs as in the deterministic approach. Details are omitted in this paper, nevertheless using Chernoff tail bounds, one can show that the probability of our estimate being too far from the real value is small enough so that the extra work does not effect the expected asymptotic cache complexity:

Theorem 5. *The expected cache complexity of the randomized algorithm for determining the mode matches the lower bound and is $O\left(\max\left\{\frac{N}{B} \log \frac{M}{B}, \frac{N}{fB}, \frac{N}{B}\right\}\right)$.*

5 Conclusion

We studied three problems related to multisets in the cache-oblivious model: duplicate removal, multi-sorting, and determining the mode. We presented the known lower bounds for the cache complexity of each of these problems. Determining the mode has the lower bound of $\Omega\left(\frac{N}{B} \log \frac{M}{B} \frac{N}{f}\right)$ where f is the multiplicity of the most frequent element and M is the size of the cache and B is size of a block in cache. The lower bound for the cache complexity of duplicate removal and multi-sorting is $\Omega\left(\frac{N}{B} \log \frac{M}{B} \frac{N}{B} - \sum_{i=1}^k \frac{N_i}{B} \log \frac{M}{B} \frac{N_i}{B}\right)$.

The cache complexities of our algorithms match the lower bounds asymptotically. Only exception is the problem of determining the mode where our deterministic algorithm can be an additive term of $O\left(\frac{N}{B} \log \log M\right)$ away from the lower bound. However, the randomized algorithm matches the lower bound.

References

1. Aggarwal, A., Vitter, J.S.: The I/O complexity of sorting and related problems. In: ICALP Proceedings. Volume 267 of LNCS., Springer-Verlag (1987) 467–478
2. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. In: FOCS Proceedings, IEEE Computer Society Press (1999) 285–297
3. Sleator, D.D., Tarjan, R.E.: Amortized efficiency of list update and paging rules. *Commun. ACM* **28(2)** (1985) 202–208
4. Munro, I., Spira, P.: Sorting and searching in multisets. *SIAM Journal on Computing* **5** (1976) 1–8
5. Arge, L., Knudsen, M., Larsen, K.: A general lower bound on the I/O-complexity of comparison-based algorithms. In: In Proceedings of WADS, Springer-Verlag (1993)
6. Brodal, Fagerberg: Cache oblivious distribution sweeping. In: ICALP. (2002)
7. Demaine, E.D.: Cache-oblivious algorithms and data structures. In: Lecture Notes from the EEF Summer School on Massive Data Sets. Lecture Notes in Computer Science, BRICS, University of Aarhus, Denmark (2002)
8. Bender, M.A., Demaine, E.D., Farach-Colton, M.: Cache-oblivious B -trees. In IEEE, ed.: Annual Symposium on Foundations of Computer Science 2000, IEEE Computer Society Press (2000) 399–409
9. Misra, J., Gries, D.: Finding repeated elements. *Science of Computer Programming* **2** (1982) 143–152

Oblivious vs. Distribution-Based Sorting: An Experimental Evaluation

Geeta Chaudhry* and Thomas H. Cormen**

Dartmouth College Department of Computer Science
{geetac, thc}@cs.dartmouth.edu

Abstract. We compare two algorithms for sorting out-of-core data on a distributed-memory cluster. One algorithm, *Csort*, is a 3-pass oblivious algorithm. The other, *Dsort*, makes two passes over the data and is based on the paradigm of distribution-based algorithms. In the context of out-of-core sorting, this study is the first comparison between the paradigms of distribution-based and oblivious algorithms. *Dsort* avoids two of the four steps of a typical distribution-based algorithm by making simplifying assumptions about the distribution of the input keys. *Csort* makes no assumptions about the keys. Despite the simplifying assumptions, the I/O and communication patterns of *Dsort* depend heavily on the exact sequence of input keys. *Csort*, on the other hand, takes advantage of predetermined I/O and communication patterns, governed entirely by the input size, in order to overlap computation, communication, and I/O. Experimental evidence shows that, even on inputs that followed *Dsort*'s simplifying assumptions, *Csort* fared well. The running time of *Dsort* showed great variation across five input cases, whereas *Csort* sorted all of them in approximately the same amount of time. In fact, *Dsort* ran significantly faster than *Csort* in just one out of the five input cases: the one that was the most unrealistically skewed in favor of *Dsort*. A more robust implementation of *Dsort*—one without the simplifying assumptions—would run even slower.

1 Introduction

This paper demonstrates the merit of oblivious algorithms for out-of-core sorting on distributed-memory clusters. In particular, we compare the performance of *Csort*, a 3-pass oblivious algorithm that makes no assumptions about the input distribution, to that of *Dsort*, a 2-pass distribution-based algorithm that makes strong simplifying assumptions about the input distribution. This difference makes *Csort* a more robust algorithm for sorting real out-of-core data. Because *Csort* is oblivious, its I/O and communication patterns are not affected

* Supported by National Science Foundation Grant IIS-0326155 in collaboration with the University of Connecticut.

** Supported in part by DARPA Award W0133940 in collaboration with IBM and in part by National Science Foundation Grant IIS-0326155 in collaboration with the University of Connecticut.

by the input distribution or the exact input sequence. In Dsort, on the other hand, the I/O and communication patterns show a strong sensitivity to the input sequence, even when it adheres to the assumed input distribution. An added benefit of Csort’s predetermined I/O and communication patterns is that it is much simpler to implement. We ran experiments with five 128-GB datasets, ranging from the more favorably skewed (for Dsort) to the less favorably skewed. There was no significant variation in the running time of Csort for these inputs, demonstrating that its running time is determined primarily by the input size. Dsort ran no faster than Csort, except in the two most-favorably biased cases; the difference was marginal in one of these two cases. One downside of using Csort is that the maximum problem size that it can handle is often smaller than what Dsort can handle.

The problem of sorting massive data comes up in several applications such as geographical information systems, seismic modeling, and Web-search engines. Such *out-of-core* data typically reside on parallel disks. We consider the setting of a distributed-memory cluster, since it offers a good price-to-performance ratio and scalability. The high cost of transferring data between disk and memory, as well as the distribution of data across disks of several machines, makes it quite a challenge to design and implement efficient out-of-core sorting programs. In addition to minimizing the number of parallel disk accesses, an efficient implementation must also overlap disk I/O, communication, and computation.

Along with merging-based algorithms, distribution-based algorithms form one of the two dominant paradigms in the literature for out-of-core sorting on distributed-memory clusters. An algorithm of this paradigm proceeds in three or four steps. The first step samples the input data and decides on $P - 1$ splitter elements, where P is the number of processors in the cluster. The second step, given the splitter elements, partitions the input into P sets S_0, S_1, \dots, S_{P-1} , such that each element in S_i is less than or equal to all the elements in S_{i+1} . The third step sorts each partition. The sorted output is just the sequence $\langle R_0, R_1, \dots, R_{P-1} \rangle$, where R_i is the sorted version of S_i . We refer to the first, second, and third steps as the *sampling* step, the *partition* step, and the *sort* step, respectively. There is often a fourth step to ensure that the output is perfectly load-balanced among the P processors. In order to give Dsort every possible advantage when comparing it to Csort, its implementation omits the sampling and load-balancing steps.

In previous work, we have explored a third way—distinct from merging-based and distribution-based algorithms—of out-of-core sorting: oblivious algorithms. An *oblivious* sorting algorithm is a compare-exchange algorithm in which the sequence of comparisons is predetermined [1,2]. For example, algorithms based on sorting networks [1,3] are oblivious algorithms.

A distribution-based algorithm, when adapted to the out-of-core setting of a distributed-memory cluster, generates I/O and communication patterns that vary depending on the data to be sorted. Due to this input dependence, the programmer ends up spending much of the effort in handling the effects of data that might lead to “bad” I/O and communication patterns. We know of two

implementations of out-of-core sorting on a cluster [4,5]. Both are distribution-based, and both sidestep one of the principal challenges of the sampling step of a distribution-based sort: locating the k th smallest of the input keys for k equal to $N/P, 2N/P, \dots, (P-1)N/P$ (a classical problem in order statistics). We refer to these k elements as *equi-partition splitters*. Solving this problem in an out-of-core setting is cumbersome [6]. Not surprisingly, therefore, both of the existing implementations assume that the $P-1$ equi-partition splitters are known in advance, eliminating the sampling and load-balancing steps altogether.

An oblivious algorithm, when adapted to an out-of-core setting, generates I/O and communication patterns that are entirely predetermined, depending only on the input size. In previous work, we have developed several implementations based on the paradigm of oblivious algorithms [7,8,9,10].

The comparison between Csort and Dsort is novel. To the best of our knowledge, this work is the first experimental evaluation that compares these two paradigms in the context of out-of-core sorting on distributed-memory clusters.¹ Moreover, both Csort and Dsort run on identical hardware, and both use similar software: MPI [12,13] for communication and UNIX file system calls for I/O. Both are implemented in C and use the standard pthreads package for overlapping I/O, communication, and computation.

There are several reasons that we did not compare Csort to NOW-Sort [4,14], the premier existing implementation of a distribution-based algorithm:

- NOW-Sort is built on top of active messages and GLUnix, and we wanted to use software that is more standard. (Our nodes run Red Hat Linux and use MPI for communication.)
- There are several differences in the hardware that NOW-Sort targets and the hardware of modern distributed-memory clusters.
- Finally, NOW-Sort does not produce output in the standard striped ordering used by the Parallel Disk Model (PDM) [15]. As Figure 1 shows, the PDM stripes N records² across D disks D_0, D_1, \dots, D_{D-1} , with N/D records stored on each disk. The records on each disk are partitioned into *blocks* of B records each. Any disk access (read or write) transfers an entire block of records between the disks and memory. We use M to denote the size of the internal memory, in records, of the entire cluster, so that each processor can hold M/P records.

PDM ordering balances the load for any consecutive set of records across processors and disks as evenly as possible. A further advantage to producing sorted output in PDM ordering is that the resulting algorithm can be used as a subroutine in other PDM algorithms. Implementing our own version of a distribution-based sort removes all these differences, allowing for a fairer comparison.

¹ There are some existing comparisons of parallel sorting programs for an in-core setting, e.g., [11].

² The data being sorted comprises N records, where each record consists of a key and possibly some satellite data.

	P_0				P_1				P_2				P_3			
	D_0		D_1		D_2		D_3		D_4		D_5		D_6		D_7	
stripe 0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
stripe 1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
stripe 2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
stripe 3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

Fig. 1. The layout of $N = 64$ records in a parallel disk system with $P = 4$, $B = 2$, and $D = 8$. Each box represents one block. The number of stripes is $N/BD = 4$. Numbers indicate record indices.

Experimental results on a Beowulf cluster show that Csort, even with its three disk-I/O passes, sorts a 128-GB file significantly faster than Dsort in two of the five cases. There is a marginal difference in the running times in two of the remaining three cases, one in the favor of Dsort and the other in Csort’s favor. In the last case, Dsort is distinctly faster than Csort. These results show that the performance of Csort is competitive with Dsort. Csort, therefore, would fare even better compared to a generalized version of Dsort, that is, one that does not assume that the equi-partition splitters are known in advance and that does not ensure load-balanced output.

The remainder of this paper is organized as follows. Section 2 describes the original column sort algorithm and summarizes Csort. Section 3 presents the design of Dsort, along with notes on the 2-pass implementation. Section 4 analyzes the results of our experiments. Finally, Section 5 offers some closing comments.

2 Csort

In this section, we briefly review the column sort algorithm [16]. After summarizing a 4-pass out-of-core adaptation, we briefly describe Csort, our 3-pass implementation. Our previous papers [7,8] contain the details of these implementations.

Column sort sorts N records arranged as an $r \times s$ matrix, where $N = rs$, r is even, s divides r , and $r \geq 2s^2$. When column sort completes, the matrix is sorted in column-major order. Column sort proceeds in eight steps. Steps 1, 3, 5, and 7 are all the same: sort each column individually. Each of steps 2, 4, 6, and 8 performs a fixed permutation on the matrix entries:

- *Step 2: Transpose and reshape:* Transpose the $r \times s$ matrix into an $s \times r$ matrix. Then “reshape” it back into an $r \times s$ matrix by interpreting each row as r/s consecutive rows of s entries each.
- *Step 4: Reshape and transpose:* This permutation is the inverse of that of step 2.
- *Step 6: Shift down by $r/2$:* Shift each column down by $r/2$ positions, wrapping the bottom half of each column into the top half of the next column. Fill the top half of the leftmost column with $-\infty$ keys, and create a new rightmost column, filling its bottom half with ∞ keys.
- *Step 8: Shift up by $r/2$:* This permutation is the inverse of that of step 6.

A 4-Pass Implementation. In our adaptation of columnsort to an out-of-core setting on a distributed-memory cluster, we assume that D , the number of disks, equals P , the number of processors.³ We say that a processor *owns* the one disk that it accesses. The data are placed so that each column is stored in contiguous locations on the disk owned by a single processor. Columns are distributed among the processors in round-robin order, so that P_j , the j th processor, *owns* columns $j, j + P, j + 2P$, and so on.

Throughout this paper, we use buffers that hold exactly β records. For out-of-core columnsort, we set $\beta = r$. We assume that each processor has enough memory to hold a constant number, g , of such β -record buffers. In other words, $M/P = g\beta = gr$, implying that $\beta = r = M/Pg$. In both Csort and Dsort, each processor maintains a global pool of g memory buffers, where g is set at the start of each run of the program.

Each pass reads records from one part of each disk and writes records to a different part of each disk.⁴ Each pass performs two consecutive steps of column-sort. That is, pass 1 performs steps 1 and 2, pass 2 performs steps 3 and 4, pass 3 performs steps 5 and 6, and pass 4 performs steps 7 and 8. Each pass is decomposed into s/P rounds. Each round processes the next set of P consecutive columns, one column per processor, through a pipeline of five stages. This pipeline runs on each processor. In each round on each processor, an r -record buffer travels through the following five stages:

Read Stage: Each processor reads a column of r records from the disks that it owns into the buffer associated with the given round.

Sort Stage: Each processor locally sorts, in memory, the r records it has just read.

Communicate Stage: Each record is destined for a specific column, depending on which even-numbered column-sort step this pass is performing. In order to get each record to the processor that owns this destination column, processors exchange records.

Permute Stage: Having received records from other processors, each processor rearranges them into the correct order for writing.

Write Stage: Each processor writes a set of r records onto the disks that it owns.

Because we implemented the stages asynchronously, at any one time each stage could be working on a buffer from a different round. We used threads in order to provide flexibility in overlapping I/O, computation, and communication. In the 4-pass implementation, there were four threads per processor. The sort, communicate, and permute stages each had their own threads, and the read and

³ Our implementation of Csort can handle any positive value of D as long as D divides P or P divides D ; we assume that $D = P$ in this paper. In our experimental setup, each node has a single disk, so that $D = P$.

⁴ We alternate the portions read and written from pass to pass so that, apart from the input and output portions, we need just one other portion, whose size is that of the data.

write stages shared an I/O thread. The threads operate on r -record buffers, and they communicate with one another via a standard semaphore mechanism.

Csort: The 3-Pass Implementation. In Csort, we combine steps 5–8 of column sort—passes 3 and 4 in the 4-pass implementation—into one pass. In the 4-pass implementation, the communicate, permute, and write stages of pass 3, along with the read stage of pass 4, merely shift each column down by $r/2$ rows (wrapping the bottom half of each column into the top half of the next column). We replace these four stages by a single communicate stage. This reduction in number of passes has both algorithmic and engineering aspects; for details, see [7,8].

3 Dsort

In this section, we outline the two passes of Dsort: Pass 1 executes the partition step, and pass 2 executes the sort step, producing output in PDM order. We continue to denote the number of records to be sorted as N , the number of processors as P , and the amount of memory per processor as M/P . As in Csort, g is the number of buffers in the global pool of buffers on each processor and $\beta = M/Pg$ denotes the size of each buffer, in records.

3.1 Pass 1 of Dsort: Partition and Create Sorted Runs

Pass 1 partitions the N input records into P *partitions* of N/P records each. After pass 1 completes, each processor has the N/P records of its partition. Similar to NOW-Sort, Dsort requires that the $P - 1$ equi-partition splitters are known in advance.

The Five Stages of Each Round. Similar to the passes of Csort, pass 1 of Dsort is decomposed into rounds. Each round processes the next set of P buffers, one β -record buffer per processor, through a pipeline of five stages. In our implementation, each of the five stages has its own thread. Below, we describe what each processor (P_i , for $i = 0, 1, \dots, P - 1$) does in the five stages.

Read Stage: Processor P_i reads β records from the disk that it owns into a buffer. Hence, a total of βP records are read into the collective memory of the cluster.

Permute Stage: Given the splitter elements, processor P_i permutes the just-read β records into P sets, $U_{i,0}, U_{i,1}, \dots, U_{i,P-1}$, where the records in set $U_{i,j}$ are destined for processor P_j . Note that, even though the entire input is equi-partitioned, these P sets need not be of the same size.

Communicate Stage: Processor P_i sends all the elements in set $U_{i,j}$ to processor P_j , for $j = 0, 1, \dots, P - 1$. At the end of this stage, each processor has received all the records that belong to its partition, out of the total of βP records that were read in collectively.

The communicate stage is implemented in three substages:

- In the first substage, processor P_i sends the size of set $U_{i,j}$ to each processor P_j , so that P_j knows how many records it will receive from P_i .
- In the second substage, each processor sends P messages, one per processor. The message destined for processor P_j contains all the records in set $U_{i,j}$.
- After the second substage, a processor might receive significantly more records than it can hold in its internal memory. The third substage, therefore, proceeds in a loop over the following three steps: allocate a buffer, keep receiving messages until the buffer fills up, and send the buffer on to the next stage. The performance of this stage depends on the input data, even with the assumption that the overall input is equi-partitioned. Since a processor may fill a non-integral number of buffers, it might be left with a partially full buffer at the end of this final substage. This partially full buffer is then used as the starting buffer (for the third substage of the communicate stage) of the next round. The only buffers that go on to the next stage are either full or empty.

Sort Stage: For each full buffer that reaches this stage, processor P_i sorts the elements in the buffer. All buffers, whether full or empty, are passed on to the next stage.

Write Stage: Every buffer reaching this stage is either full and sorted, or empty. Processor P_i writes each full buffer reaching this stage to the disk that it owns. This stage then recycles one buffer back to the read stage, and it releases all others back into memory.

At the end of pass 1, each processor has the N/P records that belong to its partition. Furthermore, these N/P records are stored on the disk of the corresponding processor as $N/\beta P$ sorted runs of β records each.

3.2 Pass 2 of Dsort: Merge Sorted Runs and Create Striped Output

Pass 2 of Dsort executes the sort stage of distribution-based sorting. Each processor uses an $(N/\beta P)$ -way merge sort to merge the $N/\beta P$ runs of β records each, and it communicates with the other processors to produce output in striped PDM order. Our implementation of pass 2 has four threads: read, merge, communicate, and write. Unlike pass 1, in which each thread received buffers from the thread of the preceding stage and sent buffers to the thread of the succeeding stage, the threads of pass 2 are not as sequential in nature. We describe how each thread operates.

Read Thread: Each processor starts out with $N/\beta P$ sorted runs on its disk. Each run has β records, or β/B blocks, assuming that B , the block size, divides β , the buffer size.

- Wait for a request of the form (r_x, h_y, loc) , meaning that the y th block h_y of the x th sorted run r_x is to be read into the memory location loc .
- Read in the y th block of the x th run, and copy the B records to the specified memory location loc .
- Signal the merge thread.

Since the $N/\beta P$ runs are being merged using an $(N/\beta P)$ -way merge sort, the order in which the various blocks of the sorted runs are required depends on the rate at which each run is being consumed by the merge sort. For example, it is possible that on a given processor, the first run starts with a record whose key value is greater than all the key values in the second run. In this scenario, all blocks of the second run are brought into memory before the first run has exhausted even a single record. Feedback from the merge thread, therefore, is essential for the read thread to bring in blocks in an efficient sequence.

Merge Thread: The merge thread initially requests some fixed number, say l , of blocks from each of the $N/\beta P$ sorted runs. This number l depends on the amount of memory available on each processor. It is essential to always have more than one block per run in memory, so that the merging process is not repeatedly suspended because of delays in disk reads. The main task of the merge thread is to create one single sorted run out of the $N/\beta P$ sorted runs produced after pass 1. It starts out by acquiring an output buffer, say $buff_0$, and starting an $(N/\beta P)$ -way merge that puts the output into $buff_0$. This merging procedure continuously checks for the following two conditions:

- The output buffer $buff_0$ is full. In this case, the merge thread sends the buffer to the communicate thread and acquires another buffer from the pool of buffers.
- The current block, say h_y , of records from some run r_x has been exhausted. The merge thread sends a request to the read thread to get the next block of run r_x . Note that, since we always keep l blocks from each run in memory, the block that is requested is h_{x+l} . After issuing a request for block h_{x+l} , the merge thread waits to make sure that block h_{x+1} , the next block of r_y needed to continue the merging, is in memory.

Communicate Thread: For each buffer that this thread receives, it spreads out the contents to all other processors, for striped PDM output. In other words, blocks h_0, h_P, h_{2P}, \dots are sent to processor P_0 , blocks $h_1, h_{P+1}, h_{2P+1}, \dots$ are sent to processor P_1 , and so on, until processor P_{P-1} , which receives blocks $h_{P-1}, h_{2P-1}, h_{3P-1}, \dots$. After the communicate thread receives one buffer's worth of data from all the other processors, it sends the buffer to the write thread.

Write Thread: The write thread writes out the β records in the buffer out to the disk owned by the processor and releases the buffer back into the pool of buffers.

4 Experimental Results

This section presents the results of our experiments on *Jefferson*, a Beowulf cluster that belongs to the Computer Science Department at Dartmouth. We start with a brief description of our experimental setup. Next, we explain the five types of input sequences on which we ran both Csort and Dsort. Finally, we present an analysis of our experimental runs.

4.1 Experimental Setup

Jefferson is a Beowulf cluster of 32 dual 2.8-GHz Intel Xeon nodes. Each node has 4 GB of RAM and an Ultra-320 36-GB hard drive. A high-speed Myrinet connects the network. At the time of our experiments, each node ran Redhat Linux 8.0. We use the C `stdio` interface for disk I/O, the `pthread`s package of Linux, and standard synchronous MPI calls within threads. We use the ChaMPIon/Pro package for MPI calls.

In all our experiments, the input size is 128 GB, the number of processors is 16, the block size is 256 KB; the record size is 64 bytes; and g , the size of the global pool of buffers, is 3. Since the record size is 64 bytes, N is 2^{31} records and B is 2^{12} records. For all cases but one, we use 128-MB buffers (i.e., $\beta = 2^{21}$ records). For Dsort, the memory requirement of the worst-case input type was such that we had to use 64-MB buffers; the experiment crashed if we used 128-MB buffers. In the case of Dsort, we set the parameter l of pass 2 to be 3.⁵

4.2 Input Generation

As we mentioned before, even with equi-partition splitters known in advance, the I/O and communication patterns of Dsort depend heavily on the exact input sequence. More specifically, as explained in Section 3.1, the performance of each round of pass 1 of Dsort depends on how evenly the βP records of that round are split among the P processors. In the best case, in each round, each processor would receive exactly β records. In the worst case, all βP records in each round would belong to the partition of a single processor.

In each of our five input types, the following is true: In each round, all βP keys are such that q out of the P processors receive $\beta P/q$ records each. In each round, therefore, $P - q$ processors receive no records. The five types of inputs are characterized by the following five values of q : 1, 2, 4, 8, and 16. For each round of any input type, the q processors that receive the βP records are chosen randomly, subject to the constraint that over all $N/\beta P$ rounds, each processor receives N/P records. The smaller the value of q , the worse the load balance of each round and the longer Dsort takes to sort the input. Note that $q = P$ represents a highly unrealistic input sequence, one where the data are perfectly load balanced across the P processors in every round, in addition to the overall input being perfectly balanced across the processors. Even values of q strictly less than P represent unrealistic scenarios in which the data destined for the q processors are perfectly load balanced across those q processors every time. Thus, our experiments are, if anything, tilted in favor of Dsort.

4.3 Results

Figure 2 shows the running times of Dsort and Csort for 128 GB of input data. Each plotted point in the figure represents the average of three runs. Variations in running times were relatively small (within 5%). The horizontal axis is organized

⁵ The parameters g and l were set experimentally, to elicit the best performance.

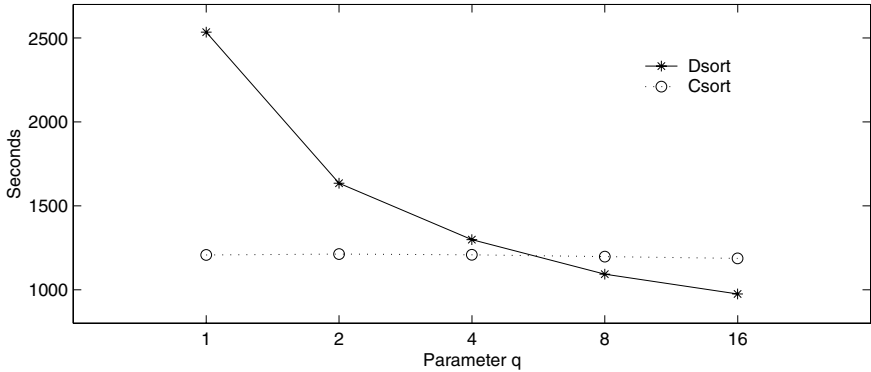


Fig. 2. Observed running times of Dsort and Csort for the five values of q : $q = 16$ is the most favorable case for Dsort, and $q = 1$ is the least favorable. These timings are for the following setting of parameters: $P = 16$, $N = 2^{31}$ records or 128 GB, and $\beta = 2^{21}$ records or 128 MB, for 64-byte records.

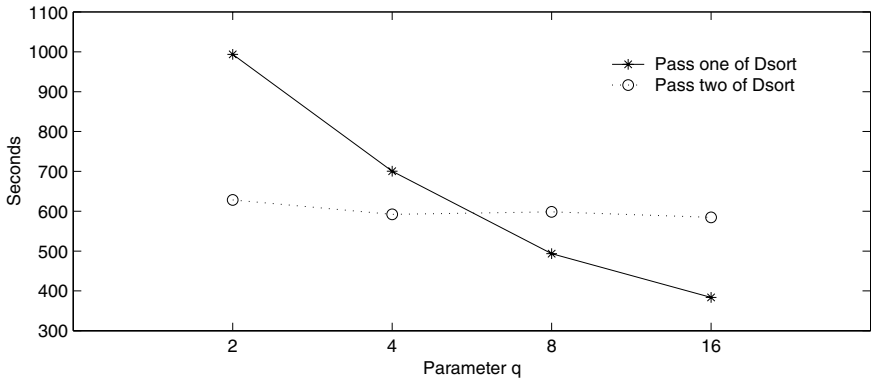


Fig. 3. Observed running times of the two passes of Dsort for four values of q : 2, 4, 8, and 16. These timings are for the following setting of parameters: $P = 16$, $N = 2^{31}$ records or 128 GB, and $\beta = 2^{21}$ records or 128 MB, for 64-byte records.

by the five values of q , with $q = 16$ and $q = 1$ being the best and worst cases for Dsort, respectively.

As mentioned before, the running time of Csort exhibits negligible variation across the various values of q , demonstrating that Csort’s predetermined I/O and communication patterns do indeed translate into a running time that depends almost entirely on the input size. Dsort shows much variation across the various values of q . As Figure 2 shows, Csort runs much faster for two of the five inputs ($q = 1$ and $q = 2$). In two of the remaining three cases ($q = 4$ and $q = 8$), the difference in the running times is marginal. Dsort runs significantly faster in the most favorable case ($q = 16$).

The variation in the running times of Dsort is due to differences in the performance of Pass 1. As explained in Section 3.1, the performance of the communi-

cate and write stages is highly sensitive to the exact input sequence. Specifically, the lower the value of q , the more uneven are the I/O and communication patterns of each round, and therefore, the longer each round takes to complete. We ran Dsort for four values of q , and for each run, we observed the running time of each of the two passes. Figure 3 demonstrates that, across all four values of q , there is little variation in the performance of pass 2. The performance of pass 1, on the other hand, varies greatly with the value of q , thus substantiating our claim that pass 1 is responsible for the variation in the running times of Dsort.

5 Conclusion

This paper presents the first study that compares the paradigm of distribution-based algorithms to that of oblivious algorithms, in the context of out-of-core sorting on distributed-memory clusters. We do so by comparing the performance of two algorithms: Csort and Dsort. Below, we summarize the main differences between Csort and Dsort:

- Csort is a 3-pass oblivious algorithm, and Dsort is a 2-pass distribution-based algorithm.
- Dsort assumes that the the equi-partition splitters are known in advance, thereby obviating the need for the sampling and load balancing steps. To sort all inputs, a distribution-based sort must remove this assumption, necessitating at least one more pass over the data. Csort makes no assumptions about the input distribution.
- The running time of Dsort, even when the input follows the above mentioned assumption, varies greatly with the exact input sequence. This variation arises because the I/O and communication patterns of Dsort are sensitive to the input sequence. The running time of Csort, on the other hand, varies negligibly with the input sequence. Our experimental results demonstrate this difference. This difference also makes Csort much simpler to implement.
- Dsort can handle problem sizes larger than those that Csort can handle. Csort can sort up to $\beta^{3/2}\sqrt{P}/2$ records (512 GB of data in our setup), whereas Dsort can sort up to β^2P/B records (2 TB in our setup).

Both Csort and Dsort are implemented using identical software, and they run on identical hardware. The results of our experiments show that Csort fares well compared to Dsort. On three out of five inputs, Csort runs faster. On one of the remaining two inputs, the difference between the running times of Csort and Dsort is marginal. Dsort runs significantly faster on the other remaining input, the one which represents the rather unrealistic case of $q = 16$.

In future work, we would like to implement a distribution-based algorithm that makes no assumptions about the input distribution and compare its performance with that of an oblivious algorithm. We also plan to continue our efforts toward designing new oblivious algorithms and engineering efficient implementations of them.

References

1. Knuth, D.E.: *Sorting and Searching*. Volume 3 of *The Art of Computer Programming*. Addison-Wesley (1973)
2. Leighton, F.T.: *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann (1992)
3. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: *Introduction to Algorithms*. second edn. The MIT Press and McGraw-Hill (2001)
4. Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H., Culler, D.E., Hellerstein, J.M., Patterson, D.A.: High-performance sorting on networks of workstations. In: *SIGMOD '97*. (1997)
5. Graefe, G.: *Parallel external sorting in Volcano*. Technical Report CU-CS-459-90, University of Colorado at Boulder, Department of Computer Science (1990)
6. Vitter, J.S.: External memory algorithms and data structures: Dealing with MASSIVE DATA. *ACM Computing Surveys* **33** (2001) 209–271
7. Chaudhry, G., Cormen, T.H., Wisniewski, L.F.: Columnsort lives! An efficient out-of-core sorting program. In: *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures*. (2001) 169–178
8. Chaudhry, G., Cormen, T.H.: Getting more from out-of-core columnsort. In: *4th Workshop on Algorithm Engineering and Experiments (ALENEX 02)*. (2002) 143–154
9. Chaudhry, G., Cormen, T.H., Hamon, E.A.: Parallel out-of-core sorting: The third way. (*Cluster Computing*) To appear.
10. Chaudhry, G., Cormen, T.H.: Slabpose columnsort: A new oblivious algorithm for out-of-core sorting on distributed-memory clusters. (*Algorithmica*) To appear.
11. Blelloch, G.E., Leiserson, C.E., Maggs, B.M., Plaxton, C.G., Smith, S.J., Zagha, M.: An experimental analysis of parallel sorting algorithms. *Theory of Computing Systems* **31** (1998) 135–167
12. Snir, M., Otto, S.W., Huss-Lederman, S., Walker, D.W., Dongarra, J.: *MPI—The Complete Reference, Volume 1, The MPI Core*. The MIT Press (1998)
13. Gropp, W., Huss-Lederman, S., Lumsdaine, A., Lusk, E., Nitzberg, B., Saphir, W., Snir, M.: *MPI—The Complete Reference, Volume 2, The MPI Extensions*. The MIT Press (1998)
14. Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H., Culler, D.E., Hellerstein, J.M., Patterson, D.A.: Searching for the sorting record: Experiences in tuning NOW-Sort. In: *1998 Symposium on Parallel and Distributed Tools (SPDT '98)*. (1998)
15. Vitter, J.S., Shriver, E.A.M.: Algorithms for parallel memory I: Two-level memories. *Algorithmica* **12** (1994) 110–147
16. Leighton, T.: Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers* **C-34** (1985) 344–354

Allocating Memory in a Lock-Free Manner^{*}

Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas

Department of Computer Science and Engineering,
Chalmers University of Technology, SE-412 96 Göteborg, Sweden
{andersg, ptrianta, tsigas}@cs.chalmers.se

Abstract. The potential of multiprocessor systems is often not fully realized by their system services. Certain synchronization methods, such as lock-based ones, may limit the parallelism. It is significant to see the impact of wait/lock-free synchronization design in key services for multiprocessor systems, such as the memory allocation service. Efficient, scalable memory allocators for multithreaded applications on multiprocessors is a significant goal of recent research projects.

We propose a lock-free memory allocator, to enhance the parallelism in the system. Its architecture is inspired by Hoard, a successful concurrent memory allocator, with a modular, scalable design that preserves scalability and helps avoiding false-sharing and heap blowup. Within our effort on designing appropriate lock-free algorithms to construct this system, we propose a new non-blocking data structure called flat-sets, supporting conventional “internal” operations as well as “inter-object” operations, for moving items between flat-sets.

We implemented the memory allocator in a set of multiprocessor systems (UMA Sun Enterprise 450 and ccNUMA Origin 3800) and studied its behaviour. The results show that the good properties of Hoard w.r.t. false-sharing and heap-blowup are preserved, while the scalability properties are enhanced even further with the help of lock-free synchronization.

1 Introduction

Some form of dynamic memory management is used in most computer programs for multiprogrammed computers. It comes in a variety of flavors, from the traditional manual general purpose allocate/free type memory allocator to advanced automatic garbage collectors.

In this paper we focus on conventional general purpose memory allocators (such as the “libc” malloc) where the application can request (allocate) arbitrarily-sized blocks of memory and free them in any order. Essentially a memory allocator is an online algorithm that manages a pool of memory (heap), e.g. a contiguous range of addresses or a set of such ranges, keeping track of which parts of that memory are currently given to the application and which parts are unused and can be used to meet future allocation requests from the

^{*} This work was supported by computational resources provided by the Swedish National Supercomputer Centre (NSC).

application. The memory allocator is not allowed to move or otherwise disturb memory blocks that are currently owned by the application.

A good allocator should aim at minimizing *fragmentation*, i.e. minimizing the amount of free memory that cannot be used (allocated) by the application. *Internal fragmentation* is free memory wasted when the application is given a larger memory block than it requested; and *external fragmentation* is free memory that has been split into too small, non-contiguous blocks to be useful to satisfy the requests from the application. Multi-threaded programs add some more complications to the memory allocator. Obviously some kind of synchronization has to be added to protect the heap during concurrent requests. There are also other issues outlined below, which have significant impact on application performance when the application is run on a multiprocessor [1]. Summarizing the goals, a good concurrent memory allocator should (i) avoid *false sharing*, which is when different parts of the same cache-line end up being used by threads running on different processors; (ii) avoid *heap blowup*, which is an overconsumption of memory that may occur if the memory allocator fails to make memory deallocated by threads running on one processor available to threads running on other processors; (iii) ensure *efficiency* and *scalability*, i.e. the concurrent memory allocator should be as fast as a good sequential one when executed on a single processor and its performance should scale with the load in the system.

The Hoard [2] concurrent memory allocator is designed to meet the above goals. The allocation is done on the basis of per-processor heaps, which avoids false sharing and reduces the synchronization overhead in many cases, improving both performance and scalability. Memory requests are mapped to the closest matching size in a fixed set of size-classes, which bounds internal fragmentation. The heaps are sets of superblocks, where each superblock handles blocks of one size class, which helps in coping with external fragmentation. To avoid heap blowup freed blocks are returned to the heap they were allocated from and empty superblocks may be reused in other heaps.

Regarding efficiency and scalability, it is known that the use of locks in synchronization is a limiting factor, especially in multiprocessor systems, since it reduces parallelism. Constructions which guarantee that concurrent access to shared objects is free from locking are of particular interest, as they help to increase the amount of parallelism and to provide fault-tolerance. This type of synchronization is called lock-/wait-free, non-blocking or optimistic synchronization [3,4,5,6]. The potential of this type of synchronization in the performance of system-services and data structures has also been pointed out earlier, in [7,4,8].

The contribution of the present paper is a new memory allocator based on lock-free, fine-grained synchronization, to enhance parallelism, fault-tolerance and scalability. The architecture of our allocation system is inspired by Hoard, due to its well-justified design decisions, which we roughly outlined above. In the process of designing appropriate data structures and lock-free synchronization algorithms for our system, we introduced a new data structure, which we call *flat-set*, which supports a subset of operations of common sets, as well as “inter-object” operations, for moving an item from one flat-set to another in a

lock-free manner. The lock-free algorithms we introduce make use of standard synchronization primitives provided by multiprocessor systems, namely single-word *Compare-And-Swap*, or its equivalent *Load-Linked/Store-Conditional*.

We have implemented and evaluated the allocator proposed here on common multiprocessor platforms, namely an UMA Sun Enterprise 450 running Solaris 9 and a ccNUMA Origin 3800 running IRIX 6.5. We compare our allocator with the standard “libc” allocator of each platform and with Hoard (on the Sun system, where we had the original Hoard allocator available using standard benchmark applications to test the efficiency, scalability, cache behaviour and memory consumption behaviour. The results show that our system preserves the good properties of Hoard, while it offers a higher scalability potential, as justified by its lock-free nature.

In the next section we provide background information on lock- and wait-free synchronization (throughout the paper we use the terms non-blocking and lock-free interchangeably). Earlier and recent related work is discussed in section 7, after the presentation of our method and implementation, as some detail is needed to relate these contributions.

2 Background: Non-blocking Synchronization

Non-blocking implementations of shared data objects are an alternative to the traditional solution for maintaining the consistency of a shared data object (i.e. for ensuring *linearizability* [9]) by enforcing mutual exclusion. Non-blocking synchronization allows multiple tasks to access a shared object at the same time, but without enforcing mutual exclusion [3,4,5,6,10]. Non-blocking synchronization can be lock-free or wait-free. *Lock-free* algorithms guarantee that regardless of the contention caused by concurrent operations and the interleaving of their steps, at each point in time there is at least one operation which is able to make progress. However, the progress of other operations might cause one specific operation to take unbounded time to finish. In a *wait-free* algorithm, every operation is guaranteed to finish in a bounded number of its own steps, regardless of the actions of concurrent operations. Non-blocking algorithms have been shown to have significant impact in applications [11,12], and there is also a library, NOBLE [13], containing many implementations of non-blocking data structures.

One of the most common synchronization primitives used in lock-free synchronization is the *Compare-And-Swap* instruction (also denoted CAS), which atomically executes the steps described in Fig. 1. CAS is available in e.g. SPARC processors. Another primitive which is equivalent with CAS in synchronization power is the *Load-Linked/Store-Conditional* (also denoted LL/SC) pair of instructions, available in, e.g. MIPS processors. LL/SC is used as follows: (i) LL loads a word from memory. (ii) A short sequence of instructions may modify the value read. (iii) SC stores the new value into the memory word, unless the word has been modified by other process(es) after LL was invoked. In the latter case the SC *fails*, otherwise the SC *succeeds*. Another useful primitive is *Fetch-And-Add*

```

atomic CAS(mem : pointer to integer;
            new, old : integer) return integer
    tmp := *mem;
    if tmp == old then
        *mem := new; /* CAS succeeded */
    return tmp;

atomic FAA(mem : pointer to integer;
            increment : integer) return integer
    tmp := *mem;
    *mem := tmp + increment;
    return tmp;

```

Fig. 1. Compare-And-Swap (denoted CAS) and Fetch-And-Add (denoted FAA)

(also denoted *FAA*), described in Fig. 1. FAA can be simulated in software using CAS or LL/SC when it is not available in hardware.

An issue that sometimes arises in connection with the use of CAS, is the so-called *ABA problem*. It can happen if a thread reads a value A from a shared variable, and then invokes a CAS operation to try to modify it. The CAS will (undesirably) succeed if between the read and the CAS other threads have changed the value of the shared variable from A to B and back to A. A common way to cope with the problem is to use version numbers of b bits as part of the shared variables [14]. An alternative method to cope with the ABA problem is to introduce special NULL values. This method is proposed and used in a lock-free queue implementation in [15]. An appropriate garbage-collection mechanism, such as [16], can also solve the problem.

3 The New Lock-Free Memory Allocator: Architecture

The architecture of our lock-free memory allocator is inspired by Hoard[2], which is a well-known and practical concurrent memory allocator for multiprocessors.

The memory allocator provides allocatable memory of a fixed set of sizes, called *size-classes*. The size of memory requests from the application are rounded upwards to the closest size-class. To reduce false-sharing and contention, the memory allocator distributes the memory into *per-processor heaps*. The managed memory is handled internally in units called *superblocks*. Each superblock contains allocatable blocks of one size-class. Initially all superblocks belong to the *global heap*. During an execution superblocks are moved to per-processor heaps as needed. When a superblock in a per-processor heap becomes almost empty (i.e. few of its blocks are allocated) it is moved back to the global heap. The superblocks in a per-processor heap are stored and handled separately, based on their size-class. Within each size-class the superblocks are kept sorted into bins based on *fullness*(cf. Fig. 2(a)). As the fullness of a particular superblock changes it is moved between the groups. A memory request (*malloc* call) first searches for a superblock with a free block among the superblocks in the “almost full” fullness-group of the requested size-class in the appropriate per-processor heap. If no suitable superblock is found there, it will proceed to search in the lower fullness-groups, and, if that, too, is unsuccessful, it will request a new superblock from the global heap. Searching the almost full superblocks first reduces external fragmentation. When freed (by a call to *free*) an allocated block is returned to the superblock it was allocated from and, if the new fullness requires so, the superblock is moved to another fullness-group.

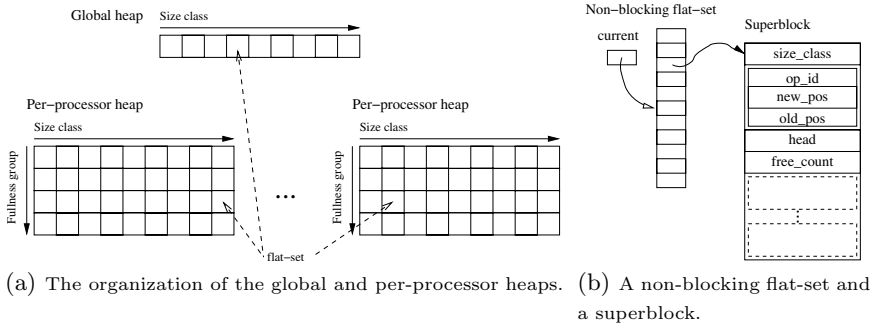


Fig. 2. The architecture of the memory allocator

4 Managing Superblocks: The Bounded Non-blocking Flat-Sets

Since the number of superblocks in each fullness-group varies over time, a suitable collection-type data structure is needed to implement a fullness-group. Hoard, which uses mutual-exclusion on the level of per-processor heaps, uses linked-lists of superblocks for this purpose, but this issue becomes very different in a lock-free allocator. While there exist several lock-free linked-list implementations, e.g. [17,18,14], we cannot apply those here, because not only do we want the operations on the list to be lock-free, but we also need to be able to move a superblock from one set to another without making it inaccessible to other threads during the move. To address this, we propose a new data structure we call a *bounded non-blocking flat-set*, supporting conventional “internal” operations (*Get_Any* and *Insert* item) as well as “inter-object” operations, for moving an item from one flat-set to another.

To support “inter”-flat-set operations it is crucial to be able to move superblocks from one set to another in a lock-free fashion. The requirements that make this difficult are: (i) the superblock should be reachable for other threads even while it is being moved between flat-sets, i.e. a non-atomic *first-remove-then-insert* sequence is not acceptable; (ii) the number of shared references to the superblock should be the same after a move (or set of concurrent move operations) finish.

Below we present the operations of the lock-free flat-set data structure and the lock-free algorithm, *move*, which is used to implement the “inter-object” operation for moving a reference to a superblock from one shared variable (pointer) to another satisfying the above requirements.

4.1 Operations on Bounded Non-blocking Flat-Sets

A bounded non-blocking flat-set provides the following operations: (i) *Get_Any*, which returns any item in the flat-set; and (ii) *Insert*, which inserts an item into the flat-set. An item can only reside inside one flat-set at the time; when an item is inserted into a flat-set it is also removed from its old location. The flat-set data

structure consists of an array of M shared locations `set.set[i]`, each capable of holding a reference to a superblock, and a shared index variable `set.current`. The data structure and operations are shown in Fig. 3 and are briefly described below.

The index variable `set.current` is used as a *marker* to speed up flat-set operations. It contains a bit used as an *empty flag* for the flat-set and a index field that is used as the starting point for searches, both for items and for free slots. The *empty flag* is set by a *Get_Any* operation that discovers that the flat-set is empty, so that subsequent *Get_Any* operations know this; the *Insert* operation and successful *Get_Any* operations clear the flag. The *empty flag* is not always set when the flat-set is empty as superblocks can be moved away from the flat-set at any time, but it is always cleared when the flat-set is nonempty. The *Insert* operation scans the array `set.set[i]` forward from the position marked by `set.current` until it finds an empty slot. It will then attempt to move the superblock reference to be inserted into this slot using the *Move* operation (described in detail below). The *Get_Any* operation first reads `set.current` to check the *empty flag*. If the *empty flag* is set, *Get_Any* returns immediately, otherwise it starts to scan the array `set.set[i]` backwards from the position marked by `set.current`, until it finds a location that contains a superblock reference. If a *Get_Any* operation has scanned the whole `set.set[i]` array without finding a reference it will try to set the *empty flag* for the flat-set. This is done at line G13 using CAS and will succeed if and only if `set.current` has not been changed since it was read at line G2. This indicates that the flat-set is empty so *Get_Any* sets the empty flag and returns failure. If, on the other hand, `set.current` has changed between line G2 and G13, then either an *Insert* is in progress or has finished during the scan (line I6 and I9) or some other *Get_Any* has successfully found a superblock during this time (line G10), so *Get_Any* should redo the scan. To facilitate moving of superblocks between flat-sets via *Insert* *Get_Any* returns both a superblock reference and a reference to the shared location containing it.

4.2 How to Move a Shared Reference: Moving Items Between Flat-Sets

The algorithm supporting the operation *Move* moves a superblock reference `sb` from a shared location `from` to a shared location `to`. The target location (i.e. `to`) is known via the *Insert* operation. The algorithm requires the superblock to contain an auxiliary variable `mv_info` with the fields `op_id`, `new_pos` and `old_pos` and all superblock references to have a version field (cf. Fig 3).

A move operation may *succeed* by returning `SB_MOVED_OK` or *fail* (abort) by returning `SB_MOVED` (if the block has been moved by another overlapping move) or `SB_NOT_MOVED` (if the `to` location is occupied). It will succeed if it is completed successfully by the thread that initiated it or by a helping thread. To ensure the lock-free property, the move operation is divided into a number of atomic suboperations. A move operation that encounters an unfinished move of the same superblock will *help* the old operation to finish before it attempts to perform its own move. The helping procedure is identical to steps 2 - 4 of the move operation described below.

```

type superblock_ref { // fits in one machine word
  ptr : integer_16; version : integer_16;};
/* superblock_ref utility functions. */
function pointer(ref : superblock_ref)
return pointer to superblock
function version(ref : superblock_ref)
return integer_16
function make_sb_ref(sb : pointer to superblock,
  op_id : integer_16) return superblock_ref
type flat-set_info { // fits in one machine word
  index : integer; empty : boolean; version : integer;};
function Get_Any(set : in out flat-set,
  sb : in out superblock_ref,
  loc : in out pointer to superblock_ref)
return status
  i, j : integer; old_current : flat-set_info;
begin
G1 loop
G2   old_current := set.current;
G3   if old_current.empty then
G4     return FAILURE;
G5   i := old_current.index;
G6   for j := 1 .. set.size do
G7     sb := set.set[i];
G8     if pointer(sb) /= null then
G9       loc := &set.set[i];
G10    set.current := (i, false); // Clear empty flag
G11    return SUCCESS;
G12   if i == 0 then i := set.size - 1 else i--;
G13   if CAS(&set.current, old_current,
G14     (old_current.index, true)) == old_current
G15   then
G16     return FAILURE;
function Insert(set : in out flat-set,
  sb : in superblock_ref,
  loc : in out pointer to superblock_ref)
return status
  i, j : integer;
begin
I1 loop
I2   i := (set.current.index + 1) mod set.size;
I3   for j := 1 .. set.size do
I4     while pointer(set.set[j]) == null do
I6     set.current := (i, false);
I7     case Move(sb, loc, &set.set[j]) is
I8       when SB_MOVED_OK:
I9         set.current := (i, false);
I10        loc := &set.set[j];
I11        return SB_MOVED_OK;
I12       when SB_MOVED:
I13         return SB_MOVED;
I14       when others:
I15         end case;
I16     i := (i + 1) mod set.size;
I17   if set.set not changed since prev. iter. then
I18     return FAILURE; /* The flat-set is full. */
function Get_Block(sb : in superblock_ref)
return block_ref
  nb, nh : block_ref;
begin
GB1 nb := sb.freelist_head;
GB2 while nb /= null do
GB3   nh := CAS(&sb.freelist_head,
GB4     nb, nb.next);
GB5   if nh == nb.next then
GB6     FAA(&sb.free_block_cnt, -1);
GB7   break;
GB8   nb := sb.freelist_head;
GB9 return nb;
structure flat-set {
  size : constant integer; current : flat-set_info;
  set[size] : array of superblock_ref;};
structure superblock {
  mv_info : move_info;
  freelist_head : pointer to block;
  free_block_cnt : integer;};
structure move_info {
  op_id : integer_16;
  new_pos : pointer to superblock_ref;
  old_pos : pointer to superblock_ref;};
type block_ref { // fits in one machine word
  offset : integer_16; version : integer_16;};
procedure Put_Block(sb : in superblock_ref,
  bl : in block_ref)
  oh : block_ref;
begin
PB1 loop
PB2   bl.next := sb.freelist_head;
PB3   oh := CAS(&sb.freelist_head, bl.next, bl)
PB4   if oh == bl.next then break;
PB5   FAA(&sb.free_block_cnt, 1);
function Move(sb : in superblock_ref,
  from : in pointer to superblock_ref,
  to : in pointer to superblock_ref)
return status
  new_op, old_op : move_info;
  cur_from : superblock_ref;
begin
M1 /* Step 1: Initiate move. */
M2 loop
M3   old_op := Load_Linked(&sb.mv_info);
M4   cur_from := *from;
M5   if pointer(cur_from) /= pointer(sb) then
M6     return SB_MOVED;
M7   if old_op.from == null then // No cur. operation.
M8     new_op := (version(cur_from), to, from);
M9     if Store_Conditional(&sb.mv_info, new_op)
M10    then break;
M11   else
M12     Move_Help(make_sb_ref(pointer(sb), old_op.op_id),
M13       old_op.old_pos,
M14       old_op.new_pos);
M15   return Move_Help(cur_from, from, to);
function Move_Help(sb : in superblock_ref,
  from : in pointer to superblock_ref,
  to : in pointer to superblock_ref)
return status
  old, new, res : superblock_ref; mi : move_info;
begin
H1 /* Step 2: Update "TO". */
H2   old := *to;
H3   new := make_sb_ref(sb, version(old) + 1);
H4   res := CAS(to, make_sb_ref(null, version(old)), new)
H5   if pointer(res) /= pointer(sb) then
H6     /* To is occupied, abandon this operation. */
H7     mi := Load_Linked(&sb.mv_info);
H8     if mi == (version(sb), to, from) then
H9       mi := (0, from, null);
H10    Store_Conditional(&sb.mv_info, mi);
H11    return SB_NOT_MOVED;
H12 /* Step 3: Clear "FROM". */
H13 CAS(from, sb, make_sb_ref(null, version(sb) + 1));
H14 /* Step 4: Remove operation information.*/
H15 mi := Load_Linked(&sb.mv_info);
H16 if mi == (version(sb), to, from) then
H17   mi := (0, to, null);
H18   Store_Conditional(&sb.mv_info, mi);
H19 return SB_MOVED_OK;

```

Fig. 3. The flat-set data structures and operations *Get_Any* and *Insert*, the superblock data structures and operations *Get_block* and *Put_Block* and the superblock *Move* operation.

1. A *Move*(sb, from, to) is initiated by atomically *registering* the operation. This is done by *Load-Linked/Store-Conditional* operations which sets sb.mv_info to (version(sb), to, from) iff the read value of sb.mv_info.from was null, which indicates that there are no ongoing move of this superblock. If the read value of sb.mv_info.op_id was nonzero, then there is an ongoing move that needs to be helped before this one can proceed. If the reference to the superblock disappears from from before this move has been registered, this move operation is abandoned and returns SB_MOVED.
2. If the current value of to is null then to is set to point to the superblock while simultaneously increasing its version. Otherwise this move is abandoned since the destination is occupied and the information about the move is removed from the superblock (as in step 4) and SB_NOT_MOVED is returned.
3. If from still contains the expected superblock reference (i.e. if no one else has helped this move) from is set to null while increasing its version.
4. If the move information is still in the superblock (i.e. if no one else has helped the move to complete) it is removed and the *move* operation returns SB_MOVED_OK.

In the presentation here and in the pseudo-code in Fig. 3 we use the atomic primitive CAS to update shared variables that fit in a single memory word, but other atomic synchronization primitives, such as LL/SC could be used as well. The auxiliary mv_info variable in a superblock might need to be larger than one word. To handle that we use the lock-free software implementation of Load-Linked/Store-Conditional for large words by Michael [19] which can be implemented efficiently from the common single-word CAS. Some hardware platforms provide a CAS primitive for words twice as wide as the standard word size, which may also be used for this.

The correctness proof of the algorithm is omitted due to space constraints; it can be found in [20].

5 Managing the Blocks Within a Superblock

The allocatable memory blocks within each superblock are kept in a lock-free IBM free-list [21]. The IBM free-list is essentially a lock-free stack implemented from a single-linked-list where the push and pop operations are done by a CAS operation on the head-pointer. To avoid ABA-problems the head-pointer contains a version field. Each block has a header containing a pointer to the superblock it belongs to and a next pointer for the free-list. The two free-list operations *Get_Block* and *Put_Block* are shown in Fig. 3. The free blocks counter, sb.free_block_cnt, is used to estimate the fullness of a superblock.

6 Performance Evaluation

Systems. The performance of the new lock-free allocator has been measured on a two multiprocessor systems: (i) an UMA Sun Enterprise 450 with 4 400MHz

UltraSPARC II (4MB L2 cache) processors running Solaris 9; (ii) a ccNUMA SGI Origin 3800 with 128 (only 32 could be reserved) 500Mhz MIPS R14000 (8MB L2 cache) processors running IRIX 6.5.

Benchmarks. We used three common benchmarks to evaluate our memory allocator: The **Larson** [2,1,22] benchmark simulates a multi-threaded server application which makes heavy use of dynamic memory. Each thread allocates and deallocates objects of random sizes (between 5 to 500 bytes) and also transfers some of the objects to other threads to be deallocated there. The benchmark result is throughput in terms of the number of allocations and deallocations per second which reflects the allocator's behaviour with respect to false-sharing and scalability, and the resulting memory footprint of the process which should reflect any tendencies for heap blowup. We measured the throughput during 60 (30 on the Origin 3800 due to job duration limits) second runs for each number threads.

The **Active-false** and **passive-false** [2,1] benchmarks measure how the allocator handles active (i.e. directly caused by the allocator) respective passive (i.e. caused by application behaviour) false-sharing. In the benchmarks each thread repeatedly allocates an object of a certain size (1 byte) and read and write to that object a large number of times (1000) before deallocating it again. If the allocator does not take care to avoid false-sharing several threads might get objects located in the same cache-line which will slow down the reads and writes to the objects considerably. In the *passive-false* benchmark all initial objects are allocated by one thread and then transferred to the others to introduce the risk of passive false-sharing when those objects are later freed for reuse by the threads. The benchmark result is the total wall-clock time for performing a fixed number (10^6) of allocate-read/write-deallocate cycles among all threads.

Implementation.¹ In our memory allocator we use the CAS primitive (implemented from the hardware synchronization instructions available on the respective system) for our lock-free operations. To avoid ABA problems we use the version number solution ([14], cf. section 2). We use 16-bit version numbers for the superblock references in the flat-sets, since for a bad event (i.e. that a CAS of a superblock reference succeeds when it should not) to happen not only must the version numbers be equal but also that same superblock must have been moved back to the same location in the flat-set, which contains thousands of locations. We use superblocks of 64KB to have space for version numbers in superblock pointers. We also use size-classes that are powers of two, starting from 8 bytes. This is not a decision forced by the algorithm; a more tightly spaced set of size-classes can also be used, which would further reduce internal fragmentation at the cost of a larger fixed space overhead due to the preallocated flat-sets for each size-class. Blocks larger than 32KB are allocated directly from the operating system instead of being handled in superblocks. Our implementation uses four fullness-groups and a fullness-change-threshold of $\frac{1}{4}$, i.e. a

¹ Our implementation is available at <http://www.cs.chalmers.se/~dcs/nbmalloc.html>.

superblock is not moved to a new group until its fullness is more than $\frac{1}{4}$ outside its current group. This prevents superblocks from rapidly oscillating between fullness-groups. Further, we set the maximum size for the flat-sets used in the global heap and for those in per-processor heaps to 4093 superblocks each (these values can be adjusted separately).

Results. In the evaluation we compare our allocator with the standard “libc” allocator of the respective platform using the above standard benchmark applications. On the Sun platform, for which we had the original Hoard allocator available, we also compare with Hoard (version 3.0.2). To the best of our knowledge, Hoard is not available for ccNUMA SGI IRIX platform.

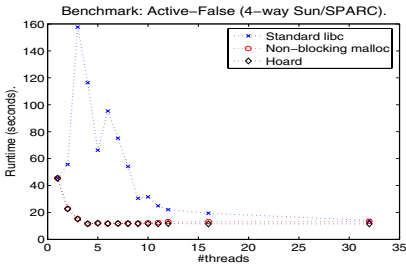
The benchmarks are intended to test scalability, fragmentation and false-sharing, which are the evaluation criteria of a good concurrent allocator, as explained in the introduction. When performing these experiments our main goal was not to optimize the performance of the lock-free allocator, but rather to examine the benefits of the lock-free design itself. There is plenty of room for optimization of the implementation.

The results from the two false-sharing benchmarks, shown in Fig. 4, show that our memory allocator, and Hoard, induce very little false-sharing. The standard “libc” allocator, on the other hand, suffers significantly from false-sharing as shown by its longer and irregular runtimes. Our allocator shows consistent behaviour as the number of processors and memory architecture changes.

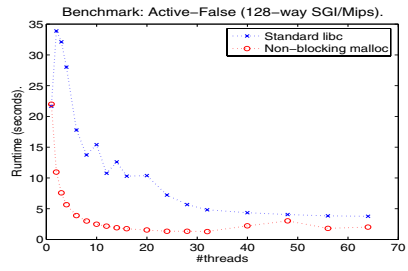
The throughput results from the Larson benchmark, shown in Fig. 4, show that our lock-free memory allocator has good scalability, not only in the case of full concurrency (where Hoard also shows extremely good scalability), but also when the number of threads increases beyond the number of processors. In that region, Hoard’s performance quickly drops from its peak at full concurrency (cf. Fig. 4(e)). We can actually observe more clearly the scalability properties of the lock-free allocator in the performance diagrams on the SGI Origin platform (Fig. 4(f)). There is a linear-style of throughput increase when the number of processors increases (when studying the diagrams recall we have up to 32 processors available on the Origin 3800). Furthermore, when the load on each processor increases beyond 1, the throughput of the lock-free allocator stays high. In terms of absolute throughput, Hoard is superior to our lock-free allocator, at least on the Sun platform where we had the possibility to compare them. This is not surprising, considering that it is very well designed and has been around enough time to be well tuned. An interesting conclusion is that the scalability of Hoard’s architecture is further enhanced by lock-free synchronization.

The results with respect to memory consumption, Fig. 4(g,h), show that for the Larson benchmark the memory usage (and thus fragmentation) of the non-blocking allocator stays at a similar level to Hoard and that the use of per-processor heaps with thresholds, while having a larger overhead than the “libc” allocator, still have almost as good scalability with respect to memory utilization as a single heap allocator.

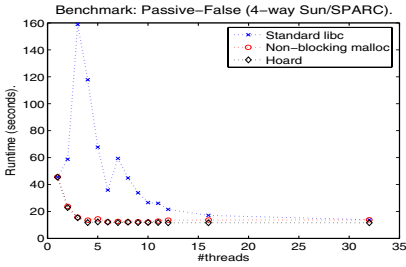
Moreover, that our lock-free allocator shows a very similar behaviour in throughput on both the UMA and the ccNUMA systems is an indication that



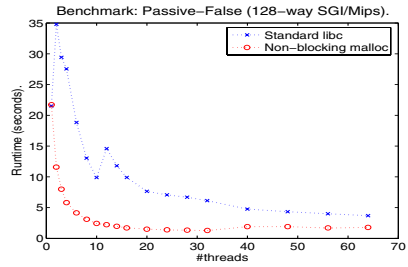
(a) Active-False: Sun SPARC 4 CPUs



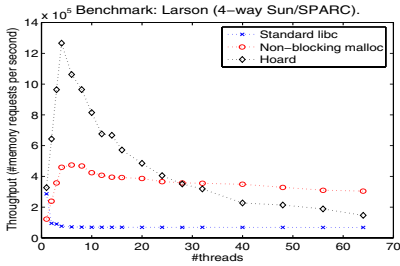
(b) Active-False: SGI MIPS 32(/128) CPUs



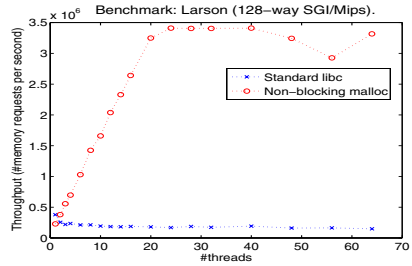
(c) Passive-False: Sun SPARC 4 CPUs



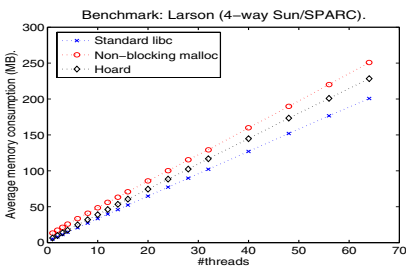
(d) Passive-False: SGI MIPS 32(/128) CPUs



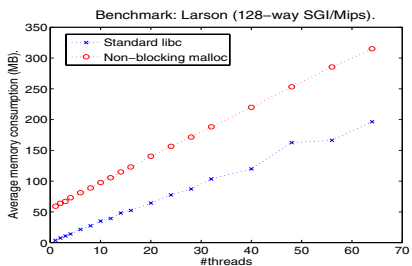
(e) Throughput: Sun SPARC 4 CPUs



(f) Throughput: SGI MIPS 32(/128) CPUs



(g) Memory consumption: Sun SPARC 4 CPUs



(h) Memory consumption: SGI MIPS 32(/128) CPUs

Fig. 4. Results from the benchmarks

there are few contention hot-spots, as these tend to cause much larger performance penalties on NUMA than on UMA architectures.

7 Other Related Work

Recently Michael presented a lock-free allocator [23] which, like our contribution, is loosely based on the Hoard architecture. Our work and Michael's have been done concurrently and completely independently, an early version of our work is in the technical report [20]. Despite both having started from the Hoard architecture, we have used two different approaches to achieve lock-freeness. In Michael's allocator each per-processor heap contains one active (i.e. used by memory requests) and at most one inactive partially filled superblock per size-class, plus an unlimited number of full superblocks. All other partially filled superblocks are stored globally in per-size-class FIFO queues. It is an elegant algorithmic construction, and from the scalability and throughput performance point of view it performs excellently, as is shown in [23], in the experiments carried out on a 16-way POWER3 platform. By further studying the allocators, it is relevant to note that: Our allocator and Hoard keep all partially filled superblocks in their respective per-processor heap while the allocator in [23] does not and this may increase the potential for inducing false-sharing. Our allocator and Hoard also keep the partially filled superblocks sorted by fullness and not doing so, like the allocator in [23] does, may imply some increased risk of external fragmentation since the fullness order is used to direct allocation requests to the more full superblocks which makes it more likely that less full ones becomes empty and thus eligible for reuse. The allocator in [23], unlike ours, uses the *first-remove-then-insert* approach to move superblocks around, which in a concurrent environment could affect the fault-tolerance of the allocator and cause unnecessary allocation of superblocks since a superblock is invisible to other threads while it is being moved. As this is work that has been carried out concurrently and independently with our contribution, we do not have any measurements of the impact of the above differences, however this is interesting to do as part of future work, towards further optimization of these allocators.

Another allocator which reduces the use of locks is LFMalloc [7]. It uses a method for almost lock-free synchronization, whose implementation requires the ability to efficiently manage CPU-data and closely interact with the operating system's scheduler. To the best of our knowledge, this possibility is not directly available on all systems. LFMalloc is also based on the Hoard design, with the difference in that it limits each per-processor heap to at most one superblock of each size-class; when this block is full, further memory requests are redirected to the global heap where blocking synchronization is used and false-sharing is likely to occur. However, a comparative study with that approach can be worthwhile, when it becomes available for experimentation.

Earlier related work is the work on non-blocking operating systems by Masalin and Pu [8,24] and Greenwald and Cheriton [4,25]. They, however, made extensive use of the *2-Word-Compare-And-Swap* primitive in their algorithms.

This primitive can update two arbitrary memory locations in one atomic step but is not available in current systems and expensive to do in software.

8 Discussion

The lock-free memory allocator proposed in this paper confirms our expectation that fine-grain, lock-free synchronization is useful for scalability under increasing load in the system. To the best of our knowledge, this, together with the allocator which was independently presented in [23] are also the first lock-free general allocators (based on single-word CAS) in the literature. We expect that this contribution will have an interesting impact in the domain of memory allocators.

Acknowledgements

We would like to thank Håkan Sundell for interesting discussions on non-blocking methods and Maged Michael for his helpful comments on an earlier version of this paper.

References

1. Berger, E.D.: Memory Management for High-Performance Applications. PhD thesis, The University of Texas at Austin, Department of Computer Sciences (2002)
2. Berger, E., McKinley, K., Blumofe, R., Wilson, P.: Hoard: A scalable memory allocator for multithreaded applications. In: ASPLOS-IX: 9th Int. Conf. on Architectural Support for Programming Languages and Operating Systems. (2000) 117–128
3. Barnes, G.: A method for implementing lock-free shared data structures. In: Proc. of the 5th Annual ACM Symp. on Parallel Algorithms and Architectures, SIGACT and SIGARCH (1993) 261–270 Extended abstract.
4. Greenwald, M., Cheriton, D.R.: The synergy between non-blocking synchronization and operating system structure. In: Operating Systems Design and Implementation. (1996) 123–136
5. Herlihy, M.: Wait-free synchronization. *ACM Transaction on Programming and Systems* **11** (1991) 124–149
6. Rinard, M.C.: Effective fine-grain synchronization for automatically parallelized programs using optimistic synchronization primitives. *ACM Transactions on Computer Systems* **17** (1999) 337–371
7. Dice, D., Garthwaite, A.: Mostly lock-free malloc. In: ISMM'02 Proc. of the 3rd Int. Symp. on Memory Management. ACM SIGPLAN Notices, ACM Press (2002) 163–174
8. Massalin, H., Pu, C.: A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91 (1991)
9. Herlihy, M.P., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* **12** (1990) 463–492

10. Hoepman, J.H., Papatriantafilou, M., Tsigas, P.: Self-stabilization of wait-free shared memory objects. *Journal of Parallel and Distributed Computing* **62** (2002) 766–791
11. Tsigas, P., Zhang, Y.: Evaluating the performance of non-blocking synchronisation on shared-memory multiprocessors. In: *Proc. of the ACM SIGMETRICS 2001/Performance 2001*, ACM press (2001) 320–321
12. Tsigas, P., Zhang, Y.: Integrating non-blocking synchronisation in parallel applications: Performance advantages and methodologies. In: *Proc. of the 3rd ACM Workshop on Software and Performance (WOSP'02)*, ACM press (2002) 55–67
13. Sundell, H., Tsigas, P.: NOBLE: A non-blocking inter-process communication library. In: *Proc. of the 6th Workshop on Languages, Compilers and Run-time Systems for Scalable Computers*. Lecture Notes in Computer Science, Springer Verlag (2002)
14. Valois, J.D.: Lock-free linked lists using compare-and-swap. In: *Proc. of the 14th Annual ACM Symp. on Principles of Distributed Computing (PODC '95)*, ACM (1995) 214–222
15. Tsigas, P., Zhang, Y.: A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In: *Proc. of the 13th annual ACM symp. on Parallel algorithms and architectures*, ACM Press (2001) 134–143
16. Michael, M.M.: Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In: *Proc. of the 21st annual symp. on Principles of distributed computing*, ACM Press (2002) 21–30
17. Harris, T.L.: A pragmatic implementation of non-blocking linked lists. In: *Proc. of the 15th Int. Conf. on Distributed Computing*, Springer-Verlag (2001) 300–314
18. Michael, M.M.: High performance dynamic lock-free hash tables and list-based sets. In: *Proc. of the 14th Annual ACM Symp. on Parallel Algorithms and Architectures (SPAA-02)*, ACM Press (2002) 73–82
19. Michael, M.M.: Practical lock-free and wait-free LL/SC/VL implementations using 64-bit CAS. In: *Proc. of the 18th Int. Conf. on Distributed Computing (DISC)*. (2004)
20. Gidenstam, A., Papatriantafilou, M., Tsigas, P.: Allocating memory in a lock-free manner. Technical Report 2004-04, Computing Science, Chalmers University of technology (2004)
21. IBM: IBM System/370 Extended Architecture, Principles of Operation. (1983) Publication No. SA22-7085.
22. Larson, P.P.Å., Krishnan, M.: Memory allocation for long-running server applications. In: *ISMM'98 Proc. of the 1st Int. Symp. on Memory Management*. ACM SIGPLAN Notices, ACM Press (1998) 176–185
23. Michael, M.: Scalable lock-free dynamic memory allocation. In: *Proc. of SIGPLAN 2004 Conf. on Programming Languages Design and Implementation*. ACM SIGPLAN Notices, ACM Press (2004)
24. Massalin, H.: *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Columbia University (1992)
25. Greenwald, M.B.: *Non-blocking synchronization and system design*. PhD thesis, Stanford University (1999)

Generating Realistic Terrains with Higher-Order Delaunay Triangulations

Thierry de Kok, Marc van Kreveld, and Maarten Löffler

Institute of Information and Computing Sciences,
Utrecht University, The Netherlands
{takok, marc, mloffler}@cs.uu.nl

Abstract. For hydrologic applications, terrain models should have few local minima, and drainage lines should coincide with edges. We show that triangulating a set of points with elevations such that the number of local minima of the resulting terrain is minimized is NP-hard for degenerate point sets. The same result applies when there are no degeneracies for higher-order Delaunay triangulations. Two heuristics are presented to reduce the number of local minima for higher-order Delaunay triangulations, which start out with the Delaunay triangulation. We give efficient algorithms for their implementation, and test on real-world data how well they perform. We also study another desirable drainage characteristic, namely few valley components.

1 Introduction

A fundamental geometric structure in computational geometry is the triangulation. It is a partitioning of a point set or region of the plane into triangles. A triangulation of a point set P partitions the convex hull of P into triangles whose vertices are exactly the points of P . The most common triangulation of a set of points is the Delaunay triangulation. It has the property that for every triangle, the circumcircle through its vertices does not contain any points inside. It maximizes the smallest angle over all possible triangulations of the point set.

If the points all have an associated attribute value like elevation, then a triangulation defines a piecewise linear interpolant. Due to the angle property, triangles in a Delaunay triangulation are generally well-shaped and are suitable for spatial interpolation. When using triangulations for terrain modeling, however, one should realize that terrains are formed by natural processes. This implies that there are linear depressions (valleys) formed by water flow, and very few local minima occur [15,17]. Local minima can be caused by erroneous triangulation: an edge may stretch from one side of a valley to the opposite side. Such an edge is an artificial dam, and upstream from the dam in the valley, a local minimum appears. See Figure 1 (left). It is often an artifact of the triangulation. Therefore, minimizing local minima is an optimization criterion for terrain modeling. In extension, the contiguity of valley lines is also a natural phenomenon. Valleys do not start and stop halfway a mountain slope, but the Delaunay triangulation may contain such artifacts. Hence, a second optimization criterion is minimizing the number of valley line components.

Terrain modeling in GIS is used for morphological processes like drainage and erosion, or for hazard analysis like avalanches and landslides. Local minima and undesirable valley lines can influence the computation of these processes, leading to unreliable outcomes. This motivates our study for the construction of realistic terrains by avoiding local minima and artifact valleys.

An alternative to deal with local minima is flooding. Here a local minimum and its surroundings are elevated until a height at which a local minimum does not appear anymore (pit filling) [9–11]. Such methods help to create terrains with better drainage characteristics, or to define a drainage basin hierarchy, but it is clear that other artifacts are introduced at the same time. Furthermore, this approach does not respect the given input elevations.

Returning to planar triangulations, there are many different ways in which one can define the quality of a triangulation of a set of points. A criterion that is always important for triangulations is the nice shape of the triangles. This can be formalized in several ways [2,3]. In this paper, nice shape is formalized by *higher-order Delaunay triangulations* [7]. They provide a class of triangulations that are all reasonably well-shaped, depending on a parameter k .

Definition 1. *A triangle in a point set P is order- k if its circumcircle contains at most k points of P . A triangulation of a set P of points is an order- k Delaunay triangulation if every triangle of the triangulation is order- k (see Figure 1 (middle)).*

So a Delaunay triangulation is an order-0 Delaunay triangulation. For any positive integer k , there can be many different order- k Delaunay triangulations. The higher k , the more freedom to eliminate artifacts like local minima, but the worse the shape of the triangles can become.

This paper discusses triangulations of a point set P of which elevations are given for terrain modeling. In Section 2 we concentrate on minimizing local minima. We show that over all possible triangulations, minimizing local minima is

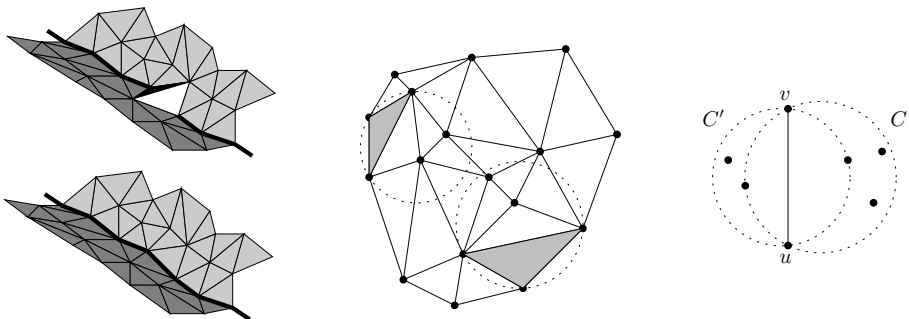


Fig. 1. Left, an artificial dam and local minimum in a terrain. Middle, an order-2 Delaunay triangulation, with two triangles and their circumcircles, showing the order. Right, the useful order of the edge \overline{uv} is 3, the maximum of the number of points inside C or C' .

NP-hard. This result relies heavily on degenerate point sets. For order- k Delaunay triangulations, NP-hardness can also be shown for non-degenerate point sets, for $k = \Omega(n^\epsilon)$ and $k \leq c \cdot n$, for some $0 < \epsilon < 1$ and some $0 < c < 1$. (For $k = 1$, an $O(n \log n)$ time algorithm that minimizes local minima was given in [7].) Then we discuss two heuristics for reducing local minima in order- k Delaunay triangulations, the flip and hull heuristics, and their efficiency. The latter was introduced before in [7]; here we give a more efficient algorithm. Then we compare the two heuristics experimentally on various terrains. We only examine orders 0 up to 8; higher orders are less interesting in practice since the interpolation quality may be less good, and artifacts may appear in visualization.

In Section 3 we extend our study to deal with valley line components as well. It appears that the removal of local minima actually can create artifact valley lines, especially for the flip heuristic. Two solutions are presented for this problem. We give a method to remove isolated valley edges, and to extend valley line components so that they join with other valley line components. In short, we try to reduce the number of valley line components, leading to the valley heuristic. We complement our methods by experiments on various terrains.

Section 4 gives the conclusions and lists several directions for further research.

2 Minimizing the Number of Local Minima

The next subsection shows NP-hardness of minimizing local minima in two settings. Subsections 2.2 and 2.3 present the flip and hull heuristics and contain a running time analysis. Both heuristics can be implemented in $O(nk^2 + nk \log n)$ time, where n is the number of points and k is the order of the higher-order Delaunay triangulation. For the hull heuristic this is an improvement over the $O(nk^3 + nk \log n)$ time bound of Gudmundsson et al. [7].

We define the *useful order* of an edge as the lowest order of a triangulation that includes this edge. In [7] it was shown that the useful order is actually defined by the number of points in one of two circles, see Figure 1 (right). Let \overline{uv} be the edge whose useful order we wish to determine, and assume without loss of generality that \overline{uv} is vertical. Let C (and C') be the circle that passes through u and v , with leftmost (resp. rightmost) center, and which does not contain in its interior any point left (resp. right) of the line through u and v . If k is the maximum of the number of points inside C and C' , then the useful order of edge \overline{uv} is k .

2.1 NP-Hardness of Minimizing Local Minima

For a set P of n points in the plane, it is easy to compute a triangulation that minimizes the number of local minima if there are no degeneracies (no three points on a line). Assume p is the lowest point. Connect every $q \in P \setminus \{p\}$ with p to create a star network with p as the center. Complete this set of edges to a triangulation in any way. Since every point but p has a lower neighbor, no point but p can be a local minimum. Hence, this triangulation is one that minimizes the number of local minima. When degeneracies are present, minimizing the number of local minima is NP-hard.

Theorem 1. *Let P be a set of n points in the plane, and assume that the points have elevations. It is NP-hard to triangulate P with the objective to minimize the number of local minima of the polyhedral terrain.*

Proof. (Sketch; full proof in full paper [5]) By reduction from maximum size non-intersecting subset in a set of line segments [1]. Let S be any set of n line segments in the plane, and assume all $2n$ endpoints are disjoint (this can easily be enforced by extending segments slightly). We place extra points (shields) in between two endpoints of different line segments. For each segment of S , assign one endpoint elevation 1 and the other 2. Assign elevation 3 to shields. Now minimizing local minima is the same as maximum non-intersecting subset in S . \square

Based on the construction in the proof above, we can show NP-hardness of minimizing the number of local minima for higher-order Delaunay triangulations even when no degeneracies exist.

Corollary 1. *Let P be a set of n points in the plane such that no three points of P lie on a line, and assume that the points have elevations. For any $0 < \epsilon < 1$ and some $0 < c < 1$, it is NP-hard to compute an order- k Delaunay triangulation that minimizes the number of local minima of the polyhedral terrain for $n^\epsilon \leq k \leq c \cdot n$.*

Proof. (Sketch; full proof in full paper [5]) Start out with the proof of the theorem above, but move each shield s slightly to break collinearity. The move should be so small that the circle through the two endpoints of different line segments for which s was shield, and through s itself, is huge. Then place many extra points to make sure that these huge circles contain many points, effectively causing that the useful order of the edge between two endpoints that may not be connected is larger than allowed. \square

2.2 The Flip Heuristic

Given a value of k , the flip heuristic repeatedly tests whether the diagonal of a convex quadrilateral in the triangulation can be flipped. It will be flipped if two conditions hold simultaneously: (i) The two new triangles are order- k Delaunay triangles. (ii) The new edge connects the lowest point of the four to the opposite point. A flip does not necessarily remove a local minimum, but cannot create one, and it can make possible that a later flip removes a local minimum.

Our algorithm to perform the flips starts with the Delaunay triangulation and $k' = 1$, then does all flips possible to obtain an order- k' Delaunay triangulation, then increments k' and repeats. This continues until $k' = k$.

We first analyze the maximum number of flips possible, and then we discuss the efficiency of the heuristic.

Lemma 1. *If an edge \overline{ab} is in the triangulation, then the flip heuristic will never have an edge \overline{cd} later with $\min(c, d) \geq \min(a, b)$ that intersects \overline{ab} .*

An immediate consequence of the lemma above is that an edge that is flipped out of the triangulation cannot reappear. There are at most $O(nk)$ pairs of points in a point set of n points that give order- k Delaunay edges [7].

Lemma 2. *The flip heuristic to reduce the number of local minima performs at most $O(nk)$ flips.*

To implement the flip heuristic efficiently, we maintain the set of all convex quadrilaterals in the current triangulation, with the order of the two triangles that would be created if the diagonal were flipped. The order of a triangle is the number of points in the circumcircle of the vertices of the triangle. Whenever a flip is done, we update the set of convex quadrilaterals. At most four are deleted and at most four new ones are created by the flip. We can find the order of the incident triangles by circular range counting queries. Since we are only interested in the count if the number of points in the circle is at most k , we implement circular range counting queries by point location in the order- $(k + 1)$ Voronoi diagram [14], taking $O(\log n + k)$ time per query after $O(nk \log n)$ preprocessing time. We conclude:

Theorem 2. *The flip heuristic to reduce the number of local minima in order- k Delaunay triangulations on n points takes $O(nk^2 + nk \log n)$ time.*

2.3 The Hull Heuristic

The second heuristic for reducing the number of local minima is the hull heuristic. It was described by Gudmundsson et al. [7], and has an approximation factor of $\Theta(k^2)$ of the optimum. The hull heuristic adds a useful order- k Delaunay edge e if it reduces the number of local minima. This edge may intersect several Delaunay edges, which are removed; the two holes in the triangulation that appear are retriangulated with the constrained Delaunay triangulation [4] in $O(k \log k)$ time. These two polygonal holes are called the hull of this higher-order Delaunay edge e . The boundary of the hull consists of Delaunay edges only. No other higher-order Delaunay edges will be used that intersects this hull. This is needed to guarantee that the final triangulation is order- k . It is known that two useful order- k Delaunay edges used together can give an order- $(2k - 2)$ Delaunay triangulation [8], which is higher than allowed. Here we give a slightly different implementation than in [7]. It is more efficient for larger values of k .

Assume that a point set P and an order value k are given. We first compute the Delaunay triangulation T of P , and then compute the set E of all useful order- k Delaunay edges, as in [7], in $O(nk \log n + nk^2)$ time. There are $O(nk)$ edges in E , and for each we have the lowest order $k' \leq k$ for which it is a useful order- k' Delaunay edge.

Next we determine the subset $P' \subseteq P$ of points that are a local minimum in the Delaunay triangulation. Then we determine the subset $E' \subseteq E$ of edges that connect a point of P' to a lower point. These steps trivially take $O(nk)$ time.

Sort the edges of E' by non-decreasing order. For every edge $e \in E'$, traverse T to determine the edges of T that intersect e . If any one of them is not a Delaunay edge or is a marked Delaunay edge, then we stop and continue with the next edge of E' . Otherwise, we remove all intersected Delaunay edges and mark all Delaunay edges of the polygonal hole that appears. Then we insert e and retriangulate the hull, the two polygons to the two sides of e , using the

Table 1. Results of the flip/hull heuristic for orders 0–8

	0	1	2	3	4	5	6	7	8
Calif. Hot Springs	47/47	43/43	33/31	29/26	25/20	24/19	23/18	21/18	18/16
Wren Peak	45/45	37/37	31/31	27/27	24/22	23/21	21/20	19/20	19/20
Quinn Peak	53/53	44/44	36/36	31/29	26/25	24/23	23/21	21/20	20/19
Sphinx Lakes	33/33	27/27	22/22	20/19	19/18	17/16	15/12	12/9	11/9

Delaunay triangulation constrained to the polygons. We also mark these edges. Finally, we remove edges from E' : If the inserted edge e made that a point $p \in P$ is no longer a local minimum, then we remove all other edges from E' where p is the highest endpoint.

Due to the marking of edges, no edge $e \in E'$ will be inserted if it intersects the hull of a previously inserted edge of E' . Every edge of E' that is not used in the final triangulation is treated in $O(\log n + k)$ time, and every edge of E' that is used in the final triangulation is treated in $O(\log n + k \log k)$ time.

Theorem 3. *The hull heuristic to reduce the number of local minima in order- k Delaunay triangulations on n points takes $O(nk^2 + nk \log n)$ time.*

The full paper [5] shows examples where the flip and hull heuristics may not give the minimum number of local minima, even for order 2.

2.4 Experiments

Table 1 shows the number of local minima obtained after applying the flip and hull heuristics to four different terrains. The terrains roughly have 1800 vertices. The vertices were chosen by random sampling 1% of the points from elevation grids. Adjacent vertices with the same elevation required special attention.

The values in the table show that higher-order Delaunay triangulations indeed can give significantly fewer local minima than the standard Delaunay triangulation (order-0). This effect is already clear at low orders, indicating that many local minima of Delaunay triangulations may be caused by having chosen the wrong edges for the terrain (interpolation).

The difference in local minima between the flip and hull heuristics shows that the hull heuristic usually is better, but there are some exceptions.

To test how good the results are, we also tested how many local minima of each terrain cannot be removed simply because there is no useful order- k Delaunay edge to a lower point. It turned out that the hull heuristic found an optimal order- k Delaunay triangulation in all cases except for five, where one local minimum too many remained. In four of these cases the flip heuristic found an optimal order- k Delaunay triangulation. In one case (Wren, order-6) it is not clear; a triangulation with 19 local minima may exist.

3 Minimizing the Number of Valley Edge Components

In a triangulation representing a terrain, there are three types of edges: ridge or diffluent edges, normal or transfluent edges, and valley or confluent edges [6,12,18].

These edges are used to delineate drainage basins and other hydrological characteristics of terrains. Flow on terrains is usually assumed to take the direction of steepest descent. This is a common assumption used in drainage network modeling [16,18]. Assuming no degeneracies (including horizontal triangles), every point on the terrain has a unique direction of steepest descent, except local minima. Hence, a flow path downward in the terrain can be defined for any point. The direction of steepest descent at a vertex can be over an edge or over the interior of a triangle. Flow on a triangle is always normal to the contour lines on that triangle.

Ridge edges are edges that do not receive flow from any point on the terrain. The incident triangles drain in a direction away from this edge. Valley edges are edges that receive flow from (part of) both incident triangles; they would be ridge edges if the terrain were upside down. Normal edges receive flow from (part of) one incident triangle and drain to the other. Valley edges can be used to define the drainage network, and ridge edges can be used for the drainage basins, also called catchment areas [12]. Many more results on drainage in GIS exist; it is beyond the scope of this paper to review it further.

Just like local minima in a triangulation for a terrain are often artifacts, so are isolated valley edges, and sequences of valley edges that do not end in a local minimum. In the latter case, flow would continue over the middle of a triangle, which usually does not correspond to the situation in real terrains. If channeled water carves a valley-like shape in a terrain, then the valley does not suddenly stop, because the water will stay channeled. This is true unless the surface material changes, or the terrain becomes nearly flat [13]. Besides being unrealistic, isolated valley edges may influence the shape of drainage basins [12].

We define a *valley (edge) component* to be a maximal set of valley edges such that flow from all of these valley edges reaches the lowest vertex incident to these valley edges. A valley component necessarily is a rooted tree with a single target that may be a local minimum. Figure 2 shows an example of a terrain with three valley components; the valley edges are shown by the direction of flow, numbers indicate the identity of each component, and squares show local minima. In this example, the direction of steepest descent from the vertex where components 1 and 2 touch is over the edge labeled 1 to the boundary. Component 3 ends in a vertex that is not a local minimum; flow proceeds over a triangle.

By the discussion above, the drainage quality of a terrain is determined by the number of local minima, and the number of valley edge components that

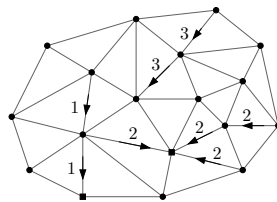


Fig. 2. Example of a terrain with three valley edge components

Table 2. Statistics for four terrains. For each terrain, counts for the Delaunay triangulation are given, and for the outcome of the flip and hull heuristics for order 8. The last row gives the number of valley edge components that do not end in a local minimum. Numbers between brackets are the additional numbers for the terrain boundary.

	Quinn	flip-8	hull-8	Wren Peak	flip-8	hull-8
Edges	5210	5210	5210	5185	5185	5185
Valley edges	753	862	684	799	860	761
Local minima	53 (12)	20 (9)	19 (6)	45 (17)	19 (12)	19 (17)
Valley components	240	289	224	259	270	247
Not min. ending	173 (8)	246 (15)	191 (8)	185 (18)	218 (23)	199 (18)

	Sphinx	flip-8	hull-8	Hot Springs	flip-8	hull-8
Edges	5179	5179	5179	5234	5234	5234
Valley edges	675	830	627	853	964	807
Local minima	33 (16)	11 (11)	9 (16)	47 (20)	18 (16)	16 (20)
Valley components	261	313	244	249	256	231
Not min. ending	213 (5)	285 (9)	218 (7)	179 (11)	210 (17)	190 (13)

do not end in a local minimum. The sum of these two numbers immediately gives the total number of valley edge components. We will attempt to reduce this number with a new heuristic called the valley heuristic. But first we analyze how many valley edges and valley components appear in the results of the flip and hull heuristics.

3.1 Consequences of the Flip and Hull Heuristics on the Valleys

The flip and hull heuristics can remove local minima of triangulations, and therefore they seem more realistic as terrains. However, the heuristics may create valley edges, including isolated ones. This is especially true for the flip heuristic. We examined the triangulations obtained from the experiments of Subsection 2.4 and analyzed the number of valley edges and valley components.

Table 2 shows statistics on four terrains. Local minima on the boundary are not counted, but their number is shown separately in brackets. The same is true for valley components that end on the boundary of the terrain, but not in a local minimum. Note that local minima on the boundary may not have any valley component ending in it. We can see that the flip heuristic has increased the number of valley edges considerably, whereas the hull heuristic decreased this number. The same is true for the number of valley components. Another observation from the table is that there are many valley edge components. The average size is in all cases between 2 and 4 edges. This shows the need for further processing of the triangulation, or for a different heuristic.

3.2 The Valley Heuristic

We apply two methods to improve the drainage quality of the terrain. Firstly, isolated valley edges can sometimes be removed by a single flip or useful order- k edge insertion, reducing the number of valley components. Secondly, if a valley

edge has a lower endpoint whose direction of steepest descent goes over a triangle, then the valley component can sometimes be extended downhill and possibly be connected to another valley component, which reduces the number of valley components. We observe:

Observation 1. (i) For a convex quadrilateral in a terrain, at most one diagonal is a valley edge. (ii) If in a triangle in a terrain, two edges are valley edges, then their common vertex is not the highest vertex of that triangle. (iii) A flip in a convex quadrilateral affects at most five edges (for being valley or not).

To remove an isolated valley edge, five candidate flips can take care of this: the valley edge itself, and the four other edges incident to the two triangles incident to the valley edge. A flip can potentially remove one isolated valley edge but create another one at the same time; such a flip is not useful and termination of the heuristic would not be guaranteed. Any flip changes the flow situation at four vertices. There are many choices possible when to allow the flip and when not. We choose to flip only if the flow situation of the four vertices of the convex quadrilateral does not change, except for the removal of the isolated valley edge, and the two new triangles are order- k Delaunay. In particular, a flip may not create new valley edges. It is undesirable to change any valley component in an unintended way. Algorithmically, identifying isolated valley edges and flipping them, if possible, can be done in $O(nk \log n)$ time.

To extend a valley component downward, we take its lowest endpoint v and change the triangulation locally to create the situation that v has a direction of steepest descent over a valley edge to a lower vertex. We do this regardless of the situation that v is a local minimum, or v has its direction of steepest descent over the interior of some triangle. We only do this with flips. For every triangle incident to v , we test if it is part of a convex quadrilateral $vpwq$, and if so, we test if the flip of \overline{pq} to \overline{vw} yields two order- k Delaunay triangles, \overline{vw} is a valley edge, w is lowest of $vpwq$, the steepest descent from v is to w , and no valley components are interrupted at p or q .

Throughout the algorithm, any vertex can be lowest point of a valley component at most twice. First as the lower endpoint of an isolated valley edge, and once more by extending valley components. Once a vertex is in a valley component with more than two edges, it will stay in such a component. Hence, there will only be $O(n)$ flips. Using point location in the order- $(k + 1)$ Voronoi diagram, we conclude:

Theorem 4. *The valley heuristic to reduce the number of valley components in order- k Delaunay triangulations on n points takes $O(nk \log n)$ time.*

3.3 Experiments

The two ways of reducing the number of valley components were applied to the Delaunay triangulation, and to the outcomes of the flip and hull heuristics. We show the results in Table 3 for order 8 only. In fact, the table shows the outcome of applying the valley heuristic (with order 8) to all outcomes of Table 2.

Table 3. Statistics for four terrains when applying the valley heuristic (order 8) to the Delaunay triangulation and the order-8 outcome of the flip and hull heuristics

	Quinn	flip-8 +v	hull-8 +v	Wren Peak	flip-8 +v	hull-8 +v
Edges	5210	5210	5210	5185	5185	5185
Valley edges	686	762	641	743	798	712
Local minima	35 (12)	20 (10)	19 (12)	31 (16)	19 (12)	19 (16)
Valley components	147	189	144	167	208	169
Not min. ending	102 (5)	148 (13)	115 (5)	112 (16)	161 (19)	126 (16)

	Sphinx	flip-8 +v	hull-8 +v	Hot Springs	flip-8 +v	hull-8 +v
Edges	5179	5179	5179	5234	5234	5234
Valley edges	597	729	565	790	895	759
Local minima	20 (16)	11 (11)	9 (16)	28 (19)	18 (16)	16 (19)
Valley components	157	212	155	169	187	161
Not min. ending	125 (5)	191 (4)	133 (6)	118 (13)	148 (11)	123 (12)

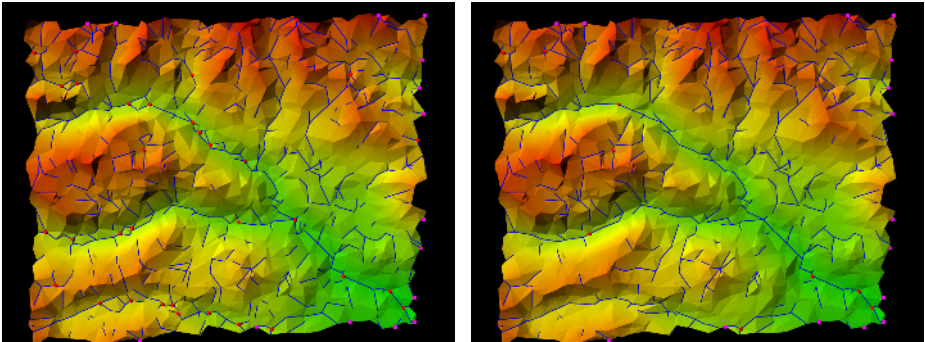


Fig. 3. Visualization of the valley edges and local minima of the Sphinx data set. Left, the Delaunay triangulation, and right outcome of hull-8 followed by the valley heuristic.

We observe that the valley heuristic succeeds in reducing the number of valley components considerably in all cases. The reduction is between 20% and 40% for all triangulations. There is no significant difference in reduction between the three types of triangulations. The valley heuristic by itself also reduces the number of local minima, as can be expected. The number of valley components is lowest when applying the valley heuristic to the outcome of the hull heuristic, and sometimes when applying the valley heuristic directly to the Delaunay triangulation. Further examination of the outcome shows that the largest reduction in the number of valley components comes from the removal of isolated valley edges. This accounts for roughly 60% to 80% of the reduction.

One terrain (Delaunay triangulation) and the outcome after the valley heuristic applied to the outcome of hull-8 is shown in Figure 3 for the Sphinx data set. Images of the outcomes of the other heuristics and for other data sets are given in the full paper [5].

4 Conclusions and Further Research

We examined the computation of triangulations for realistic terrains using higher-order Delaunay triangulations. The realistic aspect is motivated by hydrologic and other flow applications, perhaps the most important reason for terrain modelling. Realistic terrains have few local minima and few valley edge components.

Theoretically, we showed that triangulating with the minimum number of local minima is NP-hard due to alignment of points. For order- k Delaunay triangulations we obtain the same result in non-degenerate cases for $k = \Omega(n^\epsilon)$ and $k \leq c \cdot n$. The case of constant orders but at least 2 remains open.

We presented two heuristics (one new, one old) to remove local minima, analyzed their efficiency, and implemented them. It turns out that higher-order Delaunay triangulations exist with considerably fewer local minima for low orders already, and the hull heuristic is better at computing them. We tested orders 0 up to 8. Often we obtain an optimal order- k Delaunay triangulation. The hull heuristic creates fewer valley edges and valley edge components than the flip heuristic. We also presented the valley heuristic to reduce the number of valley edge components. The experiments and images suggest that the valley heuristic applied to the outcome of the hull heuristic gives the best results.

It is possible to devise a valley heuristic that inserts useful order- k edges, similar to the hull heuristic, but now to reduce the number of valley components. Furthermore, it is possible to integrate minimizing local minima and reducing valley components in one heuristic. We leave this for future work.

It would also be interesting to extend the research to computing high quality drainage basins, or a good basin hierarchy. However, it is not clear how this quality should be defined, nor how it should be combined with local minima and valley components used in this paper. Finally, other criteria than higher-order Delaunay triangulations can be used to guarantee a good shape of the triangles. Again we leave this for future research.

References

1. P.K. Agarwal and N.H. Mustafa. Independent set of intersection graphs of convex objects in 2D. In *Proc. SWAT 2004*, number 3111 in LNCS, pages 127–137, Berlin, 2004. Springer.
2. M. Bern, H. Edelsbrunner, D. Eppstein, S. Mitchell, and T. S. Tan. Edge insertion for optimal triangulations. *Discrete Comput. Geom.*, 10(1):47–65, 1993.
3. M. Bern and D. Eppstein. Mesh generation and optimal triangulation. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, volume 4 of *Lecture Notes Series on Computing*, pages 47–123. World Scientific, Singapore, 2nd edition, 1995.
4. L. P. Chew. Constrained Delaunay triangulations. *Algorithmica*, 4:97–108, 1989.
5. T. de Kok, M. van Kreveld, and M. Löffler. Generating realistic terrains with higher-order Delaunay triangulations. Technical Report CS-UU-2005-020, Institute of Information and Computing Sciences, 2005.

6. A.U. Frank, B. Palmer, and V.B. Robinson. Formal methods for the accurate definition of some fundamental terms in physical geography. In *Proc. 2nd Int. Symp. on Spatial Data Handling*, pages 583–599, 1986.
7. J. Gudmundsson, M. Hammar, and M. van Kreveld. Higher order Delaunay triangulations. *Comput. Geom. Theory Appl.*, 23:85–98, 2002.
8. J. Gudmundsson, H. Haverkort, and M. van Kreveld. Constrained higher order Delaunay triangulations. *Comput. Geom. Theory Appl.*, 30:271–277, 2005.
9. M.F. Hutchinson. Calculation of hydrologically sound digital elevation models. In *Proc. 3th Int. Symp. on Spatial Data Handling*, pages 117–133, 1988.
10. S.K. Jenson and C.M. Trautwein. Methods and applications in surface depression analysis. In *Proc. Auto-Carto 8*, pages 137–144, 1987.
11. Y. Liu and J. Snoeyink. Flooding triangulated terrain. In P.F. Fisher, editor, *Developments in Spatial Data Handling, proc. 11th Int. Sympos.*, pages 137–148, Berlin, 2004. Springer.
12. M. McAllister and J. Snoeyink. Extracting consistent watersheds from digital river and elevation data. In *Proc. ASPRS/ACSM Annu. Conf.*, 1999.
13. C. Mitchell. *Terrain Evaluation*. Longman, Harlow, 2nd edition, 1991.
14. E.A. Ramos. On range reporting, ray shooting and k -level construction. In *Proc. 15th Annu. ACM Symp. on Computational Geometry*, pages 390–399, 1999.
15. B. Schneider. Geomorphologically sound reconstruction of digital terrain surfaces from contours. In T.K. Poiker and N. Chrisman, editors, *Proc. 8th Int. Symp. on Spatial Data Handling*, pages 657–667, 1998.
16. D.M. Theobald and M.F. Goodchild. Artifacts of TIN-based surface flow modelling. In *Proc. GIS/LIS*, pages 955–964, 1990.
17. G.E. Tucker, S.T. Lancaster, N.M. Gasparini, R.L. Bras, and S.M. Rybarczyk. An object-oriented framework for distributed hydrologic and geomorphic modeling using triangulated irregular networks. *Computers and Geosciences*, 27:959–973, 2001.
18. S. Yu, M. van Kreveld, and J. Snoeyink. Drainage queries in TINs: from local to global and back again. In M.J. Kraak and M. Molenaar, editors, *Advances in GIS research II: Proc. of the 7th Int. Symp. on Spatial Data Handling*, pages 829–842, 1997.

I/O-Efficient Construction of Constrained Delaunay Triangulations

Pankaj K. Agarwal^{1,*}, Lars Arge^{1,2,**}, and Ke Yi^{1,***}

¹ Department of Computer Science, Duke University, Durham, NC 27708, USA
{`pankaj`, `large`, `yike`}@`cs.duke.edu`

² Department of Computer Science, University of Aarhus, Aarhus, Denmark
`large@daimi.au.dk`

Abstract. In this paper, we designed and implemented an I/O-efficient algorithm for constructing constrained Delaunay triangulations. If the number of constraining segments is smaller than the memory size, our algorithm runs in expected $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os for triangulating N points in the plane, where M is the memory size and B is the disk block size. If there are more constraining segments, the theoretical bound does not hold, but in practice the performance of our algorithm degrades gracefully. Through an extensive set of experiments with both synthetic and real data, we show that our algorithm is significantly faster than existing implementations.

1 Introduction

With the emergence of new terrain mapping technologies such as Laser altimetry (LIDAR), one can acquire millions of georeferenced points within minutes to hours. Converting this data into a digital elevation model (DEM) of the underlying terrain in an efficient manner is a challenging important problem. The so-called triangulated irregular network (TIN) is a widely used DEM, in which a terrain is represented as a triangulated xy -monotone surface. One of the popular methods to generate a TIN from elevation data—a cloud of points in \mathbb{R}^3 —is to project the points onto the xy -plane, compute the Delaunay triangulation of the projected points, and then lift the Delaunay triangulation back to \mathbb{R}^3 . However, in addition to the elevation data one often also has data representing various linear features on the terrain, such as river and road networks, in which case one would like to construct a TIN that is consistent with this data, that is,

* Supported in part by NSF under grants CCR-00-86013, EIA-01-31905, CCR-02-04118, and DEB-04-25465, by ARO grants W911NF-04-1-0278 and DAAD19-03-1-0352, and by a grant from the U.S.–Israel Binational Science Foundation.

** Supported in part by the US NSF under grants CCR-9984099, EIA-0112849, and INT-0129182, by ARO grant W911NF-04-1-0278, and by an Ole Rømer Scholarship from the Danish National Science Research Council.

*** Supported by NSF under grants CCR-02-04118, CCR-9984099, EIA-0112849, and by ARO grant W911NF-04-1-0278.

where the linear features appear along the edges of the TIN. In such cases it is desirable to compute the so-called *constrained Delaunay Triangulation (CDT)* of the projected point set with respect to the projection of the linear features. Roughly speaking, the constrained Delaunay triangulation of a point set P and a segment set S is the triangulation that is as close to the Delaunay triangulation of P under the constraint that all segments of S appear as edges of the triangulation.

The datasets being generated by new mapping technologies are too large to fit in internal memory and are stored in secondary memory such as disks. Traditional algorithms, which optimize the CPU efficiency under the RAM model of computation, do not scale well with such large amounts of data. This has led to growing interest in designing I/O-efficient algorithms that optimize the data transfer between disk and internal memory. In this paper we study I/O-efficient algorithms for planar constrained Delaunay triangulations.

Problem Statement. Let P be a set of N points in \mathbb{R}^2 , and let S be a set of K line segments with pairwise-disjoint interiors whose endpoints are points in P . The points $p, q \in \mathbb{R}^2$ are *visible* if the interior of the segment pq does not intersect any segment of S . The *constrained Delaunay triangulation* $CDT(P, S)$ is the triangulation of S that consists of all segments of S , as well as all edges connecting pairs of points $p, q \in P$ that are visible and that lie on the boundary of an open disk containing only points of P that are not visible from both p and q . $CDT(P, \emptyset)$ is the Delaunay triangulation of the point set P . Refer to Figure 1. For clarity, we use *segments* to refer to the “obstacles” in S , and reserve the term “edges” for the other edges in the triangulation $CDT(P, S)$.

We work in the standard external memory model [2]. In this model, the main memory holds M elements and each disk access (or I/O) transmits a block of B elements between main memory and continuous locations on disk. The complexity of an algorithm is measured in the total number of I/Os performed, while the internal computation cost is ignored.

Related Results. Delaunay triangulation is one of the most widely studied problems in computational geometry; see [5] for a comprehensive survey. Several worst-case efficient $O(N \log N)$ algorithms are known in the RAM model, which

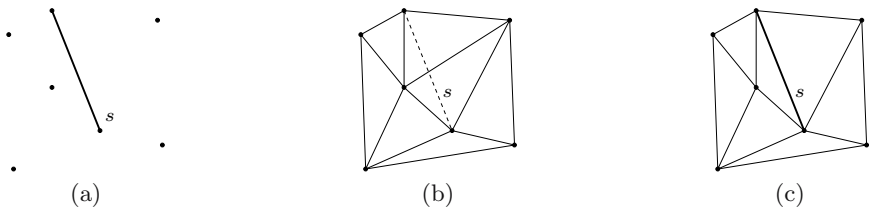


Fig. 1. (a) The point set P of 7 points and segment set S of 1 segment s . (b) $DT(P) = CDT(P, \emptyset)$. (c) $CDT(P, S)$.

are based on different standard paradigms, such as divide-and-conquer and sweep-line. A randomized incremental algorithm with $O(N \log N)$ expected running time was proposed in [12]. By now efficient implementations of many of the developed algorithms are also available. For example, the widely used software package `triangle`, developed by Shewchuk [16], has implementations of all three algorithms mentioned above. Both CGAL [7] and LEDA [14] software libraries also offer Delaunay triangulation implementations.

By modifying some of the algorithms for Delaunay triangulation, $O(N \log N)$ time RAM-model algorithms have been developed for constrained Delaunay triangulations [8,15]. However, these algorithms are rather complicated and do not perform well in practice. A common practical approach for computing $\text{CDT}(P, S)$, e.g. used by `triangle` [16], is to first compute $\text{DT}(P)$ and then add the segments of S one by one and update the triangulation.

Although I/O-efficient algorithms have been designed for Delaunay triangulations [10,11,13], no I/O-efficient algorithm is known for the constrained case.

Our Results. By modifying the algorithm of Crauser et al. [10] we develop the first I/O-efficient constrained Delaunay triangulation algorithm. It uses $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os expected, provided that $|S| \leq c_0 M$, where c_0 is a constant. Although our algorithm falls short of the desired goal of having an algorithm that performs $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os irrespective of the size of S , it is useful for many practical situations. We demonstrate the efficiency and scalability of our algorithm through an extensive experimental study with both synthetic and real-life data. Compared with existing constrained Delaunay triangulation packages, our algorithm is significantly faster on large datasets. For example it can process 10GB of real-life LIDAR data using only 128MB of main memory in roughly 7.5 hours! As far as we know, this is the first implementation of constrained Delaunay triangulation algorithm that is able to process such a large dataset. Moreover, even when S is larger than the size of main memory, our algorithm does not fail, but its performance degrades quite gracefully.

2 I/O-Efficient Algorithm

Let P be a set of N points in \mathbb{R}^2 , and let S be a set of K segments with pairwise-disjoint interiors whose endpoints lie in P . Let E be the set of endpoints of segments in S . We assume that points of P are in general position. For simplicity of presentation, we include a point p_∞ at infinity in P . We also add p_∞ to E . Below we describe an algorithm for constructing $\text{CDT}(P, S)$ that follows the framework of Crauser et al. [10] for constructing Delaunay triangulations. However, we first introduce the notion of extended Voronoi diagrams, originally proposed by Seidel [15], and define conflict lists and kernels.

Extended Voronoi Diagrams. We extend the plane to a more complicated surface as described by Seidel [15]. Imagine the plane as a sheet of paper Σ with the points of P and the segments of S drawn on it. Along each segment $s \in S$ we “glue” an additional sheet of paper Σ_s , which is also a two-dimensional plane,

onto Σ ; the sheets are glued only at s . These $K + 1$ sheets together form a surface Σ_S . We call Σ the *primary* sheet, and the other sheets *secondary* sheets. P “lives” only on the primary sheet Σ , and a segment $s \in S$ “lives” in the primary sheet Σ and the secondary sheet Σ_s . For a secondary sheet Σ_s , we define its *outer region* to be the set of points that do not lie in the strip bounded by the two lines normal to s and passing through the endpoints of s .

Assume the following connectivity on Σ_S : When “traveling” in Σ_S , whenever we cross a segment $s \in S$ we must switch sheet, i.e., when traveling in a secondary sheet Σ_s and reaching the segment s we must switch to the primary sheet Σ , and vice versa. We can define a visibility relation using this switching rule. Roughly speaking, two points $x, y \in \Sigma_S$ are *visible* if we can draw a line segment from x to y on Σ_S following the above switching rule. More precisely, x and y are visible if: $x, y \in \Sigma$ and the segment xy does not intersect any segment of S ; $x, y \in \Sigma_s$ and the segment xy does not intersect s ; $x \in \Sigma, y \in \Sigma_s$ and the segment xy crosses s but no other segment; or $x \in \Sigma_s, y \in \Sigma_t$, and the segment xy crosses s and t but no other segment. For $x, y \in \Sigma_S$, we define the distance $d(x, y)$ between x and y to be the length of the segment connecting them if they are visible, and $d(x, y) = \infty$ otherwise.

For $p, q, r \in \Sigma_S$, if there is a point $y \in \Sigma_S$ so that $d(p, y) = d(q, y) = d(r, y)$, then we define the *circumcircle* $C(p, q, r; S) = \{x \in \Sigma_S \mid d(x, y) = d(p, y)\}$. Otherwise $C(p, q, r; S)$ is undefined. Note that portions of $C(p, q, r; S)$ may lie on different sheets of Σ_S . We define $D(p, q, r; S)$ to be the open disk bounded by $C(p, q, r; S)$, i.e., $D(p, q, r; S) = \{x \in \Sigma_S \mid d(x, y) < d(p, y)\}$. Refer to Figure 2(a). Using the circumcircle definition, the constrained Delaunay triangulation can be defined in the same way as standard Delaunay triangulations, i.e., $CDT(P, S)$ consists of all triangles $\Delta uvw, u, v, w \in P$, whose circumcircles do not enclose any point of P . We define the *extended Voronoi region* of a point $p \in P$ as $EV(p, S) = \{x \in \Sigma_S \mid d(x, p) \leq d(x, q), \forall q \in P\}$, and the *extended Voronoi diagram* of P (with respect to S) as $EVD(P, S) = \{EV(p, S) \mid p \in P\}$. Seidel [15] showed that $CDT(P, S)$ is the dual of $EVD(P, S)$, in the sense that an edge pq appears in $CDT(P, S)$ if and only if $EV(p, S)$ and $EV(q, S)$ share

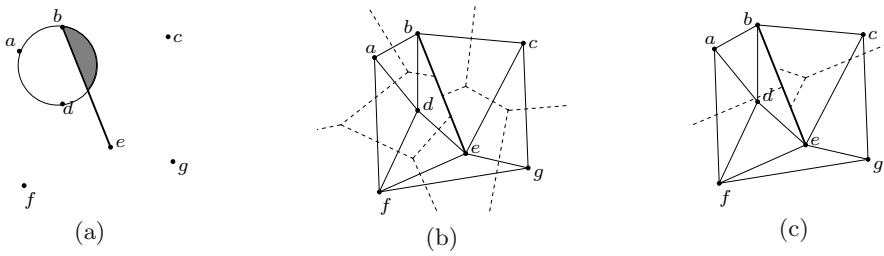


Fig. 2. (a) For the point set of Figure 1(a), a portion of $D(a, b, d; S)$ lies in the primary sheet (unshaded), the other portion lies in the secondary sheet Σ_{be} (shaded). (b) $CDT(P, S)$ (solid lines) and the portion of $EVD(P, S)$ (dashed lines) that lies in the primary sheet. (c) The portion of $EVD(P, S)$ (dashed lines) that lies in the secondary sheet Σ_{be} .

an edge. Refer to Figure 2(b) and 2(c). This duality relation will be useful in extending the algorithm by Crauser et al. [10] to computing $\text{CDT}(P, S)$.

Conflict Lists and Kernels. Let $R \subseteq P$ be a subset of points such that $E \subseteq R$. Let $e = pq$ be an edge of $\text{CDT}(R, S)$, and let Δpqu and Δpqv be the two triangles adjacent to e . (Since $p_\infty \in R$, each edge is adjacent to two triangles.) We define the *conflict list* [9] of e , denoted by $P|_e \subseteq P$, as the set of points of P that lie in $D(p, q, u; S) \cup D(p, q, v; S)$. If there exists a point $p' \in P|_e$, then at least one of Δpqu and Δpqv does not appear in $\text{CDT}(R \cup \{p'\}, S)$.

One basic step in our algorithm will be to compute a triangulation of each $P|_e$ and then merge the results together to form $\text{CDT}(P, S)$. Let $I_e = \{e\}$ if $e \in S$, and \emptyset otherwise. Then the triangulation we will compute for $P|_e$ is $\text{CDT}(P|_e, I_e)$. In order to identify the triangles of $\text{CDT}(P|_e, I_e)$ that appear in $\text{CDT}(P, S)$, we define the notion of the *kernel* of e , denoted by $\tau(e)$, which is contained in $\text{EV}(p, S) \cup \text{EV}(q, S)$. A point $x \in \text{EV}(p, S)$ (resp. $x \in \text{EV}(q, S)$) lies in $\tau(e)$ if the ray \overrightarrow{px} (resp. \overrightarrow{qx}) intersects the common edge between $\text{EV}(p, S)$ and $\text{EV}(q, S)$. Refer to Figure 3. Note that the kernel of an edge e can be determined with knowing only e and its two adjacent triangles in $\text{CDT}(R, S)$.

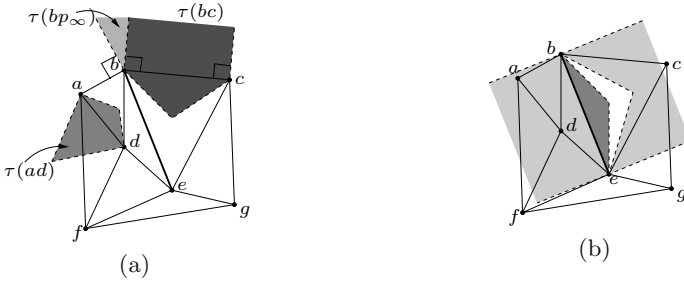


Fig. 3. (a) The kernels of edges ad, bc , and bp_∞ . (b) The kernel of the edge be ; the darker part lies in the primary sheet, and the lighter part lies in the secondary sheet Σ_{be} .

The following properties of conflict lists and kernels, whose proofs are omitted from this abstract, lead to a recursive algorithm for computing $\text{CDT}(P, S)$.

- (i) The interiors of $\tau(e), e \in \text{CDT}(R, S)$ are pairwise disjoint.
- (ii) $\{\tau(e) \mid e \in \text{CDT}(R, S)\}$ covers the points of Σ_S that do not lie in an outer region.
- (iii) Let $E \subseteq R \subseteq P$. For any edge $e \in \text{CDT}(R, S)$ and for any $u, v, w \in P|_e$ such that $C(u, v, w; S)$ is defined with ξ being the center, if $\xi \in \tau(e)$ and $D(u, v, w; I_e) \cap P|_e = \emptyset$, then $D(u, v, w; S) \cap P = \emptyset$.
- (iv) Let $E \subseteq R \subseteq P$. For any $\Delta uvw \in \text{CDT}(P, S)$ with circumcenter ξ , if e is the edge of $\text{CDT}(R, S)$ such that $\xi \in \tau(e)$, then $u, v, w \in P|_e$ and $D(u, v, w; I_e) \cap P|_e = \emptyset$.

These properties imply that if we have computed $\text{CDT}(R, S)$, we can compute $\text{CDT}(P, S)$ by repeating the following steps for each $e \in \text{CDT}(R, S)$: Compute $\text{CDT}(P|_e, I_e)$ and report a triangle $\Delta uvw \in \text{CDT}(P|_e, I_e)$ if the center of $C(u, v, w; I_e)$ lies inside $\tau(e)$. Below we describe how to do this efficiently.

Our Algorithm. As mentioned, the overall structure of our algorithm is the same as that of the algorithm of Crauser et al. [10]. We call a subset $R \subseteq P$ a p -sample if R is obtained by choosing each point of P with probability p . We choose a sequence of subsets of P , called a *gradation*:

$$P_1 \subseteq P_2 \subseteq \dots \subseteq P_l = P,$$

where $E \subseteq P_1$ and $P_i \setminus E$ is a (B/M) -sample of $P_{i+1} \setminus E$. P_1 is small enough so that $\text{CDT}(P_1, S)$ can be computed in main memory.

Initially, our algorithm constructs $\text{CDT}(P_1, S)$ using an internal memory algorithm. Then we scan P and for each point $p \in P \setminus P_1$ determine the edges of $\text{CDT}(P_1, S)$ that it is in conflict with; for each such edge e , we generate an (e, p) pair. In the end we sort these pairs to create the conflict lists for all the edges of $\text{CDT}(P_1, S)$.

Next, we proceed in $l - 1$ rounds. In the i -th round, we are given $\text{CDT}(P_i, S)$ and the conflict lists for all the edges of $\text{CDT}(P_i, S)$, and construct $\text{CDT}(P_{i+1}, S)$ and the conflict lists for the edges of $\text{CDT}(P_{i+1}, S)$ (the conflict lists need not be generated for the last round). This is accomplished by the following steps.

1. For each edge e of $\mathcal{T}_i = \text{CDT}(P_i, S)$, we scan its conflict list and determine $P_{i+1|e}$.
2. We consider each $P_{i+1|e}$ in turn:
 - 2.1 Let $t_e = \lceil |P_{i+1|e}| / c_1(M/B) \rceil$. We first take a $1/(c_2 t_e \log t_e)$ -sample Y_e of $P_{i+1|e}$; we add the four vertices of the two adjacent triangles of e if they are not chosen in the sample. Then we compute $\mathcal{T}_e = \text{CDT}(Y_e, I_e)$ using an internal memory algorithm. Next for each edge e' of \mathcal{T}_e we determine $(P_{i+1|e})|_{e'}$ by scanning $P_{i+1|e}$ on disk. If for any e we have $|(P_{i+1|e})|_{e'}| > c_1 M/B$, we repeat this step by taking a new sample Y_e .
 - 2.2 For each edge e' of \mathcal{T}_e , we load $(P_{i+1|e})|_{e'}$ into memory and compute $\mathcal{T}_{e'} = \text{CDT}((P_{i+1|e})|_{e'}, I_{e'})$. We report only the triangles of $\mathcal{T}_{e'}$ that have their circumcircles centered inside $\tau(e) \cap \tau(e')$. We then scan $P|_e$ to build the conflict lists for these triangles (unless this is the last round). We do so by allocating one main memory block for each of the $O(\frac{M}{B})$ triangles and writing points to the relevant block as they are processed; when a block is full it is written to disk.
3. After all edges of $\text{CDT}(P_i, S)$ have been processed, $\mathcal{T}_{i+1} = \text{CDT}(P_{i+1}, S)$ is simply all the triangles reported in Step 3. The conflict list for an edge of $\text{CDT}(P_{i+1}, S)$ is simply the union of the conflict lists of its two adjacent triangles.

Analysis of I/O. We wish to follow the analysis of Crauser et al. [10] that is based on the bounds on the expected size of the conflict lists and their higher

moments [9]. However, unlike [10], P_i is not a completely random sample of P_{i+1} in our case, which makes the analysis more complicated. Nevertheless, we can prove similar bounds on the expected size of conflict lists. The following lemma summarizes the main technical result, whose proof is given in the full version of the paper.

Lemma 1. *Let R be a p -sample of $P \setminus E$. For any constant integer $c \geq 1$,*

$$E \left[\sum_{e \in \text{CDT}(R \cup E, S)} |P_{|e|^c} \right] = O \left(\frac{|R \cup E|}{p^c} \right).$$

In our algorithm, $P_i \setminus E$ is a p_i -sample of $P \setminus E$, therefore,

$$E \left[\sum_{e \in \mathcal{T}_i} |P_{|e|^c} \right] = O \left(\frac{|E| + |P_i \setminus E|}{p_i^c} \right) = O \left(\frac{|E|}{p_i^c} + \frac{|P \setminus E|}{p_i^{c-1}} \right), \tag{1}$$

Assuming $|E| \leq c_1 M$, we have that $|E| \leq c'_1 E[|P_i - E|]$ for all i , which means that “on average” at least a constant fraction of the samples in P_i are random. In this case (1) becomes $O(N/p_i^{c-1})$. Setting $c = 1$ yields that the expected total size of the conflict lists is linear.

Since the conflict list size is expected linear, the initialization step of our algorithm takes expected $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os. In each round, Step 1 takes $O(\frac{N}{B})$ I/Os, and since Step 2.1 is repeated only a constant number of times with high probability, the total cost of Step 2 is $O \left(\sum_{e \in \mathcal{T}_i} t_e \log t_e \cdot \frac{|P_{|e|}|}{B} \right)$ with high probability. Using (1) we can argue that the expected value of this expression is $O(\frac{N}{B})$, with details left in the full version of the paper. Summing this expected cost over all rounds of the algorithm, we obtain the following.

Theorem 1. *The constrained Delaunay triangulation of a set of N points \mathbb{R}^2 and a set of segments S can be computed in $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ expected I/Os, provided that $|S| \leq c_0 M$, where c_0 is a constant.*

3 Experiments

Simplified Algorithm and Implementation Details. We implemented and experimented with a simplified version of the theoretical algorithm described in Section 2. The main observation behind our simplification is that one round of the multi-round theoretical algorithm is enough to handle most real-world datasets. Even if we only have 128MB of main memory, which is more than the amount of memory needed to triangulate 0.1 million points, about $(10^5)^2 = 10^{10}$ points can be processed with just one round. This naturally leads to the following simple and practical algorithm:

1. Compute a random sample P_1 of P of size $c \cdot \max\{K, \sqrt{N}\}$ that includes all endpoints of segments in S , where c is a constant.

2. Construct $\text{CDT}(P_1, S)$ in memory using the `triangle` package.
3. For each point $p \in P$ in turn we determine the edges that p is in conflict with, generating a pair (e, p) for each such edge $e \in \text{CDT}(P_1, S)$. We then sort all these pairs to construct the conflict list $P|_e$ for each edge e . If any conflict list is larger than M , we restart the algorithm and take a new sample.
4. For each edge $e \in \text{CDT}(P_1, S)$ in turn we load its conflict list $P|_e$ into memory and construct $\text{CDT}(P|_e, I_e)$ using the `triangle` package. Then we report all the triangles whose circumcenters are inside $\tau(e)$.

Note that since we compute $\text{CDT}(P_1, S)$ in Step 2, we require that both K and \sqrt{N} are smaller than the memory size.

Since Step 3 is the only nontrivial step in the algorithm, we describe it in a little more detail. We first scan through the input points, and find conflicting edges with $\text{CDT}(P_1, S)$ kept in internal memory. To find the edges in conflict with a point p (internal memory) efficiently, it is sufficient to find all triangles in conflict with p ; $\triangle uvw$ is in conflict with p if $p \in D(u, v, w; S)$. Since all triangles in conflict with p are connected, we simply first locate the triangle containing p and then perform a BFS search to find all triangles that are in conflict with p . Rather than using a complicated (internal memory) point location structure to find the triangle of $\text{CDT}(P_1, S)$ containing p , we pre-sort all points according to the Hilbert space-filling curve, which has high spatial locality, and use a simple point-location algorithm while processing the points in Hilbert order: To locate a point p , we start from the triangle γ where the previous point was located and “walk towards” p by traversing all triangles intersected by the line segment from the centroid of γ to p . Since the locations of consecutive points are likely to be very close (due to the Hilbert ordering), we in practice perform each point location query in constant time. At the end of Step 3 we sort the list of edge-point pairs.

In practice, the efficiency of our simplified algorithm mainly depends on the total size of the conflict lists. The theoretical analysis in Section 2 shows that the expected total size is linear and in practice the constant is roughly 9. We reduce the total conflict size and thus improve the overall efficiency of the algorithm by combining several adjacent edges into a single “edge group”, computing the conflict list for each edge group, and solving the subproblem for each edge group. Nevertheless, some technical subtleties need to be taken care of when implementing this idea, which we explain in details in the full version.

Experimental Setup and Datasets. We implemented our simplified constrained Delaunay triangulation algorithm in C++ using TPIE [4]. We used `double` to store the coordinates of each point. For experimentation, we used a 2.4GHz Intel XEON machine with hyperthreading, running Linux with kernel 2.4.5-smp, and a local disk system consisting of four 10000RPM 72GB SCSI disks in RAID-0 configuration. The machine had 1GB main memory, but we restricted it to use only 128MB of memory in order to obtain a large data size to memory size ratio. All input, output and temporary files were stored on the local disk system.

We experimented with both synthetic and real-life data. For the synthetic data, we used four different distributions that have been used to evaluate the

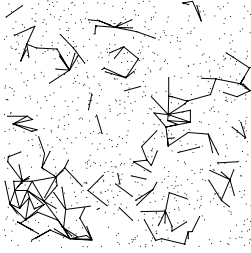


Fig. 4. Sample datasets of 1000 points from uniform distribution with segments

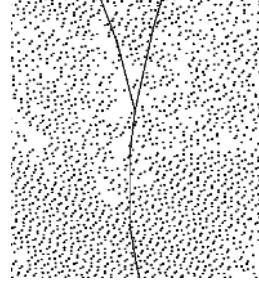


Fig. 5. LIDAR data

performance of Delaunay triangulation algorithms: uniform, normal, Kuzmin, and line singularity. See [6] for a definition of these distributions. Due to lack of space, we only report results on the uniform distribution in this abstract. Complete experiment results can be found in the full version of the paper.

After generating a point set P from one of the distributions, we generate the segment set S as follows: To obtain a segment $s \in S$, we first choose one endpoint uniformly at random from P . With some probability α we choose the other endpoint uniformly at random from P ; with probability $1 - \alpha$ we choose it uniformly at random from the endpoints of the segments already in S . We add s to S if it does not intersect any other segment in S and the length of s is smaller than some threshold δ . In our experiments we fixed $\alpha = 0.2$. An example of the segments generated this way are shown in Figure 4.

Our real-life datasets consist of LIDAR data for the Neuse River Basin of North Carolina [1]. This data consist of points $p = (x, y, z)$ in \mathbb{R}^3 and to obtain a point set P in \mathbb{R}^2 we simply used the x and y coordinates. We broke the data into a number of “tiles” geographically, and concatenated different subsets of the tiles together to create 9 datasets of increasing sizes. For the segments S , we used road data segments obtained from the TIGER/Line data [17]. The numbers of points and segments of the datasets are listed in Table 1; the last dataset covers the entire Neuse River Basin and has half of billion points. A portion of the LIDAR data is shown in Figure 5.

Delaunay Triangulation Experiments. We first investigate the performance of our algorithm when $S = \emptyset$, that is, when we are computing standard Delaunay triangulations. We compared our external memory algorithm (EM) with the

Table 1. The number of points and segments in each dataset of the Neuse River Basin

Dataset	1	2	3	4	5	6	7	8	9
# points (million)	16.8	27.7	44.5	58.5	90.8	116.2	163.1	257.1	503.7
# segments (thousand)	19.5	27.8	55.7	44.9	50.5	77.3	137.3	627.1	755.0
Input file size (MB)	336	554	890	1176	1816	2324	3262	5142	10074

(internal memory) divide-and-conquer (D&C) and incremental (INC) algorithm as implemented in the `triangle` package [16]. Since it is known that pre-sorting the points along some space-filling curve improves the performance of D&C and especially INC greatly with modern memory hierarchies [3], we sorted the points along the Hilbert curve in all our experiments. If the points are not sorted, D&C starts thrashing and takes more than 10 hours to complete on a dataset of only 5 million points; INC starts thrashing on an even smaller dataset of 2 million points. The time used to perform the Hilbert curve sort is not included in the computation times reported below.

The experimental results of our experiments on the uniform distribution with datasets of sizes varying from 10^6 to 10^7 are shown in Figure 6. Note that the 128MB main memory can only hold the data structure for triangulating roughly 1 million points. The results from the other distributions are similar. In all experiments, INC performs best. Its running time is almost linear in the data size because its data structure is visited in a highly local manner. The running time of our EM algorithm is around 20% worse than INC because of the overhead in the conflict lists. Although the D&C algorithm is faster than the two algorithms as long as the dataset fits in main memory, as soon as the dataset size grows larger, its performance quickly degenerates.

Constrained Delaunay Triangulation Experiments. Next we compared our EM algorithm with the algorithm (INC) implemented in `triangle` [16], which first constructs a Delaunay triangulation on the input points P (using the INC algorithm discussed above), and then inserts all the segments in S one by one. As before we pre-sorted the points by Hilbert values; we sorted the segments by the Hilbert value of one of their endpoints.

The running times of our first set of experiments on the uniform distribution are shown in Figure 7. We fixed the number of points to be 10^7 and generated up to 10^5 segments, each of length at most $\delta = 0.003$. The range of the number of segments are chosen to resemble the segment-to-point ratios of the real-life LIDAR datasets, as well as larger ratios. The experimental results show that

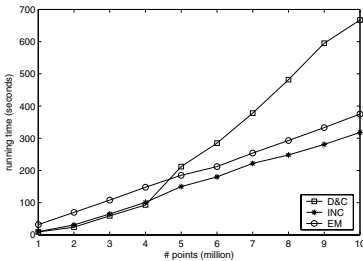


Fig. 6. Delaunay triangulation results on uniform distribution

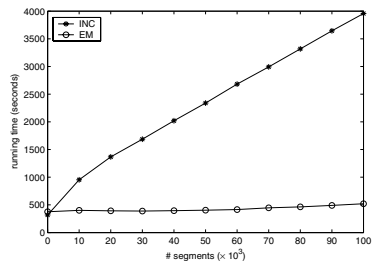


Fig. 7. Constrained Delaunay triangulation results on uniform distribution

our EM algorithm performs significantly better than INC. The main reason for this is probably that while our algorithm incrementally inserts points in a small constrained Delaunay triangulation in memory ($CDT(P_1, S)$), the INC algorithm incrementally inserts segments in a much larger (and larger than main memory) constrained Delaunay triangulation containing all the points.

The performance of the EM algorithm starts to (very slowly) degenerate at around 60,000 segments. This can be explained by the fact that the memory usage of the algorithm almost only depends on the sample size $|P_1|$; at $K = 60,000$ the sample is about the size of the main memory (we use about 5MB per 10,000 points, and sample $3K$ points; the system daemons use at least 30MB). Although in theory our algorithm only works when the sample fits in internal memory, we see that thrashing does not happen when this assumption is violated. Instead the performance of the algorithm degrades quite gracefully because the algorithm has a very local memory access pattern. Note that as the number of segments approaches N , our algorithm will degenerate into INC.

Next we investigated how the segment length affects performance. Using 10^7 points from the uniform distribution, we generated 10,000 segments with varying δ from 0.001 to 0.1 using only segments of length between $\delta/2$ and δ . The results of the experiments with these datasets are given in Figure 8. The results show that the running times of both algorithms are relatively unaffected by segment length. Maybe somewhat counter-intuitively, the running time of EM decreases as the segments get longer. This is probably because while longer segments increase the time to triangulate the sample, they also reduce the conflict list size somewhat.

The running times of our experiments with the LIDAR datasets are shown in Figure 9. Note that the smallest LIDAR dataset is larger than the largest of our synthetic dataset, thus, due to insufficient address space on a 32-bit machine (there is a 4GB limit on the address space for each process), we were unable to run INC except on the smallest dataset. In Figure 9 we show a breakdown of the running time of the EM algorithm into different phases: triangulating the

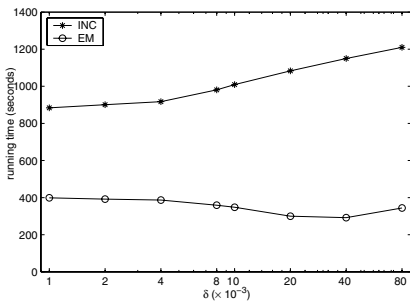


Fig. 8. Constrained Delaunay triangulation results with varying segment lengths

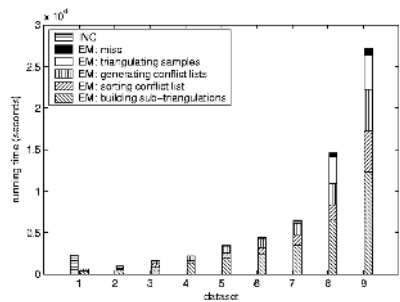


Fig. 9. Constrained Delaunay triangulation results on real datasets

samples, generating the conflict lists, sorting the conflict lists, and building the sub-triangulations. Except on the last two datasets, the total running time is dominated by the last three phases, which essentially depends on the number of points. On the last two datasets, the number of segments is much larger than in the other datasets, and the time spent on building $\text{CDT}(P_1, S)$ starts to be significant. However, since P_1 is still much smaller than the entire dataset, our algorithm is still much faster than building the entire constrained Delaunay triangulation directly.

References

1. North Carolina Flood Mapping Program. <http://www.ncfloodmaps.com>.
2. A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
3. N. Amenta, S. Choi, and G. Rote. Incremental constructions con brio. In *Proc. 19th Annu. ACM Sympos. Comput. Geom.*, pages 221–219, 2003.
4. L. Arge, O. Procopiuc, and J. S. Vitter. Implementing I/O-efficient data structures using TPIE. In *Proc. European Symposium on Algorithms*, pages 88–100, 2002.
5. F. Aurenhammer and R. Klein. Voronoi diagrams. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 201–290. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 2000.
6. G. E. Blelloch, G. L. Miller, J. C. Hardwick, and D. Talmor. Design and implementation of a practical parallel Delaunay algorithm. *Algorithmica*, 24(3):243–269, 1999.
7. *The CGAL Reference Manual*, 1999. Release 2.0.
8. L. P. Chew. Constrained Delaunay triangulations. *Algorithmica*, 4:97–108, 1989.
9. K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete Comput. Geom.*, 4:387–421, 1989.
10. A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. Ramos. Randomized external-memory algorithms for some geometric problems. *International Journal of Computational Geometry & Applications*, 11(3):305–337, June 2001.
11. M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 714–723, 1993.
12. L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7:381–413, 1992.
13. P. Kumar and E. A. Ramos. I/O-efficient construction of voronoi diagrams. Technical report, 2002.
14. K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, UK, 2000.
15. R. Seidel. Constrained Delaunay triangulations and Voronoi diagrams with obstacles. *Computer Science Division*, ??, June 1989. UC Berkeley.
16. J. R. Shewchuk. Triangle: engineering a 2d quality mesh generator and Delaunay triangulator. In *First Workshop on Applied Computational Geometry*. Association for Computing Machinery, May 1996.
17. *TIGER/Line™ Files, 1997 Technical Documentation*. Washington, DC, September 1998. <http://www.census.gov/geo/tiger/TIGER97D.pdf>.

Convex Hull and Voronoi Diagram of Additively Weighted Points

Jean-Daniel Boissonnat and Christophe Delage

INRIA Sophia-Antipolis, 2004, route des lucioles,
06902 Sophia-Antipolis cedex, France

{Jean-Daniel.Boissonnat, Christophe.Delage}@sophia.inria.fr
<http://www-sop.inria.fr/>

Abstract. We provide a complete description of dynamic algorithms for constructing convex hulls and Voronoi diagrams of additively weighted points of \mathbb{R}^d . We present simple algorithms and provide a description of the predicates. The algorithms have been implemented in \mathbb{R}^3 and experimental results are reported. Our implementation follows the CGAL design and, in particular, is made both robust and efficient through the use of filtered exact arithmetic.

1 Introduction

In this paper, we provide a complete description of dynamic algorithms for constructing convex hulls and Voronoi diagrams of additively weighted points of \mathbb{R}^d . The algorithms have been implemented in \mathbb{R}^3 and experimental results are reported.

Our motivation comes from the fact that weighted points can be considered as hyperspheres when the weights are positive and is twofold. On one hand, spheres are non linear objects and, besides the combinatorial and algorithmic questions, numerical and robustness issues deserve a careful investigation, which has not been fully done yet. On the other hand, spheres are objects of major concern in various fields, most notably structural biology, and effective implementations of basic geometric algorithms for spheres are needed.

We first revisit the problem of computing the convex hull of n weighted points of \mathbb{R}^d . This problem has already been solved optimally [1,2]. We present a simpler fully dynamic algorithm and provide a complete description of all the predicates for any d .

We then consider the construction of additively weighted Voronoi diagrams. It is known that the construction of such diagrams reduces to intersecting a power diagram in \mathbb{R}^{d+1} with half-cones [3]. We are not aware of robust implementations of this algorithm. Other algorithms have been recently designed and implemented in the planar case [4,5]. In \mathbb{R}^3 , we are only aware of two prototype implementations, one by Will [6] for computing a single cell, and one by Kim et al. [7] that computes the entire diagram. None of these implementations is provably robust. Moreover, the algorithm by Kim et al. assumes that the graph of the edges of each cell is connected, which is not true in general.

We apply our result on the construction of the convex hull of additively weighted points to the construction of a Voronoi cell in the Voronoi diagram of n additively weighted points. The construction, which makes use of inversion, is close to the algorithm of [2]. The main contribution of this work is to provide a full analysis of the predicates involved, a thorough treatment of the degenerate cases, and a CGAL implementation. Our predicates, when specialized to the planar case ($d = 2$), are simpler and of lower degree than the best predicates known so far [8,9]. Our implementation follows the CGAL design and, in particular, is made both robust and efficient through the use of filtered exact arithmetic.

The paper is organized as follows. In section 2, we establish a new correspondence between convex hulls of additively weighted points in \mathbb{R}^d and power diagrams of spheres of \mathbb{R}^d , from which we deduce an algorithm to construct such hulls. In section 3, we recall a similar correspondence for a cell in the Voronoi diagram of additively weighted points and present an algorithm for constructing such a cell. In section 4, we describe the predicates. In section 5, we show how to handle the degenerate cases. In section 6, we report on experimental results. Finally, we conclude in section 7.

In the sequel, a weighted point, or *site* for short, of \mathbb{R}^d is a pair $s = (p, w)$ where p is a point of \mathbb{R}^d , and w is a real number, we refer to p and w as the *center* and the *weight* of the site, respectively. When w is positive, we also call a weighted point a hypersphere. Given a set n hyperspheres $\Sigma = \{\sigma_1, \dots, \sigma_n\}$ of \mathbb{R}^d , $\sigma_i = (c_i, r_i)$, the *power* of a point $x \in \mathbb{R}^d$ to σ_i is $d_P(\sigma_i, x) = (x - c_i)^2 - r_i^2$ and the *power diagram* of Σ , noted $\mathcal{P}(\Sigma)$, is the subdivision of \mathbb{R}^d consisting of the n cells $P(\sigma_1), \dots, P(\sigma_n)$ where $P(\sigma_i) = \{x \in \mathbb{R}^d, d_P(\sigma_i, x) \leq d_P(\sigma_j, x), j = 1, \dots, n\}$. We write $P(\sigma_1, \dots, \sigma_k) = P(\sigma_1) \cap \dots \cap P(\sigma_k)$. When non empty, $P(\sigma_1, \dots, \sigma_k)$ is a face of $\mathcal{P}(\Sigma)$, of dimension $d - k + 1$ if the hyperspheres are in general position.

2 Convex Hull of Additively Weighted Points

Let $\mathcal{S} = \{s_1, \dots, s_n\}$ be a set of weighted points of \mathbb{R}^d . We write $s_i = (p_i, w_i)$, $i = 0, \dots, n$. We consider first the case where all the weights w_i are non negative, i.e. the sites are hyperspheres. The *convex hull* of \mathcal{S} , $\text{CH}(\mathcal{S})$, is the smallest closed convex subset of \mathbb{R}^d containing all the hyperspheres of \mathcal{S} . A *supporting hyperplane* H of \mathcal{S} is a hyperplane tangent to at least one of the hyperspheres of \mathcal{S} , and such that all the hyperspheres of \mathcal{S} lie in the same half-space limited by H . A *facet* of $\text{CH}(\mathcal{S})$ of *circularity* k , $0 \leq k < d$, is the portion of $\partial\text{CH}(\mathcal{S})$ that consists of the points whose supporting hyperplanes are tangent to a same subset of $d - k$ hyperspheres. For $d = 3$, faces of circularity 0 are planar faces tangent to three hyperspheres, faces of circularity 1 are conical patches tangent to two hyperspheres, and faces of circularity 2 are spherical patches contained in some s_i .

From Power Diagrams to Convex Hulls of Hyperspheres. Let H be a supporting hyperplane tangent to k hyperspheres, and m be the unit normal

vector of Π pointing away from \mathcal{S} . As there is exactly one supporting hyperplane that has a given oriented normal, m defines Π uniquely. Π is a supporting hyperplane tangent to s_1, \dots, s_k if and only if:

$$m \cdot (p_i - p_1) = w_1 - w_i, 1 \leq i \leq k \tag{1}$$

$$m \cdot (p_i - p_1) < w_1 - w_i, k < i \leq n \tag{2}$$

We rewrite (2) as follows:

$$\begin{aligned} & m \cdot (p_i - p_1) < w_1 - w_i \\ \iff & -m \cdot p_1 - w_1 < -m \cdot p_i - w_i \\ \iff & (m - p_1)^2 - (p_1^2 + 2w_1) < (m - p_i)^2 - (p_i^2 + 2w_i) \end{aligned}$$

(1) can be rewritten the same way. Thus, denoting $r_i^2 = p_i^2 + 2w_i$, $\sigma_i = (p_i, r_i)$ and $\Sigma = \{\sigma_1, \dots, \sigma_k\}$, this is equivalent to m being in the open $(d - k + 1)$ -face of $P(\sigma_1, \dots, \sigma_k)$ in the power diagram $\mathcal{P}(\Sigma)$. As m belongs to the unit hypersphere $\mathbb{S} = \{x \in \mathbb{R}^d : \|x\| = 1\}$, we have proven

Lemma 1. *The k -faces of $\mathcal{P}(\Sigma) \cap \mathbb{S}$ are in 1-1 correspondence with the facets of circularity k of $\partial\text{CH}(\mathcal{S})$.*

The above construction works also when some or all w_i are negative. Although a geometric interpretation in terms of convex hull is then missing, the result of our construction is called the convex hull of the weighted points, or AWCH for short, by analogy to the case of positive weights.

We now present a static algorithm and an incremental algorithm for constructing a AWCH. The affine hull of a face f is denoted $\text{aff}(f)$. We say that a k -face f' is a *sub-face* of a $(k + 1)$ -face f (and conversely that f is a *super-face* of f') when $f' \subseteq f$. For a face f and a sub-face f' of f , $H(f, f')$ denotes the halfspace of $\text{aff}(f)$ bounded by $\text{aff}(f')$ that contains f . For instance, when f is a 1-face (a line segment), f' is one of its endpoints, and $H(f, f')$ is the ray issued from f' that contains f . We assume that the s_i are in general position so that we do not have any degeneracy. Degeneracies will be considered in section 5.

Static Algorithm. The algorithm first constructs the power diagram $\mathcal{P}(\Sigma)$ and then determines, for each face f of $\mathcal{P}(\Sigma)$, whether f intersects \mathbb{S} or not. The result is stored in $\text{tag}[f]$:

- $\text{tag}[f] = \emptyset$ if and only if $\text{aff}(f)$ is outside \mathbb{S} ,
- $\text{tag}[f] = \ominus$ if and only if f does not intersect \mathbb{S} but $\text{aff}(f)$ intersects \mathbb{S} ,
- $\text{tag}[f] = \oplus$ if and only if f intersects \mathbb{S} ,
- $\text{tag}[f] = \odot$ if and only if f is inside \mathbb{S} .

Assuming we know $\text{tag}[f']$ for each sub-face f' of f , we compute $\text{tag}[f]$ as follows:

1. If $\text{aff}(f)$ does not intersect \mathbb{S} , then $\text{tag}[f] = \emptyset$,
2. else, if for each sub-face f' of f , $\text{tag}[f'] = \odot$, then, by convexity of f , $\text{tag}[f] = \odot$,

3. else, if there is a sub-face f' of f such that $tag[f'] = \odot$ or $tag[f'] = \oplus$, then, by connexity of f , $tag[f] = \oplus$.
4. else, if for each sub-face f' of f , $tag[f'] = \emptyset$, and $aff(f) \cap \mathbb{S} \subseteq H(f, f')$, then f intersects \mathbb{S} and $tag[f] = \oplus$,
5. else, $tag[f] = \ominus$.

Assume w.l.o.g. that $f = P(\sigma_1, \dots, \sigma_k)$ and $f' = P(\sigma_1, \dots, \sigma_{k+1})$. This algorithm needs to evaluate the two following geometric predicates.

k -RADICALINTERSECTION(f) determines whether $aff(f)$ is outside \mathbb{S} or not.
 k -RADICALSIDE(f, f') determines whether $aff(f) \cap \mathbb{S} \subset H(f, f')$, assuming that $aff(f)$ intersects \mathbb{S} and $aff(f')$ is outside \mathbb{S} .

To compute the tags, we proceed by induction on the dimension of the faces. This takes a time proportional to the size of the diagram. It follows that the time complexity of the algorithm is upbounded by the time complexity of a power diagram algorithm. Hence, our algorithm computes the additively weighted convex hull of n sites in $\mathcal{O}\left(n \log n + n^{\lceil \frac{d}{2} \rceil}\right)$ time.

Incremental Algorithm. We now present an incremental algorithm for computing the additively weighted convex hull. We use an incremental algorithm for constructing the power diagram, and we attach to each face f the number $num[f]$ of its sub-faces that are tagged \oplus . Updating the power diagram when inserting a new hypersphere in the power diagram amounts to creating some new faces, deleting some faces, and replacing some faces by smaller ones. In this last case, a face f is replaced by a smaller face $\bar{f} \subset f$ that is incident to the cell of the new hypersphere. \bar{f} is called a *cut face*. Notice that a cut face of dimension less than d has exactly one new subface. We denote by m the number of deleted faces plus the number of new faces.

1. For a new face f , $num[f]$ is easily computed by looking at all its sub-faces. This can be done in time proportional to m .
2. We now update $num[\bar{f}]$ for a cut face \bar{f} . We set $num[\bar{f}] = num[f]$. Then $num[\bar{f}]$ is decremented by the number of the sub-faces of f that are deleted, and updated according to the tag of the new and cut sub-faces of \bar{f} . This can be done in the following way. When the tag of a face f' changes, or f' is deleted, we update $num[f]$ for each *super-face* f of f' . Updating $num[\bar{f}]$ therefore takes $\mathcal{O}(m)$ time.
3. We update $tag[\bar{f}]$ for a cut k -face \bar{f} , $k > 1$. We compute $tag[\bar{f}]$ from $num[\bar{f}]$ and $tag[f']$, where f' is the new sub-face f' of \bar{f} . As \bar{f} is a cut face, $tag[\bar{f}]$ differs from $tag[f]$ only when $tag[f] = \oplus$. If $num[\bar{f}]$ is positive, then $tag[\bar{f}] = \oplus$. Now, if $num[\bar{f}]$ is 0, only the relative interior of \bar{f} can intersect \mathbb{S} . Hence
 - if $tag[f'] = \emptyset$, then we update $tag[\bar{f}]$ according to the outcome of k -RADICALSIDE(\bar{f}, f')
 - otherwise, as the set of the sub-faces of \bar{f} is connected (\bar{f} is of dimension at least 2), \bar{f} has the same tag as f' .

For a cut 1-face \bar{f} , we compute $tag[\bar{f}]$ directly. Updating the tag of a cut face takes a constant time.

The incremental algorithm for constructing an additively weighted convex hull has therefore the same complexity as the incremental algorithm that computes the associated power diagram. When the sites are inserted in random order, the expected time complexity of our algorithm is therefore $\mathcal{O}\left(n \log n + n^{\lceil \frac{d}{2} \rceil}\right)$.

Practical Complexity. Under realistic assumptions, our algorithms perform better than in the worst case. First, according to our experiments (see section 6), the number of hyperspheres with a non empty cell in the power diagram is usually proportional to the number of points h on the additively weighted convex hull. In that case, the running time for n insertions is $\mathcal{O}\left(n \log h + h^{\lceil \frac{d}{2} \rceil}\right)$. Moreover, h is typically much smaller than n . It is known that the convex hull of a set of n points uniformly distributed inside a sphere of \mathbb{R}^3 has $\mathcal{O}(\sqrt{n})$ points on its convex hull. The same result holds trivially for spheres with the same radius. In \mathbb{R}^3 , assuming that the number of cells in the power diagram is proportional to h and that h is $\mathcal{O}(\sqrt{n})$, the complexity of our algorithm is $\mathcal{O}(n \log n)$.

3 Additively Weighted Voronoi Diagram

The *additively weighted distance*, denoted d_+ , from a point m of \mathbb{R}^d to a site $s_i = (p_i, w_i)$ is $d_+(s_i, m) = \|p_i - m\| - w_i$. Considering the set $\mathcal{S} = \{s_1, \dots, s_n\}$ of sites, the *additively weighted Voronoi cell* of s_i , $V(s_i)$ is:

$$V(s_i) = \{m \in \mathbb{R}^d \mid \forall j, d_+(s_i, m) \leq d_+(s_j, m)\} .$$

It is possible that $V(s_i) = \emptyset$: this happens when $\forall m \in \mathbb{R}^d, \exists j, d_+(s_j, m) \leq d_+(s_i, m)$. In that case, we say that s_i is *hidden by* s_j . When dealing with hyperspheres (*i.e.* $w_i, w_j > 0$), s_i is hidden by s_j when $s_i \subseteq s_j$. The *additively weighted Voronoi diagram*, or AWVD for short, of \mathcal{S} , noted $\mathcal{V}(\mathcal{S})$, is the cell complex whose d -cells are the $V(s_i)$.

The construction of a single cell of the diagram reduces to computing an additively weighted convex hull, via an inversion. More precisely, the cell of s_1 in $\mathcal{V}(\mathcal{S})$ is combinatorially equivalent to the additively weighted convex hull of $\mathcal{S}' = \{s'_1, \dots, s'_n\}$, where s'_1 is centered at the origin and has weight 0, s'_i is centered at $\frac{p_i - p_1}{\alpha_i}$ and has weight $\frac{w_i - w_1}{\alpha_i}$, $\alpha_i = (p_i - p_1)^2 - (w_i - w_1)^2$, $i = 2 \dots n$. This scheme only works when s_1 is not hidden by, nor does it hide any other site s_i . See [2] for details.

Using the construction of a single cell as a subroutine, we can compute the whole diagram in a fully dynamic manner. All the Voronoi cells are stored in a data structure with pointers between the corresponding elements. In this data structure, $\Gamma(s)$ denotes the set of neighbors of s and $\Gamma_s(s')$ the set of neighbors of s that are also neighbors of s' . Two sites s and s' are called *neighbors* if $V(s)$ and $V(s')$ share a $(d - 1)$ -face. We add to the data structure an *infinite cell*, which is actually the plain additively weighted convex hull of the sites. That way, we handle unbounded faces, and sites lying on the convex hull seamlessly.

In order to keep track of the hidden sites, we keep, for each site s , a list *hidden*[s] of the sites s hides. We now describe the three main ingredients needed to update an additively weighted Voronoi diagram.

Localization. Given a point m of \mathbb{R}^d and a starting site s , this procedure, called $\text{LOCATE}(m, s)$, returns the cell of the diagram in which m lies. This is done by means of a simple walk: if a neighbor s' is closer to m than s , jump to s' and we iterate; otherwise, m belongs to the cell of s and we stop. This localization algorithm requires only one predicate: given two sites s_1 and s_2 , determine if a query point m is closer to s_1 than to s_2 . This predicate is called SIDEOFBISECTOR .

Insertion. The insertion procedure needs to decide whether a site s_1 hides another site s_2 : we call this predicate $\text{ISTRIVIAL}(s_1, s_2)$. To avoid ambiguities when considering diagrams of different sets of sites, we introduce the following notation. Given a set of sites \mathcal{S} and $s, s' \in \mathcal{S}$, we denote $V_{\mathcal{S}}(s)$ the cell of s in the additively weighted Voronoi diagram of \mathcal{S} , and $V_{\mathcal{S}}(s, s') = V_{\mathcal{S}}(s) \cap V_{\mathcal{S}}(s')$. A site $s \notin \mathcal{S}$ is *in conflict with* $s' \in \mathcal{S}$ if and only if $V_{\mathcal{S}}(s') \neq V_{\mathcal{S} \cup \{s\}}(s')$. Notice that only the non-hidden sites of \mathcal{S} may be in conflict with s . The *conflict graph* of a site $s \notin \mathcal{S}$ is $G = (X, E)$ where $X \subseteq \mathcal{S}$ is the set of sites in conflict with s , and $xy \in E$ if and only if $V_{\mathcal{S}}(x, y) \cap V_{\mathcal{S} \cup \{s\}}(s) \neq \emptyset$. In other words, the conflict graph of s is the dual of the restriction of $\mathcal{V}(\mathcal{S})$ to $V_{\mathcal{S} \cup \{s\}}(s)$.

Lemma 2. *The conflict graph of s is connected.*

Proof. Given two sites x and y in the conflict graph G of some s , we take p_x in $V_{\mathcal{S}}(x) \cap V_{\mathcal{S} \cup \{s\}}(s)$ and p_y in $V_{\mathcal{S}}(y) \cap V_{\mathcal{S} \cup \{s\}}(s)$. As $V_{\mathcal{S} \cup \{s\}}(s)$ is arc connected, we can follow a path from p_x to p_y in $V_{\mathcal{S} \cup \{s\}}(s)$, and each time we cross a $(d-1)$ -face of the diagram, we follow the corresponding edge of G . This gives a path from x to y in G . □

The first insertion (*i.e.* the insertion in an empty diagram) is easy: we just create a new cell covering all \mathbb{R}^d . Once there is a least one site in the diagram, the insertion procedure of a new site s is the following.

1. Locate the center of s , let s' be the site such that the center of s lies in $V(s')$.
2. If s is hidden by s' , then add s to $\text{hidden}[s']$.
3. Else, s' is a vertex in the conflict graph G of s , so we walk on G , starting from s' , and for each s'' in G :
 - if s'' is hidden by s , add s'' to $\text{hidden}[s]$,
 - else, insert s in $V(s'')$ and s'' in $V(s)$.

Removal. Removing a site s from a diagram is straightforward. Firstly, remove s from the cells adjacent to $V_+(s)$, Secondly, let $\{s_1, \dots, s_k\}$ be the neighbors of s ; for all $1 \leq i, j \leq k, i \neq j$, insert s_i into the cell of s_j to rebuild the hole made by the removal of s . And finally, insert all the sites s was hiding.

Complexity. The localization algorithm described here takes time linear in the size of the diagram, which can be improved to randomized logarithmic time by using a hierarchical data structure as in [4].

The construction of the Voronoi diagram of n sites performs $\mathcal{O}(n)$ localizations, and constructs $\mathcal{O}(n)$ additively weighted convex hulls of $\mathcal{O}(n)$ sites. The overall time to construct the Voronoi diagram of n sites is therefore:

$$\mathcal{O}\left(n \log n + n \left(n \log n + n^{\lceil \frac{d}{2} \rceil}\right)\right) = \mathcal{O}\left(n^2 \log n + n^{\lceil \frac{d}{2} \rceil + 1}\right).$$

Which gives, for $d = 3$, $\mathcal{O}(n^3)$. This bound can be improved under the following assumptions:

1. $\mathcal{O}(\sqrt{n})$ sites appear on the additively weighted convex hull of \mathcal{S} ,
2. the sites have $\mathcal{O}(1)$ neighbors,
3. the underlying power diagram of every Voronoi cell has $\mathcal{O}(s)$ non-hidden points, where s is the number of neighbors of the cell.

Those assumptions are not too restrictive, and happen to be satisfied on a variety of input data (see section 6.) Assumptions 1 and 3 implies that the construction of the infinite cell (*i.e.* the AWCH) take $\mathcal{O}(n \log n)$ time. Assumptions 2 and 3 implies that we construct $\mathcal{O}(n)$ finite cells of size $\mathcal{O}(1)$, and that it takes $\mathcal{O}(n)$ time. This leads to an expected running time of $\mathcal{O}(n \log n)$, for constructing the additively weighted Voronoi diagram.

4 Predicates

We consider a set $\mathcal{S} = \{s_1, \dots, s_n\}$ of sites, where s_i is centered at p_i , and has weight w_i . If each input data is a b -bit integer, the size of each monomial occuring in a predicate is upper bounded by $2^{(b+1)d}$. Moreover, let v be the number of variables that occur in a predicate; for the predicates considered in this

Table 1. Predicate degree summary

Algorithm	AWCH		AWVD			
Dimension	2	3	$d > 3$	2	3	$d > 3$
ISTRIVIAL				2	2	2
SIDEOFBISECTOR				4	4	4
ORIENTATION	2	3	d	4	5	$d + 2$
POWERTEST	3	4	$d + 1$	5	6	$d + 3$
1-RADICALINTERSECTION	2	2	2	6	6	6
2-RADICALINTERSECTION	4	8	8	8	16	16
3-RADICALINTERSECTION		6	12		10	20
k -RADICALINTERSECTION, $1 < k < d$			$4k$			$4k + 8$
d -RADICALINTERSECTION			$2d$			$2d + 4$
1-RADICALSIDE	1	1	1	3	3	3
2-RADICALSIDE	3	3	3	7	7	7
3-RADICALSIDE		5	5		9	9
k -RADICALSIDE, $1 < k \leq d$			$2k - 1$			$2k + 3$
Maximum degree	4	8	$4d - 4$	8	16	$4d + 4$

paper, v is a constant. It follows that a predicate of degree d requires precision $p \leq d(b+1+\log v)$. Here, the predicates are polynomials in the unknowns p_i, w_i , and the algebraic degree of each of them is given. In addition to the predicates mentioned in section 2 and 3, we need the two well-known predicates that are needed to construct the power diagram: `ORIENTATION` and `POWERTEST`.

Predicates `ISTRIVIAL` and `SIDEOFBISECTOR` are detailed in [10] for $d = 2$, and are straightforward to extend to arbitrary dimension. `ISTRIVIAL` is of degree 2, and `SIDEOFBISECTOR` is of degree 4. Basic linear algebra provides explicit formulas for the other predicates. The maximum degree of the predicates for the `AWCH` is 4 in 2D, 8 in 3D, and in general, $4d - 4$ in dimension d . The maximum degree of the predicates for the `AWVD` is 8 in 2D, 16 in 3D, and in general, $4d + 4$ in dimension d . See Table 1. This compares very well to the predicates of the algorithm for the additively weighted Voronoi diagram of [4], detailed in [8], which have a maximal degree of 16 for $d = 2$.

5 Degenerate Cases

Additively Weighted Convex Hull. Here, we show how to handle degeneracies in the algorithm for the convex hull of additively weighted points. We call a case *degenerate* when some predicate returns 0, instead of “positive” or “negative”. A simple way of dealing with these cases is to carefully choose a non-zero sign to be returned by a predicate when it evaluates to 0.

- Predicates `ORIENTATION` or `POWERTEST` return zero when Σ is not in general position. Any standard perturbation scheme will work for us.
- When predicate `k-RADICALINTERSECTION` returns 0, some $(d - k)$ -flat is tangent to \mathbb{S} (it intersects \mathbb{S} but not the open ball bounded by \mathbb{S} .) We can consider that this $(d - k)$ -flat lies outside \mathbb{S} .
- Predicate `k-RADICALSIDE(f, f')` returns 0 when the projection of the origin on $\text{aff}(f)$ is on $\text{aff}(f')$. In that case, both $\text{aff}(f)$ and $\text{aff}(f')$ intersects \mathbb{S} , or both do not intersect \mathbb{S} . As predicate `k-RADICALSIDE` is only called when $\text{aff}(f)$ intersects \mathbb{S} and $\text{aff}(f')$ does not, this predicate is never called on degenerate inputs.

Additively Weighted Voronoi Diagram. In the case of the additively weighted Voronoi diagram, the previous perturbation scheme does not work. Indeed, as we compute the Voronoi cells separately, we not only need to resolve degeneracies in each cell, but also to ensure that consistent decisions are taken when we compute the neighboring cells. A set of sites \mathcal{S} is called *degenerate* when there exists $k + 1$ sites of \mathcal{S} s_0, \dots, s_k , $1 \leq k < d$, such that $V(s_0, \dots, s_k)$ is not empty and is of dimension strictly less than $d - k$. When an input is non-degenerate, any small enough perturbation will let the combinatorial structure of the Voronoi diagram unchanged. For a face f , we define $L(f) = \{s \in \mathcal{S} : f \subseteq V(s)\}$. If a degeneracy $\{s_0, \dots, s_k\}$ is minimal (*i.e.* if $\{s_0, \dots, s_k\} \setminus \{s_i\}$ is not degenerate for $0 \leq i \leq k$) then perturbing the weight of any site in $L(V(s_0, \dots, s_k))$ will remove the degeneracy.

Given a degenerate input $\{s_0, \dots, s_k\}$, finding a minimal degeneracy is easy: w.l.o.g. we check if $\{s_0, \dots, s_{k-1}\}$ is still degenerate. As $V(s_0, \dots, s_k) \neq \emptyset$, $V(s_0, \dots, s_{k-1})$ has dimension at least 1. A degeneracy of dimension $m \geq 1$ in the intersection between the unit hypersphere and a power diagram can only appear if two $(m+1)$ -faces of the power diagram are equal, which can be tested with the `ORIENTATION` and `POWERTEST` predicates.

Now, we can handle the degeneracies in our algorithm by means of symbolic perturbations. When faced with a predicate that returns zero on $\{s_0, \dots, s_k\}$, we find a minimal degeneracy $\{s_0, \dots, s_{k'}\}$ and perturb one site, in $L(V(s_0, \dots, s_{k'}))$, say s_i , *i.e.* we replace w_i by $w_i + \epsilon$. The predicates are now polynomials in ϵ and we need to evaluate the sign of the non-zero coefficient of smallest degree. Notice that this scheme does not increase the degree of the predicates since the perturbation is linear. It can occur that some predicate returns zero, and the Voronoi diagram is not degenerate, and thus some site gets perturbed unnecessarily.

To ensure consistency from one cell to another, we just need to choose the site to perturb in a way that is independent of the site whose cell is under construction when we detect the degeneracy. One way to do that is to choose the smallest site according to some global ordering (for instance, the lexicographical order on the centers.)

6 Experimental Results

We have implemented both our algorithms for constructing the convex hull and the Voronoi diagram of weighted points in \mathbb{R}^3 . The implementations use CGAL 3.1, mainly its 3D regular triangulations (see [11]), which are the duals of the power diagrams. While not yet fully optimized, this implementation already follows the CGAL standard of genericity and robustness. The predicates are dynamically filtered to avoid problems of precision in degenerate, or near-degenerate, cases. We plan to have the code included in the CGAL library soon. The running times are obtained on a `ATHLON` running at 1333MHz, with 133MHz DDR-SDRAM memory and 256KB of L2 cache.

On Fig. 1, the degenerate input is a set of sites randomly chosen in a cube, with their weight equal to their height, so that all the sites are tangent to the lower face of the cube and the non-degenerate input is a set of sites uniformly distributed inside a sphere, the weights uniformly distributed in an interval.

On Fig. 2, the input comes from a direct application of our algorithm. The sites have their centers on a surface and the weights are of the form $-\frac{\text{lfs}(x)}{k}$ where $\text{lfs}(x)$ is an approximation of the local feature size of the surface at x , and k is a parameter. This kind of diagram has been used to efficiently compute a sizing field, for 3D meshing (see [12] for details). On Fig. 2, $k = 1$ on the left and $k = 0.3$ on the right.

Both algorithms are incremental, and as such, their running time is likely to depend on the insertion order. Fig. 1 and 2 show three insertion orders: sites with small weights first, sites with large weights first, and random. In all cases, the

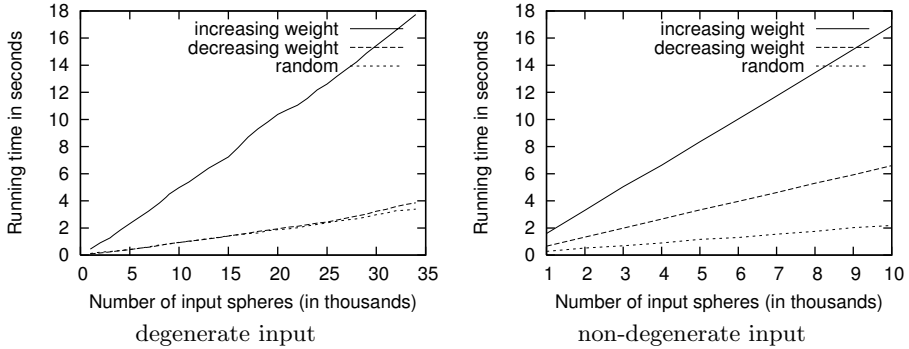


Fig. 1. Additively Weighted convex hull benchmarks, all using filtered predicates, for various input, and insertion order

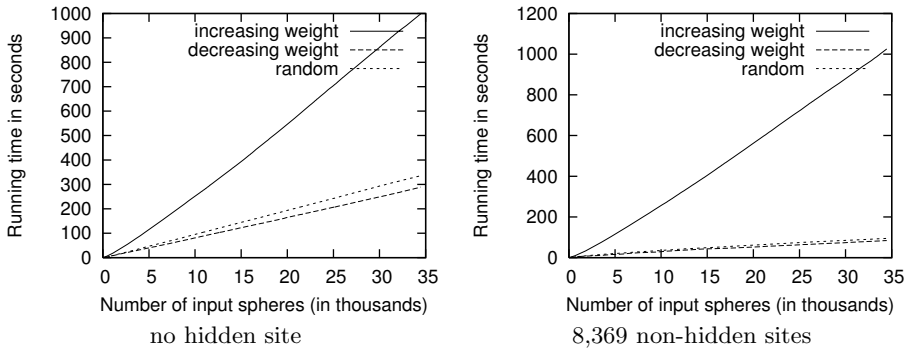


Fig. 2. Additively weighted Voronoi diagram benchmarks using filtered predicates for various input sizes, numbers of hidden sites, and insertion orders

algorithms are faster when the sites are inserted in order of decreasing weights. The reason is that a site with a larger weight tends to have more neighbors, and thus, tends to take longer to insert. The difference is even greater when there are many hidden spheres.

A screenshot is shown in Figure 3, where one cell is represented by meshing its boundary. Our implementation computes all the edges of the cell (*i.e.* facets of circularity 1 in the underlying convex hull), and sample them. Then, each face of the cell (*i.e.* each facet of circularity 2 in the convex hull) is approximated using the meshing algorithm of [13]. We plan to have the code included in the CGAL library soon.

In our experiments, we observed a remarkable phenomenon: almost all the spheres that do not contribute to the additively weighted convex hull are hidden (*i.e.* have empty cells) in the underlying power diagram. This also occur when the AWCH are cells of an additively weighted Voronoi diagram. In no Voronoi cell the examples shown here, the number of cells in the power diagram is more

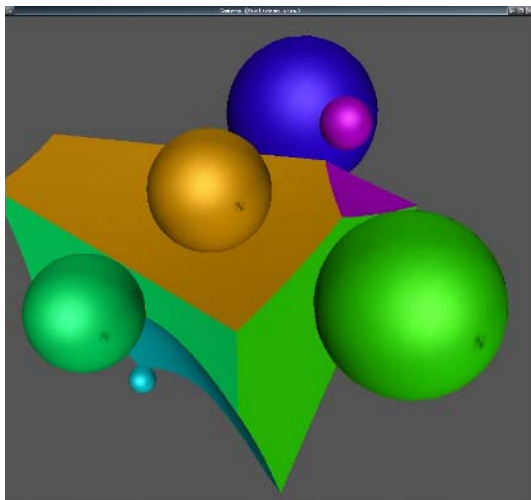


Fig. 3. A screenshot of one cell of a 3D additively weighted Voronoi diagram

than seven times the number of neighbors of the Voronoi cell. Moreover, for only 1% of the Voronoi cells, the number of cells in the underlying power diagram is more than twice the number of neighbors in the Voronoi diagram. Although this observation does not hold in general—it is possible to construct n spheres such that only $\mathcal{O}(1)$ of them contribute to the convex hull, while all of them appear in the power diagram—this makes our algorithms efficient in practice.

7 Conclusion

We have presented fully robust implementations of two algorithms for constructing the convex hull and the Voronoi diagram of additively weighted points (and hyperspheres). To the best of our knowledge, no certified algorithms existed previously.

This work does not settle the main open question in this area : what is the combinatorial complexity of the Voronoi diagram of n additively weighted points in \mathbb{R}^d ? Tight bounds are only known for $d = 2$ and odd dimensions. We hope that experimenting with our code may provide new insights such as the one mentioned in section 6 that eventually will help improving the combinatorial bounds.

References

1. Boissonnat, J.D., Cérézo, A., Devillers, O., Duquesne, J., Yvinec, M.: An algorithm for constructing the convex hull of a set of spheres in dimension d . *Comput. Geom. Theory Appl.* **6** (1996) 123–130

2. Boissonnat, J.D., Karavelas, M.: On the combinatorial complexity of Euclidean Voronoi cells and convex hulls of d -dimensional spheres. In: Proc. 14th ACM-SIAM Sympos. Discrete Algorithms (SODA). (2003) 305–312
3. Aurenhammer, F., Imai, H.: Geometric relations among Voronoi diagrams. *Geom. Dedicata* **27** (1988) 65–75
4. Karavelas, M., Yvinec, M.: Dynamic additively weighted voronoi diagrams in 2d. In: Proc. 10th European Symposium on Algorithms. (2002) 586–598
5. Kim, D.S., Kim, D., Sugihara, K.: Updating the topology of the dynamic voronoi diagram for spheres in euclidean d -dimensional space. *Computer-Aided Design* **18** (2001) 541–562
6. Will, H.M.: Fast and efficient computation of additively weighted Voronoi cells for applications in molecular biology. In: Proc. 6th Scand. Workshop Algorithm Theory. Volume 1432 of Lecture Notes Comput. Sci., Springer-Verlag (1998) 310–321
7. Kim, D.S., Cho, Y., Kim, D., Bhak, J., Lee, S.H.: Euclidean voronoi diagram of 3d spheres and applications to protein structure analysis. In Sugihara, K., ed.: 1st International Symposium on Voronoi Diagrams in Science and Engineering. (2004)
8. Karavelas, M.I., Emiris, I.Z.: Root comparison techniques applied to computing the additively weighted Voronoi diagram. In: Proc. 14th ACM-SIAM Sympos. Discrete Algorithms (SODA). (2003) 320–329
9. Anton, F.: Voronoi diagrams of semi-algebraic sets. Ph.d. thesis, University of British Columbia (2004)
10. Karavelas, M.I., Emiris, I.Z.: Predicates for the planar additively weighted Voronoi diagram. Technical Report ECG-TR-122201-01, INRIA Sophia-Antipolis (2002)
11. : The CGAL Manual. (2004) Release 3.1.
12. Alliez, P., Cohen-Steiner, D., Yvinec, M., Desbrun, M.: Variational tetrahedral meshing. In: SIGGRAPH. (2005)
13. Boissonnat, J.D., Oudot, S.: Provably good surface sampling and approximation. In: Proc. 1st Symp. on Geometry Processing. (2003) 9–18

New Tools and Simpler Algorithms for Branchwidth*

Christophe Paul¹ and Jan Arne Telle²

¹ CNRS - LIRMM, Montpellier, France
paul@lirmm.fr

² Department of Informatics, University of Bergen, Norway
telle@ii.uib.no

Abstract. We provide new tools, such as k -troikas and good subtree-representations, that allow us to give fast and simple algorithms computing branchwidth. We show that a graph G has branchwidth at most k if and only if it is a subgraph of a chordal graph in which every maximal clique has a k -troika respecting its minimal separators. Moreover, if G itself is chordal with clique tree T then such a chordal supergraph exists having clique tree a minor of T . We use these tools to give a straightforward $O(m+n+q^2)$ algorithm computing branchwidth for an interval graph on m edges, n vertices and q maximal cliques. We also prove a conjecture of F. Mazoit [13] by showing that branchwidth is polynomial on a chordal graph given with a clique tree having a polynomial number of subtrees.

1 Introduction

Branchwidth and treewidth are connectivity parameters of graphs and whenever one of these parameters is bounded by some fixed constant on a class of graphs, then so is the other [14]. Since many graph problems that are in general NP-hard can be solved in linear time on such classes of graphs both treewidth and branchwidth have played a large role in many investigations in algorithmic graph theory. Recently there has been a focus on branchwidth [6,5,4,7,8] to give e.g. good heuristics for the travelling salesman problem and fast parameterized algorithms for various types of optimization problems. These algorithms always involve a stage that constructs a branch-decomposition with small branchwidth, and another stage solving the problem using the decomposition by a running time depending heavily on that branchwidth. Efficient algorithms computing optimal branch-decompositions, like we give in this paper, could therefore be the crucial factor that can make or break the application.

The study of branchwidth has not enjoyed the rich toolbox that treewidth has with its connections to k -trees, chordal graphs of maximum clique size, intersection graphs of subtrees of a tree etc. We try to rectify this in the current paper, by introducing various new tools like k -troikas, k -good chordal graphs and good subtree representations, whose definitions will follow later. To give an example

* Research conducted while the second author was on sabbatical at LIRMM.

using only standard terminology, we remark that using these tools we arrive at a succinct expression of the common basis of treewidth and branchwidth: For any $k \geq 2$ a graph G on vertices v_1, v_2, \dots, v_n has branchwidth at most k (treewidth at most $k - 1$) if and only if there is a cubic tree T with subtrees T_1, T_2, \dots, T_n such that if v_i and v_j adjacent then subtrees T_i and T_j share at least one edge (node) of T , and each edge (node) of T is shared by at most k of the subtrees (replace underlined words by the words in parenthesis.)

The understanding of branchwidth of special graph classes is relatively limited. We give a brief overview of the literature. In a paper from 1994 Seymour and Thomas showed that branchwidth is NP-complete in general, and followed this by their celebrated ratcatcher method computing branchwidth of planar graphs in polynomial time [15]. In 1997 Bodlaender and Thilikos used fairly brute-force methods to give a linear-time algorithm deciding if a graph has branchwidth at most some constant k [1] and a very elegant algorithm for graphs of branchwidth 3 [2]. Then in 1999 Kloks, Kratochvil and Müller [12,11] pushed into new territory by showing that branchwidth is NP-complete already for split graphs and bipartite graphs, with the bulk of their paper being an $O(n^3 \log n)$ algorithm for branchwidth of interval graphs with the comment that "it is somewhat surprising that this algorithm is by no means straightforward and its correctness proof requires a nontrivial proof." In contrast, using our branchwidth tools for the case of interval graphs we arrive at a straightforward $O(n^2)$ algorithm whose self-contained correctness proof is easy to follow. In fact, our algorithm has runtime $O(m + n + q^2)$ for an interval graph on m edges, n vertices and q maximal cliques. In a recent investigation Mazoit gave a polynomial-time algorithm for branchwidth of circular-arc graphs and conjectured that branchwidth can be computed in polynomial-time for chordal graphs given with a clique tree having a polynomial number of subtrees [13]. We prove his conjecture in this paper. Indeed, it follows by a generalization of the interval graph algorithm since we show a structural property stating that branchwidth of a chordal graph with clique tree T can be found by considering chordal supergraphs whose clique tree is a minor of T .

In Section 2 we give some standard definitions. In Section 3 we use subtree-representations to characterize graphs of branchwidth k as subgraphs of chordal graphs. In Section 4 we study the central new concept of k -troikas in a purely set-theoretic setting. In Section 5 we give a simple algorithm computing branchwidth for interval graphs and more generally for chordal graphs with a clique tree having a polynomial number of subtrees.

2 Standard Definitions

We consider simple undirected and connected graphs G with vertex set $V(G)$ and edge set $E(G)$. We denote G subgraph of H by $G \subseteq H$ which means that $V(G) = V(H)$ and $E(G) \subseteq E(H)$. For a set $A \subseteq V(G)$, $G(A)$ denotes the subgraph of G induced by the vertices in A . A is called a *clique* if $G(A)$ is complete. The set of neighbors of a vertex v in G is $N(v) = \{u \mid uv \in E(G)\}$. A vertex set $S \subset V(G)$ is a *separator* if $G(V(G) \setminus S)$ is disconnected. Given two vertices u and v , S is a *u, v -separator* if u and v belong to different connected

components of $G(V(G) \setminus S)$. A u, v -separator S is *minimal* if no proper subset of S separates u and v . In general, S is a *minimal separator* of G if there exist two vertices u and v in G such that S is a minimal u, v -separator. A graph is *chordal* if it contains no induced cycle of length ≥ 4 . In a *clique tree* of a chordal graph G the nodes are in 1-1 correspondence with the maximal cliques of G and the set of nodes whose maximal cliques contain a given vertex form a subtree. For further terminology, see e.g. [10]. We usually refer to nodes of a tree and vertices of a graph.

A *branch-decomposition* (T, μ) of a graph G is a tree T with nodes of degree one and three only, together with a bijection μ from the edge-set of G to the set of degree-one nodes (leaves) of T . For an edge e of T let T_1 and T_2 be the two subtrees resulting from $T \setminus \{e\}$, let G_1 and G_2 be the graphs induced by the edges of G mapped by μ to leaves of T_1 and T_2 respectively, and let $mid(e) = V(G_1) \cap V(G_2)$. The width of (T, μ) is the size of the largest $mid(e)$ thus defined. For a graph G its *branchwidth* $bw(G)$ is the smallest width of any branch-decomposition of G .¹

3 Good Subtree-Representations

Definition 1. A subtree-representation $R = (T, \{T_1, T_2, \dots, T_n\})$ is a pair where T is a tree with vertices of degree at most three and T_1, T_2, \dots, T_n are subtrees of T . Its edge intersection graph $EI(R)$ has vertex set $\{v_1, v_2, \dots, v_n\}$ and edge set $\{v_i v_j : T_i \text{ and } T_j \text{ share an edge of } T\}$, while its vertex intersection graph $VI(R)$ has the same vertex set but edge set $\{v_i v_j : T_i \text{ and } T_j \text{ share a node of } T\}$. For a node u of T , we call the set of vertices $X_u = \{v_i : T_i \text{ contains } u\}$ the bag of u , and $\{X_u : u \in V(T)\}$ the bags of R .

With the above terminology we can easily move between the view of a subtree-representation R as a tree T with a set of subtrees $\{T_1, T_2, \dots, T_n\}$ or as a tree T with a set of bags $\{X_u : u \in V(T)\}$. When manipulating the latter we must simply ensure that for any vertex in $EI(R)$ the set of bags containing that vertex corresponds to a set of nodes of T inducing a subtree, i.e. a connected subgraph.

Definition 2. The edge-weight of subtree-representation $R = (T, \{T_1, \dots, T_n\})$ is the maximum, over all edges uv of T , of the number of subtrees in $\{T_1, \dots, T_n\}$ that contain edge uv . R is a good subtree-representation if $EI(R) = VI(R)$.

We are in this paper only interested in the edge intersection graphs of subtree-representations having bounded edge-weight k . We start by showing that we can restrict ourselves to good subtree-representations if we want.

Lemma 1. For any subtree-representation R of edge-weight k there exists a good subtree-representation R' of edge-weight k with $EI(R) = EI(R') = VI(R')$.

¹ The graphs of branchwidth 1 are the stars, and constitute a somewhat pathological case. To simplify we therefore restrict attention to graphs having branchwidth $k \geq 2$, in other words our statements are correct only for graphs having at least two vertices of degree more than one.

Lemma 2. *A graph G has branchwidth at most $k \Leftrightarrow$ there is a good subtree-representation R of edge-weight at most k with $G \subseteq EI(R)$.*

Proof: \Rightarrow : Take a branch-decomposition (T, μ) of G of width k , i.e. $|mid(e)| \leq k$ for each $e \in E(T)$. We construct a subtree-representation $R = (T', S)$ of edge-weight k with $G \subseteq EI(R)$. T' is constructed from T by for each leaf l of T adding a new leaf l' and making it adjacent to l . For vertex $a \in V(G)$ consider the smallest spanning subtree of T containing all leaves of T that are mapped by μ to an edge incident with a . The subtree T_a will be this subtree augmented by leaf l' for each leaf l of T that it contains. This completes the description of $R = (T', \{T_a : a \in V(G)\})$. For any two adjacent vertices $\{a, b\}$ of G we have $\mu^{-1}(l) = \{a, b\}$ for some leaf l of T , and thus the subtrees corresponding to a and b share the edge ll' of T' which implies that $G \subseteq EI(R)$. If vertex a has subtree T_a containing edge e of T , then there are edges incident with a mapped to leaves in both subtrees of T arising from deleting the edge e , and thus $a \in mid(e)$. But this means that the edge-weight of R is at most k . If R is not good then we can make it good by applying Lemma 1.

\Leftarrow : Let $R = (T, S)$ be a good subtree-representation R of edge-weight at most k with $G \subseteq EI(R)$. We construct a branch-decomposition (T', μ) of G with width k . Associate each edge ab of G with an edge e of T such that the subtrees T_a and T_b corresponding to a and b both contain e . Subdivide the tree edge e by as many new nodes as there are edges of G associated to e , thus creating for each edge ab associated to e a new tree node e_{ab} . Furthermore, add a new leaf node l_{ab} , make it adjacent to e_{ab} and set $\mu(ab) = l_{ab}$. Let T''' be the tree we have constructed so far. It contains T as a minor. Consider the smallest spanning subtree T'' of T''' having the set of leaves $\{l_{ab} : ab \in E(G)\}$. Iteratively contract edges of T'' incident to a vertex of degree two until all inner vertices have degree three. The resulting tree is T' . Note that as we constructed T' from T in stages we could at each stage have updated the subtree T_a corresponding to vertex a to a new subtree T'_a so that we would still have a subtree-representation $R' = (T', S')$ with $G \subseteq EI(R')$. For example, T'_a should contain every 'subdivision node' on a tree edge f if T_a contained f , it should contain l_{ab} for any edge ab incident with a , and it should naturally shrink if it contained a removed leaf or contracted edge. Moreover, (T', S') has edge-weight at most k since never during this process did we increase the edge-weight beyond what it was. T' has nodes of degree one and three only and μ is a bijection between its leaves and the edges of G so (T', μ) is a branch-decomposition of G . It remains to show that it has width k , i.e. that for any edge e of T' we have $|mid(e)| \leq k$. We claim that $mid(e) \subseteq \{a : T'_a \text{ contains edge } e\}$. Consider $a \in mid(e)$. There must exist two leaves l_{ab}, l_{ac} of T' , one in each of the two subtrees of $T' \setminus e$, such that $a \in \mu^{-1}(l_{ab})$ and $a \in \mu^{-1}(l_{ac})$. Since the subtree T'_a of a contains both l_{ab} and l_{ac} it must also contain e . □

We introduce the concept of k -troikas² which is a central tool in our investigation of branchwidth.

² A troika is a horse-cart drawn by three horses, and when the need arises any two of them should also be able to pull the cart.

Definition 3. A k -troika (A, B, C) of a set X are 3 subsets of X , called the three parts, such that $|A| \leq k$, $|B| \leq k$, $|C| \leq k$, and $A \cup B = A \cup C = C \cup B = X$. (A, B, C) respects S_1, S_2, \dots, S_q if any $S_i, 1 \leq i \leq q$ is contained in at least one of A, B or C .

Definition 4. A k -good chordal graph is a chordal graph in which every maximal clique X has a k -troika respecting the minimal separators contained in X .

Theorem 1. A graph G has branchwidth at most $k \Leftrightarrow G$ is subgraph of a k -good chordal graph

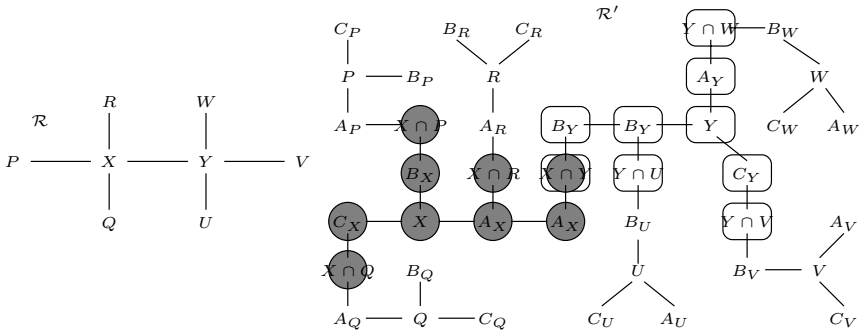


Fig. 1. On right a clique tree of a k -good chordal graph H with k -troika of any maximal clique M being (A_M, B_M, C_M) . On left, the constructed subtree-representation R' of edge-weight k such that $H \subseteq EI(R')$. The square nodes correspond to the ternary subtree associated with clique Y and the grey nodes to the ternary subtree associated to clique X . Both ternary subtrees share the leaf $X \cap Y$ where they connect.

Proof: \Rightarrow : By Lemma 2 there exists a good subtree-representation R of edge-weight k with $G \subseteq EI(R) = VI(R)$. Since $VI(R)$ is a vertex intersection graph of subtrees of a tree it is a chordal graph [9], and $H = EI(R) = VI(R)$ will indeed be our chordal graph H having G as a subgraph. By the Helly property of (vertex) intersection of subtrees of a tree, every maximal clique of H is a bag X_u for some node u of the tree. If $|X_u| \leq k$ then it clearly has a k -troika respecting any subset, so let us assume $|X_u| > k$. Since any pair a, b of nodes from X_u is adjacent in H , we must have $\{a, b\}$ contained also in one of the neighboring bags. Let the intersection of X_u and the bags of its three neighbors be A, B and C . This means that any two of A, B, C must have union X_u since if for example $a \in X_u$ but $a \notin A \cup B$ then we would be forced to have $C = X_u$, since C would have to contain a and all its neighbors in X_u contradicting the fact that R has edge-weight k . Any minimal separator S of the chordal graph H is the intersection of two maximal cliques corresponding to two bags X_u, X_v . If we assume $A = X_u \cap X_w$, for w the neighbor of u on the path from u to v in T , then we have $S = X_u \cap X_v \subseteq A$ since the subtree corresponding to a vertex $a \in (X_u \cap X_v) \setminus A$ would be disconnected.

\Leftarrow : Consider any clique tree of the k -good chordal graph H containing G . In fact this can be viewed as a pair $R = (T, S)$ just as our subtree-representations with $H = VI(R)$ and every bag inducing a maximal clique of H , except that nodes of T can have degree larger than 3. We construct from this a subtree-representation $R' = (T', S')$ of edge-weight k with $G \subseteq H \subseteq EI(R')$ which by Lemma 2 and Lemma 1 will imply that G has branchwidth at most k . Let X be a maximal clique whose node in T has q neighbors corresponding to maximal cliques Z_1, Z_2, \dots, Z_q , and let (A, B, C) be the k -troika of X respecting minimal separators $X \cap Z_1, \dots, X \cap Z_q$. This means there exists a partition P_A, P_B, P_C of $\{1, 2, \dots, q\}$ such that $X \cap Z_i \subseteq A$ for $i \in P_A$, $X \cap Z_i \subseteq B$ for $i \in P_B$, $X \cap Z_i \subseteq C$ for $i \in P_C$. For maximal clique X we construct a ternary subtree as follows: We have a central node with bag X adjacent to three paths: one path with $\max\{1, |P_A|\}$ bags A , one path with $\max\{1, |P_B|\}$ bags B and one with $\max\{1, |P_C|\}$ bags C . For each $i \in \{1, 2, \dots, q\}$ we have a leaf-node with bag $X \cap Z_i$ as neighbor of a node on these paths, e.g. if $i \in P_A$ the leaf-node should be the neighbor of a node with bag A , if $i \in P_B$ then B , and if $i \in P_C$ then C , such that q of the nodes on the 3 paths get one leaf each. (see Figure 1). Construct such a ternary subtree for each maximal clique X , i.e. for each node of T . Then, for each pair of maximal cliques X, Y that are bags of two neighboring nodes in T we identify the following two leaves into a single node: $X \cap Y$ in the subtree constructed for X and $Y \cap X$ in the subtree constructed for Y . The resulting tree T' has no node of degree more than three and together with bags as indicated it forms the subtree-representation $R' = (T', S')$. R' has edge-weight at most k since any part of a k -troika has size at most k . We show that $H \subseteq EI(R')$. For any edge $ab \in E(H)$ we have $\{a, b\} \subseteq X$ for some maximal clique X . The k -troika (A, B, C) of X has the property that any vertex $a \in X$ must be in two out of A, B, C , so that we must have $\{a, b\}$ contained in one of A, B or C . Thus the edge ab is in $EI(R')$ and $H \subseteq EI(R')$. \square

4 k -Troikas

This section will be devoted to a study of the conditions under which a set X has a k -troika respecting a given set of subsets. As with branchwidth, we restrict attention to the case $k \geq 2$. These conditions on the given sets, which will turn out to be testable by simple algorithms, will in conjunction with Theorem 1 be useful for designing algorithms computing branchwidth of graphs.

Observation 1. *If X has a k -troika respecting S_1, S_2, \dots, S_q then $|S_i| \leq k$ for each $1 \leq i \leq q$ and $|X| \leq \lfloor 3k/2 \rfloor$.*

The above is obvious, every subset must be of size at most k since it must be contained in a part of size at most k , and the fact that every pair of parts must have union X means that every element of X must belong to at least two parts which implies $2|X| \leq 3k$. Note that the case of respecting a single subset is trivial, the necessary and sufficient conditions are that the subset has at most k elements and $|X| \leq \lfloor 3k/2 \rfloor$. Likewise, if $|S_1 \cup S_2 \cup \dots \cup S_q| \leq k$ then G has a

k -troika respecting S_1, S_2, \dots, S_q precisely when $|X| \leq \lfloor 3k/2 \rfloor$ since we may as well view the union of all the subsets as a single subset.

4.1 k -Troikas Respecting Two Subsets

In this section we consider conditions under which a set X has a k -troika respecting two subsets S_1, S_2 . As mentioned above we assume that $|S_1 \cup S_2| > k$ and also wlog that any k -troika (A, B, C) respecting S_1, S_2 has $S_1 \subseteq A$ and $S_2 \subseteq B$. Note that if X has a k -troika respecting S_1, S_2 then it has one where no element of X belongs to all three parts. This motivates the following definition.

Definition 5. A k -triplartition of a set X is a partition of X into three (disjoint) partition classes, such that the sum of sizes of any two partition classes is at most k . A k -triplartition (T_1, T_2, T_3) of X respects S_1, S_2 if $S_1 \cap S_2 \subseteq T_3$, $S_1 \subseteq T_1 \cap T_3$, and $S_2 \subseteq T_2 \cap T_3$.

Observation 2. If (T_1, T_2, T_3) is a k -triplartition of X then $(T_1 \cup T_3, T_2 \cup T_3, T_2 \cup T_1)$ is a k -troika of X , and the former respects S_1, S_2 iff the latter does. Conversely, if (A, B, C) is a k -troika of X with $A \cap B \cap C = \emptyset$ then $(A \cap C, B \cap C, B \cap A)$ is a k -triplartition of X , and the former respects S_1, S_2 iff the latter does (assuming $|S_1 \cup S_2| > k$ as discussed above).

In view of this observation, when it comes to k -troikas respecting two subsets S_1, S_2 we need only consider those that arise from k -triplartitions. In Observation 1 we gave some obviously necessary conditions on $|X|, |S_1|, |S_2|$. What other necessary conditions do we have? Note that if $|X| = 3k/2$ and k is even then only a 'balanced' k -triplartition with each partition class having $k/2$ vertices will do. Since we must have $S_1 \cap S_2 \subseteq T_3$ the case where $|S_1 \cap S_2| > k/2$ therefore implies a stronger size restriction on X . The best we could hope for is to set $T_3 = S_1 \cap S_2$ and put $k - |S_1 \cap S_2|$ vertices into each of T_1 and T_2 which yields:

Observation 3. If X has a k -troika respecting S_1, S_2 then $|X| \leq |S_1 \cap S_2| + 2(k - |S_1 \cap S_2|) = 2k - |S_1 \cap S_2|$

Note that we did not need to preface this observation by the condition "if $|S_1 \cap S_2| > k/2$ " since $|X| \leq \lfloor 3k/2 \rfloor$ and $|S_1 \cap S_2| \leq k/2$ together imply $|X| \leq 2k - |S_1 \cap S_2|$. As the next theorem shows, these obviously necessary conditions are also sufficient (ONCAS).

Theorem 2. A set X has a k -troika respecting S_1, S_2 (assume $|S_1 \cup S_2| > k$) if and only if $|X| \leq \lfloor 3k/2 \rfloor$, $|S_1| \leq k$, $|S_2| \leq k$ and $|X| \leq 2k - |S_1 \cap S_2|$

Corollary 1. The smallest k such that X has a k -troika respecting S_1, S_2 is $\max\{|S_1|, |S_2|, \lfloor 2|X|/3 \rfloor, \min\{|S_1 \cup S_2|, (\lfloor |X| + |S_1 \cap S_2|)/2 \rfloor\}\}$ and can be computed in constant time given $|S_1|, |S_2|, |X|, |S_1 \cap S_2|$.

Note that $|S_1 \cup S_2|$ is easily found from $|S_1|, |S_2|, |S_1 \cap S_2|$. The two terms inside the minimum covers the two cases where the resulting smallest k -troika (A, B, C) has either $S_1 \cup S_2 \subseteq A$ or $S_1 \subseteq A$ and $S_2 \subseteq B$, respectively. Let us remark that for the interval graph algorithm the above Corollary suffices, since we then only deal with 2 minimal separators for each maximal clique.

4.2 *k*-Troikas Respecting *q* Subsets

We first consider the case of a set X respecting three subsets S_1, S_2, S_3 and denote by L the elements of X not belonging to any subset and by $U_i, 1 \leq i \leq 3$ the elements belonging to S_i only: $L = X \setminus (S_1 \cup S_2 \cup S_3), U_1 = S_1 \setminus (S_2 \cup S_3), U_2 = S_2 \setminus (S_1 \cup S_3), U_3 = S_3 \setminus (S_2 \cup S_1)$ (see Figure 2).

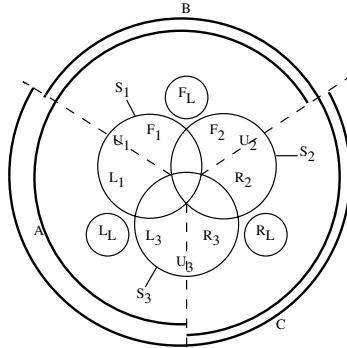


Fig. 2. Venn diagram of a set X consisting of the 6 circles $S_1, S_2, S_3, F_L, L_L, R_L$. If X has k -troika (A, B, C) respecting S_1, S_2, S_3 then we may as well require $S_1 \cap S_2 \cap S_3 = A \cap B \cap C$. The sets A, B, C are illustrated using the dotted lines at 2, 6 and 10 o'clock, e.g. A contains elements between 6 and 2 o'clock. Elements belonging to only one of the S_i sets are named U_i and further partitioned in two parts by the dotted lines.

Lemma 3. X has a k -troika A, B, C with $S_1 \subseteq A, S_2 \subseteq B, S_3 \subseteq C \Leftrightarrow$ the following system of linear equations in 5 non-negative integer variables a, b, c, d, e has a solution:

$$\begin{aligned} a &\leq |U_1|; b \leq |U_2|; c \leq |U_3|; d + e \leq |L| \\ |S_3| + |U_2| + a - b + d + e &\leq k \\ |S_1| + |U_3| + |L| + b - c - e &\leq k \\ |S_2| + |U_1| + |L| - a + c - d &\leq k \end{aligned}$$

The only other possibility is that the union of two of the subsets is at most k and in this case we may appeal to the conditions for respecting two subsets, giving:

Lemma 4. X has a k -troika respecting $S_1, S_2, S_3 \Leftrightarrow$ it has one satisfying the conditions of Lemma 3 or it has one where either $S_1 \cup S_2, S_3$ or $S_1 \cup S_3, S_2$ or $S_2 \cup S_3, S_1$ satisfies the conditions of Lemma 2.

To respect $q > 3$ subsets we simply note that since each subset must be contained in one of the three parts of the k -troika, there must exist a partition of the subsets into three classes such that every subset in the same class is contained in the same part.

Theorem 3. *X has a k -troika respecting $S_1, S_2, \dots, S_q \Leftrightarrow$ there exists a partition of $\{1, 2, \dots, q\}$ into three classes P_1, P_2, P_3 such that by Lemma 4 X has a k -troika respecting the 3 subsets $W_1 = \bigcup_{i \in P_1} S_i, W_2 = \bigcup_{i \in P_2} S_i, W_3 = \bigcup_{i \in P_3} S_i$.*

Since a set of size q has 3^q partitions into three classes we have:

Corollary 2. *In time $O(\text{poly}(|X|)3^q)$ we can decide if a set X has a k -troika respecting subsets S_1, S_2, \dots, S_q .*

5 Algorithms Computing Branchwidth

Throughout this section G is a chordal graph with m edges, n vertices, maximal cliques $\{X_1, X_2, \dots, X_q\}$, having a clique-tree T_G with nodes $\{1, 2, \dots, q\}$ such that node i corresponds to maximal clique X_i . Mazoit [13] conjectured that branchwidth is computable in polynomial-time for any chordal graph given with a clique tree having polynomially many subtrees. We will prove his conjecture, but along the way we also give a fast algorithm for the case of interval graphs, i.e. when the clique tree is a path. We first define a merged supergraph of G which is obtained by taking certain sets of maximal cliques that are connected in T_G and merging each set into a larger clique.

Definition 6. *H is a merged supergraph of G if there exists a partition of T_G into subtrees $\{H_1 \dots H_h\}$ (each node $j \in V(T_G)$ belongs to one and only one subtree H_i) such that the set of maximal cliques in H is: $\{X'_i = \bigcup_{j \in H_i} X_j\}$ ($1 \leq i \leq h$).*

It is straightforward to see that a merged supergraph H of a chordal graph G is chordal with clique-tree T_H built by making maximal cliques X'_i and X'_j adjacent iff H_i and H_j contains two adjacent nodes of T_G , in other words T_H is a minor of T_G . We first show that to find the branchwidth k of G it suffices to search for k -good chordal graphs among the merged supergraphs of G .

Lemma 5. *Let G be a chordal graph of $\text{bw}(G) = k$ and let H be a k -good chordal supergraph of G . Let X be a maximal clique of G whose neighboring maximal cliques in T_G are $X_1, X_2 \dots X_l$. If X does not have a k -troika respecting the minimal separators in X , then there exists X_i ($1 \leq i \leq l$) such that $X_i \cup X$ is a clique in H .*

Lemma 6. *A chordal graph G has $\text{bw}(G) \leq k \Leftrightarrow$ there exists a k -good chordal graph H that is a merged supergraph of G .*

Proof: \Leftarrow : By Theorem 1 the existence of a k -good chordal graph H that is a merged supergraph of G implies that $\text{bw}(G) \leq k$.

\Rightarrow : By induction on the number q of maximal cliques of G . If G has at most 2 maximal cliques, then Lemma 5 establishes the claim. Assume by induction that the property holds for any chordal graph of branchwidth k having $q \geq 2$ maximal cliques. If G is not a k -good chordal graph, then it has a maximal clique X which does not have a k -troika respecting the minimal separators $X_1 \cap$

$X, X_2 \cap X, \dots, X_l \cap X$, where $X_1 \dots X_l$ are the neighbors of X in the clique tree T_G . Since G has branchwidth k it has some k -good chordal supergraph in which, by Lemma 5, some neighbor X_j ($1 \leq j \leq l$) has been merged with X into a bigger clique. But then consider the merged supergraph of G arising from merging exactly X and X_j into one clique. It has $q - 1$ maximal cliques and by the induction hypothesis there is a k -good chordal graph H which is a merged supergraph of G' and therefore also of G . \square

5.1 Branchwidth of Interval Graphs

A graph is an interval graph iff it enjoys a *consecutive clique arrangement* (cca) that is an ordering of its maximal cliques $\mathcal{C} = (X_1, \dots, X_q)$ such that for any vertex x , the maximal cliques containing x occur consecutively. From any linear time interval graph recognition algorithm such a cca can be computed (see e.g. [3]). It is well known that for any $1 < i \leq q$, the set $S_i = X_{i-1} \cap X_i$ is a minimal separator. Let $S_1 = S_{q+1} = \emptyset$ be dummy separators. Let us denote by $X_{i,j} = \cup_{i \leq g \leq j} X_g$ ($1 \leq i \leq j \leq q$) a merged set of consecutive cliques.

Given a cca $\mathcal{C}_G = (X_1 \dots X_q)$ of an interval graph G , a merged supergraph H of G has caa $\mathcal{C}_H = (X'_1 \dots X'_h)$ with $h \leq q$ such that for any $1 \leq i \leq h$, $X'_i = X_{l_i, r_i}$ with $l_1 = 1, l_i = r_{i-1} + 1$ for $i > 1$ and $r_h = q$. Note that a merged supergraph of an interval graph is also an interval graph.

Our algorithm first computes for each pair $1 \leq i \leq j \leq q$ the smallest value $K[i, j]$ such that if we merge the consecutive cliques $X_{i,j}$ into one big clique, it will have a $K[i, j]$ -troika respecting S_i and S_{j+1} . Then by simple dynamic programming it computes the best way of merging various such sets into a merged supergraph, see Figure 3. Incrementally, in step j , we optimize over the possible cutoff points $1 \leq i \leq j$ that define the 'rightmost' merged set of cliques $X_{i,j}$. We prove correctness before considering the running time.

```

Pre-processing (see below) to find  $|S_i|, |X_i|, |S_i \cap S_j|, |X_{i,j}|$ 
For  $1 \leq i \leq j \leq q + 1$  Do Compute  $K[i, j]$  by the formula of Corollary 1
 $A[0] = 0$ 
For  $j = 1$  to  $q$  Do  $A[j] = \min\{\max\{A[i - 1], K[i, j]\} : 0 < i \leq j\}$ 
    
```

Fig. 3. Computation of $bw(G) = A[q]$ for interval graph G

Theorem 4. *The computed value $A[q]$ is the branchwidth of interval graph G .*

Proof: Let us prove by induction that, for $1 \leq i \leq q$, $A[i] = bw(G_i)$ where G_i is the graph induced by $X_{1,i}$ with an extra dummy vertex x_i adjacent to S_{i+1} . By Corollary 1 $K[i, j]$ is the minimum such that set $X_{i,j}$ has a $K[i, j]$ -troika respecting S_i and S_{j+1} . As $A[1] = K[1, 1]$, X_1 has a $A[1]$ -troika respecting S_2 . Therefore $\{x_1\} \cup S_2$ also has a $A[1]$ -troika respecting S_2 . Theorem 1 implies that $bw(G_1) = A[1]$. Assume that $A[j-1] = bw(G_{j-1})$ for $j > 1$. Let H_j be the merged

supergraph of G_j such that $bw(G_j) = bw(H_j)$. Then by Lemma 5 the maximal clique X_j is contained in H_j in a maximal clique $X' = X_{i,j}$ for some $1 \leq i \leq j$. It therefore follows from Lemma 6, that $bw(G_j) \leq \max\{A[i-1], K[i, j]\}$ for any $1 \leq i \leq j$ and thus $bw(G_j) = A[j]$. We proved that $bw(G_q) = A[q]$. Since G_q is the union of two connected components, the first one being G itself and the second an isolated vertex x_q , $bw(G) = bw(G_q)$. \square

By Corollary 1 the computation of matrices K and A takes time $O(q^2)$ if the values $|S_i|$, $|X_i|$, $|S_i \cap S_{j+1}|$, and $|X_{i,j}|$ can be accessed in $O(1)$ time. We now show that these values can be made available in array locations $S[i]$, $X[j]$, $S[i, j]$, $X[i, j]$ by pre-processing stage. Any interval graph recognition algorithm [3] is able to output in $O(n+m)$ time the size $X[i] = |X_i|$ of any maximal clique and $S[i] = |S_i|$ of any minimal separator, and also for any vertex x the range $[Left(x), Right(x)]$ of consecutive cliques containing x . From those values, assuming for any $1 \leq i \leq q$ $X[i, i] = |X_i|$, we have for $i + 1 \leq j \leq q$, $X[i, j] = X[i, j - 1] + X[j] - S[j]$. To find the values $S[i, j] = |S_i \cap S_{j+1}|$ fast, we first compute the intermediary $q \times q$ -matrix M such that for $i < j$, $M[i, j] = |(S_i \cap S_j) \setminus S_{j+1}|$. Since $|S_i \cap S_j| = \sum_{h \leq j} |(S_i \cap S_j) \setminus S_{j+1}|$, the array $S[i, j]$ can be computed as follows:

```

Initialize each entry of  $M[i, j]$  to 0;
For any  $S_i$  ( $2 \leq i \leq q$ ) and  $x \in S_i$  Do If  $Right(x) = j$  Then add 1 to  $M[i, j]$ 
For  $i = 2$  to  $q$  Do
     $S[i, q] = M[i, q]$ 
    For  $j = q - 1$  downto  $i$  Do  $S[i, j] = S[i, j + 1] + M[i, j]$ 
```

As the sum of the sizes of the minimal separators of an interval graph is bounded by m , this preprocessing requires $O(m + n + q^2)$ time. We have shown:

Theorem 5. *Branchwidth of an interval graph $G = (V, E)$ on m edges, n vertices and $q \leq n$ maximal cliques can be computed in time $O(n + m + q^2)$.*

5.2 Clique Trees with Polynomial Number of Subtrees

For a subtree T' of a tree T we define its *connection points* as the pairs of vertices $a_1b_1, a_2b_2, \dots, a_pb_p$ such that a_ib_i is an edge of T with $a_i \in T'$ and $b_i \in T \setminus T'$. Assume clique tree T_G of chordal graph G has a polynomial number of subtrees T_1, T_2, \dots, T_t , ordered by size. Let T_i have connection points $a_1b_1, a_2b_2, \dots, a_pb_p$. Define the *connection separators* of T_i to be $S_j = X_{a_j} \cap X_{b_j}$ for $1 \leq j \leq p$, where X_{a_j}, X_{b_j} are the maximal cliques of G corresponding to tree nodes a_j, b_j . Define $K[i]$ to be True if $V(T_i)$ has a k -troika respecting the connection separators S_1, S_2, \dots, S_p of T_i . The following algorithm will in polynomial time decide if G has branchwidth at most k :

Theorem 6. *For a chordal graph G given with a clique tree having a polynomial number t of subtrees the above algorithm will in polynomial time decide if branchwidth of G is at most k .*

For $i = 1$ to t **Do** Compute boolean $K[i]$ by the system of equations of Theorem 3
 $A[i] = T$ if $K[i] = T$ or if $\exists e \in E(T_i)$ with $A[e_1] = T$ and $A[e_2] = T$
for subtrees T_{e_1}, T_{e_2} of $T_i \setminus e$; otherwise $A[i] = F$

Fig. 4. Branchwidth of $G \leq k$ iff $A[t] = T$

References

1. H.L. Bodlaender and D.M. Thilikos. Constructive linear time algorithms for branchwidth. In *24th International Colloquium on Automata, Languages, and Programming (ICALP)*, Vol. 1256 of *Lecture Notes in Computer Science*, p. 627–637, 1997.
2. H.L. Bodlaender and D.M. Thilikos. Graphs with branchwidth at most three. *Journal of Algorithms*, 32:167–194, 1999.
3. K. Booth and G. Lueker. Testing of the consecutive ones property, interval graphs, and graph planarity testing using PQ-tree algorithms. *Journal of Computer and System Sciences*, 13:335–379, 1976.
4. W. Cook and P.D. Seymour. Tour merging via branch-decompositions. *Journal on Computing*, 15:233–248, 2003.
5. E. Demaine, F. Fomin, M. Hajiaghayi, and D.M. Thilikos. Fixed-parameter algorithms for (k,r) -center in planar graphs and map graphs. In *30th International Colloquium on Automata, Languages, and Programming (ICALP)*. Vol. 2719 of *Lecture Notes in Computer Science*, p. 829–844, 2003.
6. F. Fomin and D. Thilikos. Dominating sets in planar graphs: Branch-width and exponential speedup. In *14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, p. 168–177, 2003.
7. F. Fomin and D. Thilikos. A simple and fast approach for solving problems on planar graphs. In *22nd Annual Symposium on Theoretical Aspect of Computer Science (STACS)* Vol. 2996 of *Lecture Notes in Computer Science*, p. 56–67, 2004.
8. F. Fomin and D. Thilikos. Fast parameterized algorithms for graphs on surfaces: Linear kernel and exponential speedup. In *31st International Colloquium on Automata, Languages, and Programming (ICALP)*, Vol. 3142 of *Lecture Notes in Computer Science*, p. 581–592, 2004.
9. F. Gavril. The intersection graphs of subtrees in trees are exactly the chordal graphs. *Journal of Combinatorial Theory Series B*, 16:47–56, 1974.
10. M.C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Acad. Press, 1980.
11. T. Kloks, J. Kratochvil, and H. Müller. New branchwidth territories. In *16th Ann. Symp. on Theoretical Aspect of Computer Science (STACS)* Vol. 1563 of *Lecture Notes in Computer Science*, p. 173–183, 1999.
12. T. Kloks, J. Kratochvil, and H. Müller. Computing the branchwidth of interval graphs. *Discrete Applied Mathematics* 145:266–145, 2005.
13. F. Mazoit. A general scheme for deciding the branchwidth. Technical Report RR2004-34, LIP - École Normale Supérieure de Lyon, 2004.
<http://www.ens-lyon.fr/LIP/Pub/Rapports/RR/RR2004/RR2004-34.pdf>.
14. N. Robertson and P.D. Seymour. Graph minors X: Obstructions to tree-decomposition. *Journal on Combinatorial Theory Series B*, 52:153–190, 1991.
15. P.D. Seymour and R. Thomas. Call routing and the ratcatcher. *Combinatorica*, 14(2):217–241, 1994.

Treewidth Lower Bounds with Brambles^{*}

Hans L. Bodlaender¹, Alexander Grigoriev², and Arie M.C.A. Koster³

¹ Institute of Information and Computing Sciences, Utrecht University,
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
`hansb@cs.uu.nl`

² Department of Quantitative Economy, University of Maastricht,
P.O. Box 616, 6200 MD Maastricht, The Netherlands
`a.grigoriev@ke.unimaas.nl`

³ Zuse Institute Berlin (ZIB), Takustraße 7, D-14195 Berlin-Dahlem, Germany
`koster@zib.de`

Abstract. In this paper we present a new technique for computing lower bounds for graph treewidth. Our technique is based on the characterisation of the treewidth as the maximum order of a bramble of the graph. We give two algorithms: one for general graphs, and one for planar graphs. The algorithm for planar graphs is shown to give a lower bound for the treewidth that is at most a constant factor away from the exact treewidth. For both algorithms, we report on extensive computational experiments that show that the algorithms give often excellent lower bounds, in particular when applied to (close to) planar graphs.

1 Introduction

Motivation. In many applications of the notion of treewidth, it is desirable that we can compute tree decompositions of small width of given graphs. Unfortunately, finding a tree decomposition of optimal width and determining the exact treewidth are NP-hard; see [3]. Much research has been done in recent years on the problem to determine the treewidth of the graph: this includes a faster exponential time algorithm [17], a theoretically optimal but due to the large constant factor hidden in the O -notation impractical linear time algorithm for the fixed parameter case [5], a polynomial time algorithm for graphs with polynomially many minimal separators [11], a branch and bound algorithm [18], preprocessing methods [9,8], upper bound heuristics (see e.g., [2,13,21]), and lower bound heuristics. An overview with many references can be found in [6].

In this paper, we focus on lower bound methods for treewidth. Lower bound algorithms are interesting and useful for a number of different reasons. When running a branch and bound algorithm to compute the treewidth of a graph (see e.g., [18]), a good lower bound helps to quickly cut off branches. Lower bounds

^{*} This work was partially supported by the Netherlands Organisation for Scientific Research NWO (project *Treewidth and Combinatorial Optimisation*) and partially by the DFG research group "Algorithms, Structure, Randomness" (Grant number GR 883/9-3, GR 883/9-4).

inform us on the quality of upper bounds. Also, a high lower bound can tell that we should not aim for a solution of a problem on a certain graph instance with treewidth techniques: Suppose we decide we want to solve a certain problem on a given graph with a dynamic programming algorithm on a tree decomposition. If we have a large lower bound for the treewidth of that graph, we know in advance that this dynamic programming algorithm will use much time, and hence we should direct our attention to trying different methods.

In recent years, several treewidth lower bound methods have been found and evaluated. A trivial lower bound for the treewidth is the minimum degree of a vertex. Better lower bounds are obtained by looking at the minimum degrees of induced subgraphs or of graphs obtained by contractions (see [10]). Often slightly better than the minimum degree is a lower bound by Ramachandramurthi [24], which is (for non-complete graphs) the minimum over all pairs of non-adjacent vertices v, w , of the maximum degree of v and w . Another lower bound, found by Lucena [23], and analysed in [7] is based on maximum cardinality search. Combining these bounds with contractions gives often considerable improvements [10,22]. Significant improvements can be obtained by using these techniques in combination with a method introduced by Clautiaux et al. [12], based upon adding edges between vertices that have many common neighbours or disjoint paths between them.

The experiments carried out in [10,22] show that for several graphs, the existing lower bound methods give good bounds that are often close and in several cases equal to the known upper bounds. However, there are also several instances where each of these methods yields rather small lower bounds that are far away from the real treewidth. Instances of this type are often planar graphs, or graphs that are in a certain sense close to being planar, e.g., graphs obtained by taking the union of a small number of TSP-tours on a point set in the plane (see [14]). The reason that the known techniques appear to fail for these instances probably is due to the fact that they — in a certain sense — are all degree-based, and planar graphs always have vertices of small degree, cf. [30]. Thus, we were searching for a treewidth lower bound method using a different principle that works well for graphs that are planar or close to planar, or have in some other sense ‘not many high degree vertices’. In this paper, we present such a different method, based on the notion of *bramble* (for the first time, brambles appeared in [27] with the name *screens*).

Notations and Techniques. Let $G = (V, E)$ be a graph. Two subsets of V are said to *touch* if they have a vertex in common or E contains an edge between them. Reed [25] calls a set \mathcal{B} of mutually touching connected vertex sets a *bramble*. A subset of V is said to *cover* \mathcal{B} if it is a hitting set for \mathcal{B} (i.e. a set which intersects every element of \mathcal{B}). The *order* of a bramble \mathcal{B} is the minimum size of a hitting set for \mathcal{B} . The *bramble number* of G is the maximum order of all brambles of G . The relationship between the bramble order and the treewidth was obtained by Seymour and Thomas [27]:

Theorem 1 (Seymour and Thomas [27]). *Let k be a non-negative integer. A graph has treewidth k if and only if it has bramble number $k + 1$.*

For a short proof of this theorem we refer to Bellenbaum and Diestel [4].

So, finding a high order bramble immediately implies getting a good lower bound for the treewidth. In this paper, we give algorithms that construct brambles and compute their orders: in Section 2 for general graphs; and in Section 3 for planar graphs. We finish the paper presenting the computational results.

A graph H is a *minor* of G , if H can be obtained from G by a series of zero or more vertex deletions, edge deletions, and edge contractions. We use several standard graph theoretic notions, and skip the definitions here, including those of tree decomposition and treewidth.

2 Brambles in General Graphs

Algorithm \mathcal{A}_1 gets as input an arbitrary undirected graph G , finds several brambles of G or minors of G , and outputs the bramble with largest order thus found.

Preprocessing.

1. Take an arbitrary vertex r in V . Define $V_0 := \{r\}$. Let us refer to this set as to the level 0 set of vertices.
2. Suppose that $k \geq 1$ preprocessing steps have been done and k level sets V_0, V_1, \dots, V_{k-1} were defined. We define level k set V_k as follows. Consider the vertices $V'_k \subseteq V \setminus \{\bigcup_{i=0}^{k-1} V_i\}$ adjacent to the vertices from V_{k-1} . If those vertices form a connected subgraph then $V_k := V'_k$. Otherwise, let us find a connectivity closure of V'_k , i.e. a subset $V''_k \subseteq V \setminus \{\bigcup_{i=0}^{k-1} V_i\}$ such that the graph induced by $V'_k \cup V''_k$ is connected. This can be done, for instance, by adding to initially empty set V''_k all shortest paths between the components in V'_k . If there is no connectivity closure for the set V'_k then redefine $V_{k-1} := V \setminus \{\bigcup_{i=0}^{k-2} V_i\}$ and go to step 3 of preprocessing. Notice, that the described procedure is a classical Breadth-First-Search (BFS) extended to taking connectivity closures. Let the number of obtained level sets be $R - 1$.
3. Add to the graph a dummy vertex q ; connect all vertices from the level $R - 1$ to q ; and define the level R set by $V_R := \{q\}$. For simplicity, let us call the level sets simply by levels.

Basic Step. For all $0 \leq i < j \leq R$ we do the following.

1. Contract the first i levels V_0, \dots, V_{i-1} and the last $R - j$ levels V_{R-j+1}, \dots, V_R into vertices s_i and t_j respectively. Denote the resulting graph by $G_{i,j}$
2. Find a minimum (s_i, t_j) -cut in $G_{i,j}$ by max-flow techniques; see [1]. Let $c_{i,j}$ be the cardinality of such a cut.
3. In $G_{i,j}$ find $c_{i,j}$ vertex disjoint (s_i, t_j) -paths P'_ℓ , $\ell = 1, \dots, c_{i,j}$, delete from each of these paths vertices s_i and t_j , and denote the resulting paths by P_ℓ .
4. Define $\mathcal{B}_{i,j}$ as follows. Let $\{s_i\}$ be an element of $\mathcal{B}_{i,j}$. For all $i \leq k \leq R - j$ and $1 \leq \ell \leq c_{i,j}$ let the set $V_k \cup P_\ell$ be an element of $\mathcal{B}_{i,j}$.

Output. Output the maximum cardinality set $\mathcal{B}_{i,j}$, $0 \leq i < j \leq R$. STOP.

Theorem 2. *Set $\mathcal{B}_{i,j}, 0 \leq i < j \leq R$, is a bramble of $G_{i,j} \setminus \{t_j\}$. The order of this bramble equals $\min\{c_{i,j} + 1, R - j - i + 2\}$.*

Proof. Take any pair i, j such that $0 \leq i < j \leq R$ and consider the corresponding graph $G_{i,j}$. Let us check whether all conditions in the definition of brambles are satisfied for the set $\mathcal{B}_{i,j}$.

First of all, let us verify that each element of $\mathcal{B}_{i,j}$ forms a connected set. From preprocessing we know, that each level is a connected set. The paths constructed in the basic step of \mathcal{A}_1 are clearly connected and each of those paths crosses all levels in $G_{i,j}$. Hence a union of any level and any path forms a connected set.

Secondly, all elements of $\mathcal{B}_{i,j}$ are mutually touching: all elements are touching $\{s_i\}$, and again all paths from the basic step in \mathcal{A}_1 pass through all levels in $G_{i,j}$. Therefore, $\mathcal{B}_{i,j}$ is a bramble for $G_{i,j}$.

Clearly, the min-cut together with s_i forms a cover for the constructed bramble. A set of representative vertices, one from each level $k, i \leq k \leq R - j$, together with s_i also form a cover for the constructed bramble. Thus, the order of the bramble is at most $\min\{c_{i,j} + 1, R - j - i + 2\}$. Since the levels are non-intersecting, the paths constructed at the basic step of \mathcal{A}_1 are vertex disjoint, and the bramble contains all combinations of these levels and paths, the order of the bramble is at least $\min\{c_{i,j} + 1, R - j - i + 2\}$, which completes the proof. \square

Theorem 3. *The lower bound $LB_1 = \max_{0 \leq i < j \leq R} |\mathcal{B}_{i,j}| - 1$ on the treewidth of a general graph G can be obtained by Algorithm \mathcal{A}_1 in time $O(n^3m)$, where $|\mathcal{B}_{i,j}|$ is the order of bramble $\mathcal{B}_{i,j}$, n is the number of vertices in G , and m is the number of edges in G .*

Proof. By Theorem 1 the order of any bramble is at most the treewidth of the graph plus one. By Theorem 2, Algorithm \mathcal{A}_1 finds a bramble $\mathcal{B}_{i,j}$ for each graph $G_{i,j}$. Since $G_{i,j}$ is a minor of G , we derive that $LB_1 = \max_{0 \leq i < j \leq R} |\mathcal{B}_{i,j}| - 1$ is a lower bound for the treewidth of graph G .

Computing the partition of V into the level sets requires at most R times calling an all-pairs shortest paths subroutine. Such a subroutine can be implemented in $O(mn)$ time; see [1]. Thus, the preprocessing requires at most $O(Rnm)$ time.

In the basic step of the algorithm we compute c vertex disjoint (s, t) -paths in a graph, which can be done in $O(nm)$ time; see [1]. Together with enumeration over all possibilities for i and j , it brings the time complexity of the basic step up to $O(R^2nm)$. Since $R \leq n$, we have that the total running time of \mathcal{A}_1 is at most $O(n^3m)$, as required. \square

At cost of an additional multiplicative factor of n we can find a root vertex r for the BFS that provides the best lower bound for the treewidth, cf. Section 4.

3 Brambles in Planar Graphs

Algorithm \mathcal{A}_1 can be significantly improved when the input graph is restricted to be planar. Consider the following Algorithm \mathcal{A}_2 that finds brambles in several minors of a connected planar graph G .

Input. A planar embedding of G with no edge crossings. It is well known that such an embedding can be constructed in linear time. Without loss of generality we assume that the exterior face of this embedding is a simple cycle containing at least three vertices (G is a simple graph without parallel edges).

Preprocessing. Let *North*, *East*, *South*, and *West* be four simple paths (possibly atomic) belonging to the exterior face such that *North* has one common endpoint with *East* and one common endpoint with *West*, and so has *South*. Moreover, let the lengths of these four paths be roughly the same, i.e. the length may vary by at most one vertex. Notice that such paths always exist and can be found in linear time. We add in the exterior four dummy vertices $\mathcal{N}, \mathcal{E}, \mathcal{S}$, and \mathcal{W} and connect them to all vertices in *North*, *East*, *South*, and *West* respectively. Further in the paper we always refer to the vertices incident to $\mathcal{N}, \mathcal{E}, \mathcal{S}, \mathcal{W}$ as to *North*, *East*, *South* and *West*, respectively.

Basic Step. We view the algorithm as a rooted tree with a root corresponding to the graph constructed in the preprocessing. At each node of the tree we perform the following steps.

1. Given a node i in the tree and the planar graph G_i associated with this node. Let c_i denote the size of a minimum $(\mathcal{N}, \mathcal{S})$ -cut in graph $G_i \setminus \{\mathcal{E}, \mathcal{W}\}$ and d_i the size of a minimum $(\mathcal{W}, \mathcal{E})$ -cut in $G_i \setminus \{\mathcal{N}, \mathcal{S}\}$. Find c_i vertex disjoint paths connecting \mathcal{N} and \mathcal{S} , and d_i vertex disjoint paths connecting \mathcal{W} and \mathcal{E} . Denote $b_i = \min\{c_i, d_i\}$. Clearly, $G_i \setminus \{\mathcal{N}, \mathcal{E}, \mathcal{S}, \mathcal{W}\}$ contains a $b_i \times b_i$ grid as a minor. We create an order b_i bramble \mathcal{B}_i for this $b_i \times b_i$ grid taking the set of all crosses in the grid; see [4].
2. If $b_i = c_i$, we create the child nodes of i as follows. The $(\mathcal{N}, \mathcal{S})$ -cut C of cardinality c_i specifies two disconnected subsets $V_i^{\mathcal{N}}$ and $V_i^{\mathcal{S}}$ of vertices in $G_i \setminus C$ where $V_i^{\mathcal{N}}$ is connected to \mathcal{N} and $V_i^{\mathcal{S}}$ is connected to \mathcal{S} . To define the first child node of i we contract in G_i the connected subgraph induced by $\{\mathcal{N}\} \cup V_i^{\mathcal{N}} \cup C$ into a vertex \mathcal{N} . After this contraction, edges $(\mathcal{N}, \mathcal{W})$ and $(\mathcal{N}, \mathcal{E})$ will appear, see Figure 1 (a)-(b).

We remove those edges from the graph. Clearly, after the contraction, *West* and *North*, and *East* and *North*, have no endpoints in common. We add one edge from \mathcal{W} to the exterior of the graph and one edge from \mathcal{E} to the exterior of the graph such that *North* will again has one common endpoint with *West* and one common point with *East*, see Figure 1 (c). Let the resulting graph $G_i^{\mathcal{N}}$ be the first child node of i .

The second child node we obtain similarly by contraction of the subgraph induced by $\{\mathcal{S}\} \cup V_i^{\mathcal{S}} \cup C$ into a vertex \mathcal{S} . The connected components of $G_i \setminus C$ different from $V_i^{\mathcal{N}}$ and $V_i^{\mathcal{S}}$ form also the child nodes of the tree. For those child nodes we perform the preprocessing again to establish the dummy vertices.

3. If $b_i = d_i$, we similarly create the child nodes with respect to *East-West*.
4. We recurse on the child nodes unless the number of non-dummy vertices in the node graph becomes less or equal to the largest value b_i observed by the algorithm (so far).

Output. Output the maximum cardinality bramble \mathcal{B}_i . STOP.

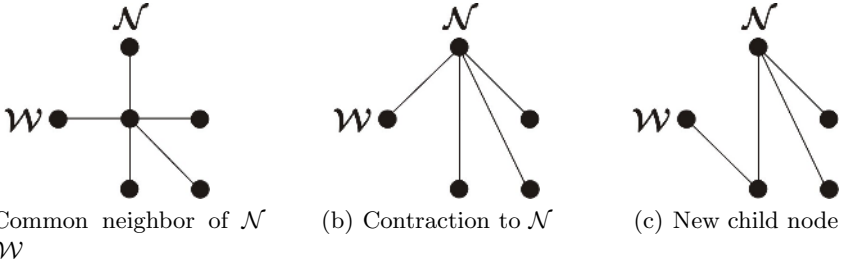


Fig. 1. Child node creation

Theorem 4. *The lower bound $LB_2 = \max_i b_i$ on the treewidth of a planar graph can be obtained by Algorithm \mathcal{A}_2 in time $O(n^2 \log n)$.*

Proof. We already observed that for any node i in the tree, b_i is a side size of a square grid minor in G . Since the treewidth of a square grid is its side size we directly have that $LB_2 = \max_i b_i$ is a lower bound on the treewidth of G .

Notice that the preprocessing requires only linear time. In the basic step of the algorithm we compute two minimum cuts in a planar network, which can be done in $O(n \log n)$ time; see [1]. Since the number of nodes in the tree is $O(n)$, performing the basic step on all $O(n)$ nodes, we have the total running time of \mathcal{A}_2 at most $O(n^2 \log n)$, as required. \square

It is noticeable that Algorithm \mathcal{A}_2 , besides estimation of the treewidth, approximates another parameter of the planar graph, namely the side size of the largest grid minor. It is well known that graphs having a large treewidth must have also a large grid as a minor; see, e.g., [16,15,26]. The following theorem and corollary present the algorithmic consequences of this fact.

Theorem 5. *For any planar graph G , the lower bound on the treewidth returned by Algorithm \mathcal{A}_2 satisfies inequality $\frac{S-16}{4} \leq LB_2 \leq S$, where S is a side size of the largest grid minor in G , and these bounds for LB_2 are tight.*

Proof. By construction, LB_2 is the side size of a grid minor of G . Since S is the side size of the largest grid minor, the inequality $LB_2 \leq S$ always holds. Thus, it remains to prove the lower bound for LB_2 .

We prove the bound by contradiction. Assume $LB_2 < \frac{S-16}{4}$. Let M be the largest square grid minor in G , thus having the side size S . From the fact that Algorithm \mathcal{A}_2 processes the child nodes until they contain at most $b_i \leq LB_2$ non-dummy vertices and from the assumption that $LB_2 < \frac{S-16}{4} \ll S^2$, there is a node in the tree such that M is cut by *North*, *East*, *South*, and *West*. Assume $|North| \geq S/4$ and there is an interior vertex of M belonging to *North*. Then we immediately derive that there is a (grand-) parent node j in the tree such that the value b_j calculated by Algorithm \mathcal{A}_2 in that node is at least $\frac{S}{4} - 4 = \frac{S-16}{4}$ yielding $LB_2 \geq \frac{S-16}{4}$. The same we derive for *East*, *South* and *West*. Therefore, none of the cuts, *North*, *East*, *South*, or *West*, is such that it contains an interior vertex of M and the cardinality of the cut is at least $S/4$.

Since $\mathcal{T} = North \cup East \cup South \cup West$ forms a cut of M and in the interior of M this total cut has at most $S - 1$ points, \mathcal{T} can cut off at most $S^2/2$ points from M . Therefore, there is a branch of the tree where all node graphs contain at least $S^2/2 \gg \frac{S-16}{4}$ vertices. Since this holds also for a leaf in this branch, we derive that there must exist a node i in the tree with value b_i returned by Algorithm \mathcal{A}_2 at least $S^2/2 \geq \frac{S-16}{4}$ yielding the desired contradiction.

To prove the asymptotical tightness of the bound, consider an $S \times S$ grid. Let the dummy nodes be connected to one side of the grid as depicted on Figure 2. At

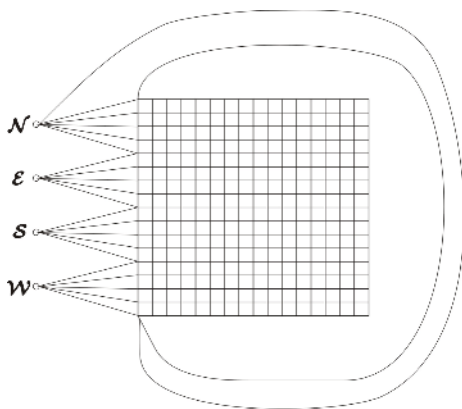


Fig. 2. Tightness of the lower bound

the first basic step of the algorithm both minimum cuts, $(\mathcal{N}, \mathcal{S})-$ and $(\mathcal{W}, \mathcal{E})-$, have cardinality $S/4$. Proceeding the algorithm further on does not improve the lower bound obtained at the first basic step. Therefore, on the given instance the algorithm outputs the lower bound $LB_2 = S/4$.

Tightness of the upper bound on LB_2 follows directly from the consideration of $S \times S$ grid when each dummy node is connected to one side of the grid dedicated especially to this dummy node. In this case the algorithm outputs $LB_2 = S$ recorded at the first basic step of the algorithm. \square

Notice that at cost of additional factor of n in the running time we can improve Algorithm \mathcal{A}_2 by "rotating" *North, East, South, and West*, i.e., choosing the best possible combination of common endpoints of the paths.

From Theorem 5 and the result by Robertson, Seymour, and Thomas [26] that every planar graph of treewidth $tw(G)$ has an $\Omega(tw(G)) \times \Omega(tw(G))$ grid graph as a minor, we deduce the following corollary.

Corollary 1. *Algorithm \mathcal{A}_2 is a constant approximation algorithm for the treewidth and for the branchwidth on planar graphs.*

Proof. From Theorem 5 we have that $S = \Theta(LB_2)$. From the result by Robertson, Seymour, and Thomas [26], we have that $tw(G) = \Theta(bw(G)) = \Theta(S) = \Theta(LB_2)$, where $bw(G)$ is a branchwidth of graph G . \square

We complete this section with a brief discussion of advantages and disadvantages of Algorithm \mathcal{A}_2 in comparison with the known approximation algorithms for the treewidth on planar graphs. The earliest and the most studied algorithm was proposed by Seymour and Thomas in [28]. The algorithm runs in $O(n^2)$ time and has performance ratio $3/2$. The biggest disadvantage of the Seymour and Thomas algorithm is that it is not memory friendly and even for the medium size graphs it easily runs out of memory; see [19,20]. Recently, Hicks in [20] made several attempts to get a memory friendly algorithm based on the Seymour and Thomas ideas. He derived two memory friendly algorithms with running time $O(n^3)$. Clearly, Algorithm \mathcal{A}_2 is also a memory friendly algorithm and it has nearly the same running time as Seymour and Thomas' algorithm, and better running time than the Hicks' algorithm. The disadvantage of Algorithm \mathcal{A}_2 is the worse performance ratio in comparison to the other algorithms. On the other hand, Algorithm \mathcal{A}_2 has another advantage, namely, that this is a polynomial time 4-approximation algorithm for finding the largest square grid minor in a planar graph.

4 Computational Experiments

Algorithms \mathcal{A}_1 and \mathcal{A}_2 have been implemented in C++ as to compare their quality with previously studied treewidth lower bounds in practice. In this section we report on the obtained results for two selected sets of instances. The first set contains a number of general graphs, that have been used in previous studies [10,22] and originate from different applications like probabilistic networks, frequency assignment, and vertex coloring. The second set of instances consists of planar graphs that have been used by Hicks [19,20] before. From both sets we selected some instances that are representative for the whole set and/or show an interesting behaviour. The CPU times reported are in seconds and obtained on a Linux-operated PC with 3.0 GHz Intel Pentium 4 processor.

Algorithm \mathcal{A}_1 for general graphs has been tested on both the selected planar and non-planar graphs. The algorithm is enhanced by recording the maximum $|\mathcal{B}_{i,j}|$ so far and testing whether following $G_{i,j}$ can beat this maximum.

As pointed out before, the maximum can be increased further by taking all vertices as root vertex r once. The additional $O(n)$ complexity of the algorithm can be reduced in practice by sorting or limiting the number of root vertices. In principle the algorithm has to be executed for only one of the vertices for which the maximum is achieved. However, we cannot select on this value before computing it. Experiments have shown that the *eccentricity* of a vertex is a reasonable criterion to sort/limit the root vertices. The eccentricity $\varepsilon(v)$ of a vertex v is the maximum depth of a breadth first search with v as root, cf. [29] and hence the best lower bound with root r is limited by $\varepsilon(r)$. Figure 3 shows for two planar graphs the eccentricity and LB_1 for all possible root vertices, sorted first according to non-increasing eccentricity and second to non-increasing LB_1 . If the best bound achieved so far is at least $\varepsilon(r)$ for some $r \in V$, we do not have to run algorithm \mathcal{A}_1 with r as root. By sorting the vertices according to

non-increasing eccentricity, the number of root vertices for which the algorithm should be executed is limited in our computations this way. Figure 3 in fact shows that the computation times can be reduced further by limiting the number of root vertices to those with high eccentricity.

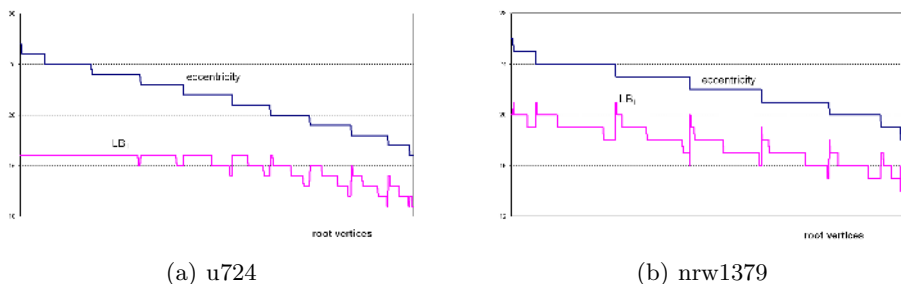


Fig. 3. LB_1 and eccentricity for all possible root vertices

Table 1 reports the results for non-planar graphs, in comparison with a contraction degeneracy $\delta C(G)$ lower bound [10]. Different behaviour can be observed: for the graphs originating from probabilistic networks, frequency assignment, and coloring, LB_1 is outperformed by the contraction degeneracy, both in time and value. For the graphs originating from a solution approach for the traveling salesman problem, LB_1 is significantly higher than $\delta C(G)$. It is known that these graphs are *close to* planar, which restricts the contraction degeneracy to exceed small values (i.e., $\delta C(G) \leq 5 + \gamma(G)$, where $\gamma(G)$ is the genus of G [30]). As was hoped for, the new lower bound turns out to be profitable

Table 1. Results for algorithm \mathcal{A}_1 on selected non-planar graphs

instance	$ V $	$ E $	δC	CPU	LB_1	CPU	UB
link	724	1738	11	0.02	7	89.95	13
munin1	189	366	10	0.00	4	2.19	11
munin3	1044	1745	7	0.01	4	195.35	7
pignet2	3032	7264	38	0.12	5	3456.77	135
celar06	100	350	11	0.00	3	1.50	11
celar07pp	162	764	15	0.01	3	19.40	18
graph04	200	734	20	0.02	5	0.38	55
school1	385	19095	122	0.59	3	46.22	188
school1-nsh	352	14612	106	0.39	3	45.34	162
zeroin.i.1	126	4100	50	0.04	2	0.30	50
fl3795-pp	1433	3098	6	0.04	6	1501.53	13
fnl4461-pp	1528	3114	5	0.05	14	4703.67	33
pcb3038-pp	948	1920	5	0.03	12	383.77	25
rl5915-pp	863	1730	5	0.02	10	470.76	23
rl5934-pp	904	1800	5	0.02	12	378.17	23

for exactly those instances, closing the gap to the best known upper bound UB substantially. Table 2 shows the results for planar graphs. The same behaviour as for the close to planar graphs can be observed.

Table 2. Results for algorithms \mathcal{A}_1 and \mathcal{A}_2 on selected planar graphs

instance	$ V $	$ E $	δC	CPU	LB_1	CPU	LB_2	CPU	LB_2R	CPU	$\beta(G) - 1$
d1655	1655	4890	5	0.04	20	995.62	17	13.76	18	574.05	28
d2103	2103	6290	5	0.07	23	1331.97	24	22.21	24	276.97	28
nrv1379	1379	4115	5	0.04	21	567.85	20	9.56	22	74.30	30
pr1002	1002	2972	5	0.03	16	605.80	17	4.69	17	57.55	20
pr2392	2392	7125	5	0.07	21	16391.82	19	21.46	20	417.70	28
tsp225	225	622	5	0.01	10	15.68	9	0.43	9	7.88	11
u2152	2152	6312	5	0.06	23	60192.11	23	12.53	23	1069.80	30
u2319	2319	6869	5	0.06	41	2625.04	31	25.64	33	1011.65	43
u724	724	2117	5	0.02	16	550.48	14	2.98	14	58.46	17

Table 2 also shows the lower bound LB_2 computed by algorithm \mathcal{A}_2 . As initial outer face, the longest face of the computed planar embedding is taken, and partitioned in (roughly) equally sized parts *North*, *East*, *South*, and *West*. Comparing LB_1 and LB_2 , there is no clear winner. In some cases LB_2 is better than LB_1 , but more often it is slightly worse. The computation time of LB_2 is however significantly less than that of LB_1 , which could be of importance if the bound is incorporated in a branch and bound approach.

The 3/2-approximation algorithm of Seymour and Thomas [28] for planar graphs computes in fact the branchwidth $\beta(G)$. It is well-known that the $\beta(G) - 1$ is a lower bound on the treewidth. The values reported in Table 2 are taken from Hicks [19]. In all cases, this lower bound is higher than LB_1 and LB_2 , as is the computational effort, cf. [19].

As pointed out in Section 3, the lower bound can be enhanced by rotation of *North*, *East*, *South*, and *West*. In column LB_2R , we report such results. A slight improvement in comparison with the case without rotation can be observed.

5 Concluding Remarks

The treewidth of a graph can be characterised by the notion of brambles, introduced by Seymour and Thomas [28]. In this work, we developed bramble construction algorithms as to bound the treewidth from below. The constructed brambles turn out to be profitable, both in theory and practice, for graphs that are (*close to*) planar. These results complement previously studied treewidth lower bounds that turned out to be good for graphs that are *far from* planar.

References

1. R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.
2. E. Amir. Efficient approximations for triangulation of minimum treewidth. In *Proceedings of the 17th Conference on Uncertainty in Artificial Intelligence*, pages 7–15, 2001.
3. S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k -tree. *SIAM J. Alg. Disc. Meth.*, 8:277–284, 1987.
4. P. Bellenbaum and R. Diestel. Two short proofs concerning tree-decompositions. *Combinatorics, Probability, and Computing*, 11:541–547, 2002.
5. H. L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25:1305–1317, 1996.
6. H. L. Bodlaender. Discovering treewidth. In P. Vojtáš, M. Bieliková, and B. Charron-Bost, editors, *SOFSEM 2005: 31st Conference on Current Trends in Theory and Practice of Computer Science*, pages 1–16. Springer-Verlag, Lecture Notes in Computer Science 3381, 2005.
7. H. L. Bodlaender and A. M. C. A. Koster. On the Maximum Cardinality Search lower bound for treewidth. In J. Hromkovič, M. Nagl, and B. Westfechtel, editors, *Proc. 30th International Workshop on Graph-Theoretic Concepts in Computer Science WG 2004*, pages 81–92. Springer-Verlag, Lecture Notes in Computer Science 3353, 2004.
8. H. L. Bodlaender and A. M. C. A. Koster. Safe separators for treewidth. In *Proceedings 6th Workshop on Algorithm Engineering and Experiments ALENEX04*, pages 70–78, 2004.
9. H. L. Bodlaender, A. M. C. A. Koster, F. v. d. Eijkhof, and L. C. van der Gaag. Pre-processing for triangulation of probabilistic networks. In J. Breese and D. Koller, editors, *Proceedings of the 17th Conference on Uncertainty in Artificial Intelligence*, pages 32–39, San Francisco, 2001. Morgan Kaufmann.
10. H. L. Bodlaender, A. M. C. A. Koster, and T. Wolle. Contraction and treewidth lower bounds. In S. Albers and T. Radzik, editors, *Proceedings 12th Annual European Symposium on Algorithms, ESA2004*, pages 628–639. Springer, Lecture Notes in Computer Science, vol. 3221, 2004.
11. V. Bouchitté and I. Todinca. Treewidth and minimum fill-in: Grouping the minimal separators. *SIAM J. Comput.*, 31:212–232, 2001.
12. F. Clautiaux, J. Carlier, A. Moukrim, and S. Négre. New lower and upper bounds for graph treewidth. In J. D. P. Rolim, editor, *Proceedings International Workshop on Experimental and Efficient Algorithms, WEA 2003*, pages 70–80. Springer Verlag, Lecture Notes in Computer Science, vol. 2647, 2003.
13. F. Clautiaux, A. Moukrim, S. Négre, and J. Carlier. Heuristic and meta-heuristic methods for computing graph treewidth. *RAIRO Oper. Res.*, 38:13–26, 2004.
14. W. Cook and P. D. Seymour. Tour merging via branch-decomposition. *Inform. J. on Computing*, 15(3):233–248, 2003.
15. E. D. Demaine and M. Hajiaghayi. Graphs excluding a fixed minor have grids as large as treewidth, with combinatorial and algorithmic applications through bidimensionality. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA2005*, pages 682–689, 2005.
16. R. Diestel, T. R. Jensen, K. Y. Gorbunov, and C. Thomassen. Highly connected sets and the excluded grid theorem. *J. Comb. Theory Series B*, 75:61–73, 1999.

17. F. V. Fomin, D. Kratsch, and I. Todinca. Exact (exponential) algorithms for treewidth and minimum fill-in. In *Proceedings of the 31st International Colloquium on Automata, Languages and Programming*, pages 568–580, 2004.
18. V. Gogate and R. Dechter. A complete anytime algorithm for treewidth. In *Proceedings of the 20th Annual Conference on Uncertainty in Artificial Intelligence UAI-04*, pages 201–208, Arlington, Virginia, USA, 2004. AUAI Press.
19. I. V. Hicks. Planar branch decompositions I: The ratcatcher. *INFORMS Journal on Computing* (to appear), 2005.
20. I. V. Hicks. Planar branch decompositions II: The cycle method. *INFORMS Journal on Computing* (to appear), 2005.
21. A. M. C. A. Koster, H. L. Bodlaender, and S. P. M. van Hoesel. Treewidth: Computational experiments. In H. Broersma, U. Faigle, J. Hurink, and S. Pickl, editors, *Electronic Notes in Discrete Mathematics*, volume 8. Elsevier Science Publishers, 2001.
22. A. M. C. A. Koster, T. Wolle, and H. L. Bodlaender. Degree-based treewidth lower bounds. In S. E. Nikolettseas, editor, *Proceedings of the 4th International Workshop on Experimental and Efficient Algorithms WEA 2005*, pages 101–112. Springer Verlag, Lecture Notes in Computer Science, vol. 3503, 2005.
23. B. Lucena. A new lower bound for tree-width using maximum cardinality search. *SIAM J. Disc. Math.*, 16:345–353, 2003.
24. S. Ramachandramurthi. The structure and number of obstructions to treewidth. *SIAM J. Disc. Math.*, 10:146–157, 1997.
25. B. A. Reed. *Tree width and tangles, a new measure of connectivity and some applications*, volume 241 of *LMS Lecture Note Series*, pages 87–162. Cambridge University Press, Cambridge, UK, 1997.
26. N. Robertson, P. D. Seymour, and R. Thomas. Quickly excluding a planar graph. *J. Comb. Theory Series B*, 62:323–348, 1994.
27. P. D. Seymour and R. Thomas. Graph searching and a minimax theorem for tree-width. *J. Comb. Theory Series B*, 58:239–257, 1993.
28. P. D. Seymour and R. Thomas. Call routing and the ratcatcher. *Combinatorica*, 14(2):217–241, 1994.
29. D. B. West. *Introduction to graph theory*. Prentice Hall, 2001.
30. T. Wolle, A. M. C. A. Koster, and H. L. Bodlaender. A note on contraction degeneracy. Technical Report UU-CS-2004-042, Institute of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, 2004.

Minimal Interval Completions

Pinar Heggernes¹, Karol Suchan², Ioan Todinca², and Yngve Villanger¹

¹ Department of Informatics, University of Bergen,
N-5020 Bergen, Norway

{pinar, yngvev}@ii.uib.no

² LIFO, Université d'Orleans, PB 6759,
F-45067 Orleans Cedex 2, France

{todinca, suchan}@lifo.univ-orleans.fr

Abstract. We study the problem of adding edges to an arbitrary graph so that the resulting graph is an interval graph. Our objective is to add an inclusion minimal set of edges, which means that no proper subset of the added edges can result in an interval graph when added to the original graph. We give a polynomial time algorithm to obtain a minimal interval completion of an arbitrary graph, thereby resolving the complexity of this problem.

1 Introduction

The class of interval graphs is an important and well-studied graph class with applications in many fields, like biology, chemistry, and archeology [5]. In addition, many problems that are NP-complete on general graphs have polynomial-time, and even linear-time, algorithms on interval graphs. Thus it is of interest to compute an interval embedding of a given graph with few added edges. An interval graph can be obtained from any graph by adding edges, and the resulting graph is called an *interval completion* of the original graph. If the added set of edges is of minimum cardinality, the resulting interval completion is called *minimum*. Unfortunately, computing minimum interval completions is an NP-hard problem [3], [6]. The *pathwidth* problem is concerned with computing an interval completion in which the size of the largest clique is minimized. Many difficult problems have efficient solutions for graphs of bounded pathwidth; however also this problem is NP-hard [7]. This has given motivation to study the problem of adding an inclusion minimal set of fill edges to a given graph to obtain an interval completion. That is, no proper subset of the added edges can give an interval completion of the original graph. In this case, the resulting interval completion is called *minimal*. Interval completions with minimum number of edges or with minimum clique size are among the minimal interval completions of the input graph. The computational complexity of minimal interval completions has been open until now. In this paper, we show that the problem can be solved in polynomial time.

Interval graphs are a subset of chordal graphs, and they are exactly the class of graphs that are both chordal and AT-free [11]. Adding edges to an arbitrary

graph so that the resulting graph is chordal, gives a chordal completion, or a *triangulation* of the input graph. Minimum and minimal triangulations are defined analogous to minimal and minimum interval completions. Computing minimum triangulations is NP-hard [14], whereas computing minimal triangulations is a well studied problem, and it motivates the study of minimal interval completions [8]. A triangulation is minimal if and only if no single added edge can be removed without destroying chordality [13]. This useful property gives algorithmic tools that have been used in several minimal triangulation algorithms. Unfortunately, the analogous statement for interval completions is not true. That is, it might be the case that no single added edge can be removed from an interval completion without destroying the interval graph property, but still the interval completion might not be minimal because there are several edges that can be removed simultaneously to give a minimal interval completion. Furthermore, examples can be constructed to show that a minimal triangulation followed by a minimal AT-free completion, or vice versa, does not necessarily result in a minimal interval completion. Consequently, the minimal interval completion problem is more difficult than the minimal triangulation problem, and it was not known until now whether minimal interval completions could be computed in polynomial time.

We use an incremental approach for some fixed ordering of vertices to compute a minimal interval completion of a given arbitrary graph G : At each step, a new vertex is taken into account and we compute a minimal interval completion of the subgraph of G induced by the vertices considered so far. For each new vertex u processed in this way, edges are added only between u and the vertices processed before. The overall time complexity of our algorithm is $O(n^4)$.

2 Preliminaries

We start this section with standard definitions, and at the end of this section we give new definitions specific to our problem and our algorithm.

We consider simple and connected input graphs. A graph is denoted by $G = (V, E)$, with $n = |V|$, and $m = |E|$. For a set $A \subseteq V$, $G[A]$ denotes the subgraph of G induced by the vertices in A . Vertex set A is called a *clique* if $G[A]$ is complete. For a vertex $v \in V$ or a subset $A \subseteq V$, we will informally use $G - v$ and $G - A$ to denote the graphs $G[V \setminus \{v\}]$ and $G[V \setminus A]$, respectively. A path is a sequence $[v_1, v_2, \dots, v_p]$ of vertices such that v_i is adjacent to v_{i+1} , for all $1 \leq i < p$. A cycle is a path such that the first and last vertices are adjacent.

The *neighborhood* of a vertex v in G is $N_G(v) = \{u \mid uv \in E\}$, and the *closed neighborhood* of v is $N_G[v] = N_G(v) \cup \{v\}$. Similarly, for a set $A \subseteq V$, $N_G(A) = \bigcup_{v \in A} N_G(v) \setminus A$, and $N_G[A] = N_G(A) \cup A$. When graph G is clear from the context, we will omit subscript G .

An edge that is added to the input graph G is called a *fill edge*, and the process of adding edges between a vertex x and a vertex set A is called *filling* A .

A vertex set $S \subset V$ is a *separator* if $G - S$ is disconnected. Given two vertices u and v , S is a *u, v -separator* if u and v belong to different connected components of $G - S$, and S is then said to *separate* u and v . A u, v -separator S is *minimal* if no proper subset of S separates u and v . In general, S is a *minimal separator* of

G if there exist two vertices u and v in G such that S is a minimal u, v -separator. It can easily be verified that S is a minimal separator if and only if $G - S$ has two distinct connected components C_1 and C_2 such that $N_G(C_1) = N_G(C_2) = S$. In this case, C_1 and C_2 are called *full components*, and S is a minimal u, v -separator for every pair of vertices $u \in C_1$ and $v \in C_2$.

A *chord* of a cycle is an edge connecting two non-consecutive vertices of the cycle. A graph is *chordal*, or equivalently *triangulated*, if it contains no induced chordless cycle of length ≥ 4 .

A graph G is an *interval graph* if continuous intervals can be assigned to each vertex of G such that two vertices are neighbors if and only if their intervals intersect. In other terms, interval graphs are the intersection graphs of subpaths of a path.

Every interval graph is also chordal. A graph is an interval graph if and only if it is chordal and AT-free [11].

Theorem 1 ([4]). *A graph G is interval if and only if there is a path (\mathcal{K}, P) whose vertex set is the set of all maximal cliques of G , such that the subgraph of (\mathcal{K}, P) induced by the maximal cliques of G containing vertex v is connected, for each vertex v of G .*

Such a path will be called a *clique path* CP_G of G . The vertices of a clique path (maximal cliques of G) are also called *bags*. Let the maximal cliques of an interval graph G be labelled $1, 2, \dots, k$, according to the order in which they appear in a clique path of G . Then, as a consequence of Theorem 1, an interval representation of G can be obtained by assigning to each vertex v the interval that consists of the labels of the maximal cliques containing v . In this way, every clique path of G is equivalent to an interval representation of G .

Due to space restrictions, all the results of this section are given without proofs. The proofs can be found in the full version of the paper [9].

Lemma 1 (see e.g. [5]). *Let G be an interval graph and let CP_G be any clique path of G . A set of vertices S is a minimal separator of G if and only if S is the intersection of two maximal cliques of G that are neighbors in CP_G . In particular, all minimal separators of G are cliques.*

We now give some definitions particularly related to our results. Let S and T be two minimal separators of G such that no one of them is a subset of the other. Suppose that T (resp. S) intersects a unique component $C_T(S)$ (resp. $C_S(T)$) of $G - S$ (resp. $G - T$) and that $C_T(S)$ (resp. $C_S(T)$) is full for S (resp. T). We define the *piece between S and T* by $P(S, T) = S \cup T \cup (C_T(S) \cap C_S(T))$.

A *block B* of G is a vertex subset such that B is the piece between $P(S, T)$ for some minimal separators S, T (in which case we say that B is a *two-block*), or $B = S \cup C$ for some minimal separator S and a full component C of $G - S$ (in this case B is a *one-block*). We say that the separators S and T (respectively S) *border* the block B .

Lemma 2. *Let B be a block of an interval graph G . Then B is the union of maximal cliques of G contained in B .*

Let us note that, for any block B of an interval graph G , and for any clique path CP_G of G , the maximal cliques contained in B form a (connected) subpath of CP_G .

Lemma 3. *Let B be a block of an interval graph G and let $CP_G = (\mathcal{K}, P)$ be any clique path of G . The set of bags corresponding to maximal cliques contained in B form a connected subpath P_B of P .*

Clearly for any clique path CP_G of G , its subpath P_B defines a clique path of $G[B]$. The converse is not true, there are clique paths of $G[B]$ which can not be extended into clique paths of G . Consider the case when B is a one-block. The next lemma characterizes the clique paths of B extendable into clique paths of the whole graph.

Lemma 4. *Let B be a block of an interval graph G .*

1. *A clique path $CP_{G[B]}$ of $G[B]$ can be extended into a clique path of G if and only if each separator bordering B is contained in a maximal clique that is a (different) endpoint of $CP_{G[B]}$.*

2. *In a clique path CP_G of G , subpath P_B can be replaced by any clique path of $G[B]$ satisfying the above property.*

Lemma 5. *Let H be a minimal interval completion of an arbitrary graph G . Let G' be a graph obtained from G by adding a new vertex x , with neighborhood $N_{G'}(x)$. There is a minimal interval completion H' of G' such that $H' - x = H$.*

Hence, for computing a minimal interval completion of G , we introduce the vertices of G one by one in the order x_1, x_2, \dots, x_n . Given a minimal interval completion of H_i of $G_i = G[\{x_1, \dots, x_i\}]$, we compute an interval completion H_{i+1} of G_{i+1} by adding vertex x_{i+1} and the edges between x_{i+1} and $N_{G_{i+1}}$ to H_i , together with a well chosen set of additional edges incident to x_{i+1} .

3 Principles of the Algorithm

From now on we consider as input an interval graph $G = (V, E)$. A new vertex x is added to G , together with a set of edges incident to x . For the rest of this document, let G' denote the graph $G+x$. We want to compute a minimal interval completion H of G' , obtained by adding edges incident to x only. We say that such a minimal interval completion *respects* G .

Take any clique path $CP_H = (\mathcal{K}, P)$ of H . By property of clique paths, the cliques containing x form a subpath P_x of P . Now, let us get back to G . Delete x from every bag in CP_H , and possibly remove the bags that do not correspond to maximal cliques of G . This yields CP_G - a clique path of G , which gives a linear ordering of maximal cliques of G . We say in this case that CP_G was obtained by *pruning* the vertex x from CP_H .

Definition 1. *An interval completion H of G' respects a clique path CP_G of G if CP_G can be obtained by pruning x from some clique path of H .*

Clearly the maximal cliques that come from P_x still form a subpath of CP_G . Our aim is to do the converse: to find a clique path $CP_G = (\mathcal{K}, P)$ of G and a subpath of CP_G in which, by adding vertex x to every bag and possibly transforming the bordering separators into new bags of H (with x contained), we obtain a minimal interval completion of G' respecting CP_G .

A clique path CP_G is called *nice* if there exists a minimal interval completion H respecting CP_G . Let K_L (resp. K_R) be the leftmost (resp. rightmost) clique of CP_G such that x has a neighbor in $K_L \setminus K_{L+1}$ (resp. $K_R \setminus K_{R-1}$) in the graph G' . Any minimal interval completion H respecting CP_G satisfies:

- $xu \in E(H)$ for every vertex u contained in a clique strictly between K_L and K_R of CP_G .
- $xv \notin E(H)$ for every vertex v that is not contained in any clique situated between K_L and K_R in the clique path.

We point out that, in some cases, it might be required to add fill edges between x and some vertices of K_L or K_R . Note that any clique path obtained from a nice one by permutations of the cliques strictly between K_L and K_R , or outside the interval between K_L and K_R will also be a nice clique path of G . We will not look for one particular clique path, but a class of them, yielding the same interval completion. Working towards finding a nice clique path of G and cliques K_L and K_R , we refine ordered partitions of the set \mathcal{K} of all maximal cliques of G .

Definition 2. Consider an ordered partition $OP = [\mathcal{K}_1, \dots, \mathcal{K}_p]$ of \mathcal{K} such that the maximal cliques in each part \mathcal{K}_i correspond to a block of G and, moreover, there exists a clique path CP_G of G such that the subpaths P_1, \dots, P_p formed by the maximal cliques in $\mathcal{K}_1, \dots, \mathcal{K}_p$ appear in this order (see also Lemma 3). Such an ordered partition is called *valid*.

Observe that, when each part of a valid ordered partition OP is a singleton, OP is exactly a clique path of G . Here, again we say that the ordered partition OP is *nice* if it can be refined into a nice clique path CP_G .

Once we obtain a nice ordered partition OP . Let \mathcal{K}_L (resp. \mathcal{K}_R) be the leftmost (resp. rightmost) part of OP such that x has a neighbor appearing in \mathcal{K}_L but not in \mathcal{K}_{L+1} (resp. appearing in \mathcal{K}_R but not in \mathcal{K}_{R-1}). We must fill all the cliques strictly between \mathcal{K}_L and \mathcal{K}_R . It remains to add a well chosen set of edges between x and blocks O_L and O_R , corresponding to the sets of maximal cliques \mathcal{K}_L and \mathcal{K}_R respectively. This will be done by refining recursively the sets of cliques \mathcal{K}_L and \mathcal{K}_R .

For obtaining the nice ordered partition OP we distinguish between two cases. In the first case, the neighborhood of x in G is a clique. We show in the full paper [9] that, if G' is not already an interval graph, is it sufficient to add edges from x to a well chosen minimal separator of G . Let us consider now the main case, when the neighborhood of x in G is not a clique.

4 The Main Case

The main and most difficult case of our algorithm is when the neighborhood of x in G' is not a clique. We denote by \mathcal{S}_x the set of all minimal a, b -separators of G , with $a, b \in N_{G'}(x)$ and non-adjacent. Clearly \mathcal{S}_x is not empty.

Theorem 2 ([1]). *Let H be any chordal supergraph of G' such that $H - x = G$. Consider a minimal a, b -separator S of G , where a and b are neighbors of x in G' . Then S is in the neighborhood of x in H .*

Theorem 3. *Let H be any interval supergraph of G' such that $H - x = G$. Any block $B = P(S_L, S_R)$ that is a piece between two separators in \mathcal{S}_x , is contained in the neighborhood of x in H' .*

The proof of Theorem 3 is given in the full paper [9].

From now on we fix a block $B = P(S_L, S_R)$ with $S_L, S_R \in \mathcal{S}_x$, if such a block exists. For each component C_j of $G - B$, let $O_j = N(C_j) \cup C_j$. Note that $N(C_j)$ is a minimal separator, so $O_j, j = 1, \dots, k$ are full one-blocks associated to separators that are subsets of S_L or S_R . Let $\mathcal{O}(B)$ denote the set of all these one-blocks associated to B , and \mathcal{B} the set $\mathcal{O} \cup \{B\}$. Notice that the blocks \mathcal{B} induce a partition of the set \mathcal{K} of all maximal cliques of G . Indeed, each $K \in \mathcal{K}$ is contained in exactly one element of \mathcal{B} . Moreover, by Lemma 3, for any clique path $CP_G = (\mathcal{K}, P)$ of G , the cliques contained in a block $BL \in \mathcal{B}$ induce a subpath of CP_G . Therefore, there is a natural correspondence between clique paths and valid ordered partitions of set \mathcal{K} induced by \mathcal{B} .

Consider such a valid ordered partition $OP = [\mathcal{K}_1, \dots, \mathcal{K}_p]$ corresponding to clique path CP_G . Instead of working with the ordered partition OP , we work with the pair (L, R) where L is the list of one-blocks of $\mathcal{O}(B)$ situated to the left of B in the clique path, ordered from left to right; R is the list of blocks to the right of B , from right to left. More formally, let \mathcal{K}_i be the part corresponding to block B (i.e., the elements of \mathcal{K}_i are exactly the maximal cliques of B). Then $L = [O_1, O_2, \dots, O_{i-1}]$ and $R = [O_p, O_{p-1}, \dots, O_{i+1}]$, where O_j is the one-block corresponding to part \mathcal{K}_j . Clearly the pair (L, R) identifies the ordered partition OP . Now given two lists L and R such that each one-block of $\mathcal{O}(B)$ is in exactly one list, we want to know if the pair (L, R) defines a valid ordered partition.

Graph G might not contain a block $B = P(S_L, S_R)$ as defined above. All the statements proved in the rest of the paper remain true when $B = P(S_L, S_R)$ is replaced by $B = S$, for any $S \in \mathcal{S}_x$.

Consider two one-blocks O_i and O_j in $\mathcal{O}(B)$. We want to know whether there exists a clique path CP_G of G in which the subpath PO_j is between PO_i and P_B . The answer is yes only if $O_i \preceq O_j$ according to the pre-order defined as follows:

Definition 3. *For every $O \in \mathcal{O}(B)$ let*
 $big(O) = \max\{K \cap B \mid K \in \mathcal{K}, K \subseteq O\}$, $small(O) = \min\{K \cap B \mid K \in \mathcal{K}, K \subseteq O\}$
Two one-blocks O_i, O_j satisfy $O_i \preceq O_j$ if $big(O_i) \subseteq small(O_j)$.

Lemma 6. *Let O_i and O_j be two one-blocks of $\mathcal{O}(B)$. There is a clique path CP_G of G such that the subpath PO_j is between PO_i and P_B only if $O_i \preceq O_j$.*

It is actually more convenient to prove the following stronger statement. It shows that this pre-order captures all possible clique paths of G .

Theorem 4. *Let L, R be two lists such that each block of $\mathcal{O}(B)$ appears in exactly one of them. Then (L, R) defines a valid ordered partition if and only if L and R are directed paths of partially pre-ordered set $(\mathcal{O}(B), \preceq)$.*

Proof. For the “only if” part, note that for two blocks O_i and O_j there exist a clique path in which P_{O_j} is between P_{O_i} and P_B only if $O_i \cap B$ is contained in every maximal clique of $G[O_i]$. Hence L and R must be chains of $(\mathcal{O}(B), \preceq)$.

Conversely, if we partition $(\mathcal{O}(B), \preceq)$ into two chains (L, R) we can construct a clique path of G in which the subpaths corresponding to one-blocks in L (resp. R) are situated to the left (resp. right) of P_B , respecting the order of the chains L, R . In this setting, we only need to get a clique path for each block $BL \in \mathcal{B}$ that lets us glue them all together into a clique path of G .

By using Lemma 4, we can construct a clique path of every $O \in \mathcal{O}(B)$ with $big(O)$ at one end. In a similar way we can get a clique path of B with separators S_L and S_R bordering B in G at the ends. The conclusion follows. \square

Our aim is to construct ordered partitions that are not only valid, but also nice. Unfortunately the pre-order $(\mathcal{O}(B), \preceq)$ is not enough to manage nice ordered partitions (and thus minimal interval completions). The reason is that, for one-blocks that are equivalent with respect to \preceq , their placement is not without impact on the minimality of the interval completion obtained.

Definition 4. We say that a one-block O is:

- clean if $O \cap N_{G'}(x) \subset B$
- hit if it is not clean
- sparable if there is an interval completion of $G'_d[O \cup \{x\}]$, constructed from $G'[O \cup \{x\}]$ by adding a dummy vertex d adjacent to x and to all the vertices in the separator bordering O , in which x is non adjacent to at least one vertex of O .

In particular, all clean blocks are sparable.

Definition 5. Let (L, R) be any valid order partition of G . An interval completion $H(L, R)$ is good for (L, R) if it respects some clique path obtained by refining (L, R) and it is inclusion minimal for this property.

The following proposition characterizes the best interval completions that can be obtained from a valid ordered partition (L, R) . The proof is given in the full paper [9].

Lemma 7. Let (L, R) define a valid ordered partition. Let O_L and O_R be the first hit blocks of L and R respectively¹. An interval completion H of G' is good for (L, R) if and only if:

1. For each block O appearing after O_L in L or after O_R in R , O is filled.
2. For each block O appearing before O_L in O or before O_R in R , O stays clean in H .

¹ If O_R is not defined, imagine it as added at the end of the list R ; the set of blocks appearing after O_R in R becomes empty, the set of blocks appearing before it is exactly R . The case when O_L is undefined is symmetrical.

3. For each $O \in \{O_L, O_R\}$, the graph $H_d[O \cup \{x\}]$ is a minimal interval completion of $G'_d[O \cup \{x\}]$. Here $H_d[O \cup \{x\}]$ is obtained from $H[O \cup \{x\}]$ by adding a dummy vertex x adjacent to x and all the vertices in the separator bordering O (see also Definition 4).

In particular, if H is a good interval completion for (L, R) and $O \in \{O_L, O_R\}$ is sparable, then it is not filled in H .

Theorem 5. *Let (L, R) define a valid ordered partition. Denote by $Spared(L, R)$ the set of one-blocks appearing before O_L in L , before O_R in R or being in $\{O_L, O_R\}$ and sparable. The valid ordered partition (L, R) is nice if and only if $Spared(L, R)$ is inclusion-maximal.*

Proof. According to Lemma 7, the set $Spared(L, R)$ is exactly the set of blocks that may not be filled in an interval completion respecting (L, R) .

Let (L, R) be such that $Spared(L, R)$ is maximal and H be a good interval completion for (L, R) . Suppose that H is strictly contained in some interval completion H' of G' . Denote by (L', R') the valid ordered partition corresponding to some path decomposition of H' . Let $O \in \mathcal{O}(B)$ such that $N_{H'}(x) \cap O$ is strictly contained in $N_H(x) \cap O$. Note that O is in $Spared(L, R)$, by maximality of this set. If O is a hit block (i.e. x has a neighbor in $O \setminus B$ in the graph G) then $O \in \{O_L, O_R\}$, contradicting Lemma 7. If O is not hit, then $N_H(x) \cap O$ is exactly the separator bordering O . By Theorem 3, this separator is filled in any interval completion of G' , leading again to a contradiction.

Conversely, let H be a minimal interval completion of G' . Suppose there exists a valid order partition (L', R') with $Spared(L, R) \subset Spared(L', R')$. Then all the blocks that are clean in H are also clean in any interval completion H' good for (L', R') . If O is sparable but hit in (L, R) , then O is O_L or O_R . Thus O is either $O_{L'}$ or $O_{R'}$. By Lemma 7 we can take H' such that $H'[O] = H[O]$. Consequently $H \subseteq H'$. Choose now a block O in $Spared(L', R') \setminus Spared(L, R)$, so O is not filled in H' . We obtain that H' is strictly contained in H , a contradiction. \square

Suppose that O_1, O_2, O_3 are equivalent with respect to \preceq , and that they represent all one-blocks that should be placed to the left of B . Let O_1 be clean, O_2 hit but sparable, and O_3 not sparable. If we put them in any order different than $1 - 2 - 3$, then the edges we add form a superset of those added by order $1 - 2 - 3$, following Theorem 5.

Let $(\mathcal{O}(B), \preceq)$ denote the pre-order $(\mathcal{O}(B), \preceq)$ refined to incorporate this differentiation.

Definition 6. *For any $O_i, O_j \in \mathcal{O}(B)$, $O_i \leq O_j$ if $O_i \preceq O_j$ and (not $O_j \preceq O_i$ or $priority(O_i) \leq priority(O_j)$), where*

$$priority(O) = \begin{cases} 1, & \text{if } O \text{ is clean;} \\ 2, & \text{if } O \text{ is hit but sparable;} \\ 3, & \text{if } O \text{ is not sparable.} \end{cases}$$

Note that one can check in polynomial time if a one-block is sparable (see also [9]).

Lemma 8. *A block O is sparable if and only if there exists a vertex $y \in O \setminus (B \cup N_{G'}(x))$ such that the graph obtained from $G'_d[O \cup \{x\}]$ by filling $O \setminus \{y\}$ is an interval graph. It can be checked in $O(nm)$ time if a given one-block $O \in \mathcal{O}(B)$ is sparable.*

The algorithm **NiceOrderedPartition** below produces a nice ordered partition (L, R) . The full proof of the algorithm can be found in [9]. Let us give here a description of the algorithm and some hints about its proof.

The algorithm relies on the pre-order \leq . We transform it into a linear order \leq_{top} by sorting the one-blocks in topological order (the permutation of elements in the same equivalence class does not affect the minimality of interval completions obtained). The graph (\mathcal{O}, \preceq) together with a topological ordering \leq_{top} of the refined pre-order is taken as input. The set $\mathcal{O}(B)$, the pre-orders and the topological order have to be computed before launching this procedure. This is possible due to Lemma 8.

procedure NiceOrderedPartition

Input: $(\mathcal{O}(B), \preceq, \leq_{top})$ the pre-order on the set of all one-blocks together with a topological order of the refined pre-order

Output: (L, R) - lists of one-blocks to be constructed s.t. (L, R) defines a nice ordered partition

Variables: M - an array used to store forcing information on the one-blocks.

$$M[O] = \begin{cases} \text{“L”} & - O \text{ is forced to be in } L; \\ \text{“R”} & - O \text{ is forced to be in } R; \\ \text{“L or R”} & - O \text{ can be either in } L \text{ or in } R. \end{cases}$$

MinHitFound - information if the minimal hit one-block has been found

MinHitSide - information on the side where the minimal hit one-block has been put, initialized to be L but R would be fine as well

procedure shake(O)

forall $X \in \mathcal{O}$ s.t. $(O \leq_{top} X)$ and $(X \text{ incomp. with } O \text{ for } \preceq)$ and $(M[X] = \text{“L or R”})$

$M[X] := \text{opposite}(M[O])$ **do**

shake(X)

end

$MinHitFound := \text{false}$

$MinHitSide := L$

$L \leftarrow \emptyset; R \leftarrow \emptyset$

forall $O \in \mathcal{O}$ **do** $M[O] := \text{“L or R”}$

forall $O \in \mathcal{O}$ in the topological order **do**

if $(M[O] = \text{“L or R”})$ **then**

if O is sparable **then**

move O on the top of $\text{opposite}(MinHitSide)$ and $M[O] := \text{opposite}(MinHitSide)$

shake(O)

else

move O on the top of $MinHitSide$ and $M[O] := MinHitSide$

shake(O)

endif

else

```

    move  $O$  on the top of  $M[O]$ 
  endif
  if (not MinHitFound and  $O$  is hit) then
    MinHitFound := true
    MinHitSide :=  $M[O]$ 
  endif
endforall
return ( $L, R$ )

```

Theorem 6. *The pair (L, R) produced by Algorithm **NiceOrderedPartition** defines a nice ordered partition. The running time of the algorithm is $O(n^2)$.*

Proof. (Hints, see [9].) The algorithm takes the one-blocks of \mathcal{O} sorted in a topological order, and processes them one by one. Our goal is to obtain a pair of lists (L, R) like in Theorem 5. Initially all the blocks are marked “ L or R ”, which means that they can be put either in L or in R . When we decide to put a block in one of the lists, other blocks may be forced to appear in the opposite or in the same list, because of incomparability relations. Indeed if two blocks are incomparable with respect to \preceq , they must be placed on different sides of B . This forcing is explored through the procedure “shake”, by marking the blocks “ L ” or “ R ”.

We can think of the algorithm as having three phases. The first phase lasts as long as we do not encounter any hit block. The blocks considered in the first phase can be put in any convenient list; they will stay clean. When we meet the first hit block we enter the second phase. We denote by *MinHitSide* the list containing this block. Suppose without loss of generality that this list is L . We are in the second phase as long as R has no hit block. When we treat a block O we should put it in R if it is sparable; by putting it in L it would be filled. On the contrary, if O is not sparable we put it in L , thus keeping the possibility to spare other blocks. Eventually, the third phase starts when both L and R contain hit blocks. In this case we put the current block in any convenient list, it will be filled anyway. □

5 Putting Everything Together

Given an interval graph $G = (V, E)$ and a vertex x outside of G , we run the following procedure with $(G, x, N_{G'}(x))$ as input. A minimal interval completion of G' is obtained as $H' = (V', E')$ with $V' = V \cup \{x\}$ and $E' = E \cup \{\{x, y\} \mid y \in \text{IntervalCompletion}(G, x, N_{G'}(x))\}$.

We construct the block B as in Theorem 3 and then call the algorithm **NiceOrderedPartition** to obtain the nice ordered partition (L, R) . In order to compute a minimal interval completion of G' , according to Lemma 7 it remains to compute, by recursive calls, minimal interval completions of $G'_d[O \cup \{x\}]$ for each $O \in \{O_L, O_R\}$.

Here we might encounter a difficulty that we have not yet discussed. During this recursive call on $G'_d[O \cup \{x\}]$ it is possible that \mathcal{S}_x is restricted to the minimal

separator S bordering O in G (even when $G'_d[O \cup \{x\}]$ is not an interval graph). If we choose S as the block B , the algorithm calls itself again on $G'_d[O \cup \{x\}]$ and it will not terminate. In this case we define B as the union of maximal cliques of the input graph containing $N(x) \setminus \{d\}$, the neighborhood of x except the dummy vertex. In this case also **NiceOrderedPartition** produces a nice ordered partition, as explained in [9].

procedure MinIntervalCompletion

input: G, x, N - the neighborhood of x in G' , d - dummy vertex

output: modified N - the neighborhood of x in an interval completion of G'

if G' is an interval graph then return \emptyset

Compute \mathcal{S}_x .

if $N(d)$ is not the only minimal separator in \mathcal{S}_x then

$B :=$ a maximal piece between two minimal separators in \mathcal{S}_x or a
minimal separator $S \in \mathcal{S}_x$ if such a piece between does not exist

$N := N \cup B$

else

$B :=$ the union of all maximal cliques containing $N \setminus \{d\}$

endif

Compute the one-blocks $\mathcal{O}(B)$

Check sparability of all one-blocks.

Compute the graph $(\mathcal{O}(B), \leq)$ and a topological order \leq_{top}

$(L, R) \leftarrow \mathbf{NiceOrderedPartition}((\mathcal{O}(B), \leq, \leq_{top}))$

forall O in L above O_L do $N := N \cup O$

forall O in R above O_R do $N := N \cup O$ // no iterations if O_R is undefined

foreach $O \in \{O_L, O_R\}$ do

if O is sparable then

let $G_d[O] := G[O]$ plus a dummy vertex d adjacent to $O \cap B$

$N := N \cup \mathbf{MinIntervalCompletion}(G_d[O], x, N \cap O \cup \{d\}, d) \setminus \{d\}$

else if O is not sparable then $N := N \cup O$

return N

Theorem 7. *Algorithm MinIntervalCompletion computes a minimal interval completion of G' in $O(n^3)$ time.*

Proof. The algorithm is clearly polynomial. We prove in the full paper [9] that the algorithm can be implemented to run in $O(n^4)$ time, based on a $O(n^3)$ implementation of the **NiceOrderedPartition** procedure. Recently we improved the latter running time to $O(n^2)$, which gives the desired time bound. \square

Using the incremental approach, Theorem 7 and the similar result for the case when the neighborhood of x is a clique (see [9]), we deduce:

Theorem 8. *There is an algorithm computing a minimal interval completion of an arbitrary graph in $O(n^4)$ time.*

6 Conclusion

In this paper we give the first polynomial time algorithm that computes a minimal interval completion of an arbitrary graph. With small modifications, our algorithm can be transformed to produce any of the minimal interval completions of the input graph. A natural question is whether the algorithm can be adapted into heuristics specialized in finding interval completions with few fill edges or small clique size. We are now able to adapt the algorithm in order to choose, at each iteration, an interval completion that (locally) adds a minimum number of edges or minimizes the cliquesize.

Acknowledgment

We would like to thank Fedor Fomin for useful discussions on the subject.

References

1. A. Berry, P. Heggernes, and Y. Villanger. A vertex incremental approach for dynamically maintaining chordal graphs. In *Algorithms and Computation - ISAAC 2003*, pages 47–57. Springer Verlag, 2003. LNCS 2906.
2. V. Bouchitté and I. Todinca. Treewidth and minimum fill-in: grouping the minimal separators. *SIAM J. on Computing*, 31(1):212 – 232, 2001.
3. M. R. Garey and D. S. Johnson. *Computers and Intractability*. W. H. Freeman and Co., 1978.
4. P. C. Gilmore and A. J. Hoffman. A characterization of comparability graphs and of interval graphs. *Canadian J. Math.*, 16:539–548, 1964.
5. M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, 1980.
6. P. W. Goldberg, M. C. Golumbic, H. Kaplan, and R. Shamir. Four strikes against physical mapping of DNA. *J. Comput. Bio.*, 2(1):139–152, 1995.
7. J. Gustedt. On the pathwidth of chordal graphs. *Discrete Applied Mathematics*, 45(3):233–248, 2003.
8. P. Heggernes. Minimal triangulations of graphs: A survey. To appear *Discrete Math.*
9. P. Heggernes, K. Suchan, I. Todinca, and Y. Villanger. Minimal interval completions. Technical Report RR2005-04, LIFO - University of Orléans, 2005. <http://www.univ-orleans.fr/SCIENCES/LIFO/prodsci/rapports/RR2005.htm.en>.
10. T. Kloks, D. Kratsch, and J. Spinrad. On treewidth and minimum fill-in of asteroidal triple-free graphs. *Theor. Comput. Sci.*, 175:309–335, 1997.
11. C.G. Lekkerkerker and J.C. Boland. Representation of a finite graph by a set of intervals on the real line. *Fund. Math.*, 51:45–64, 1962.
12. A. Parra and P. Scheffler. Characterizations and algorithmic applications of chordal graph embeddings. *Disc. Appl. Math.*, 79:171–188, 1997.
13. D. Rose, R.E. Tarjan, and G. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM J. Comput.*, 5:146–160, 1976.
14. M. Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM J. Alg. Disc. Meth.*, 2:77–79, 1981.

A 2-Approximation Algorithm for Sorting by Prefix Reversals

Johannes Fischer and Simon W. Ginzinger

LFE Bioinformatik und Praktische Informatik,
Ludwig-Maximilians-Universität München,
Amalienstr. 17, D-80333 München

{Johannes.Fischer, Simon.Ginzinger}@bio.ifi.lmu.de

Abstract. Sorting by Prefix Reversals, also known as Pancake Flipping, is the problem of transforming a given permutation into the identity permutation, where the only allowed operations are reversals of a prefix of the permutation. The problem complexity is still unknown, and no algorithm with an approximation ratio better than 3 is known. We present the first polynomial-time 2-approximation algorithm to solve this problem. Empirical tests suggest that the average performance is in fact better than 2.

1 Introduction

The problem of sorting a permutation by *prefix reversals*, also known as *pancake flipping*, was first considered in a computational context by Gates and Papadimitriou [7]. The problem may informally be described as follows: given a permutation of the first n integers, transform it into the sorted sequence $1, \dots, n$ by using as few prefix reversals as possible, an operation that flips the first x elements. In [7], and likewise in a subsequent article by Heydari and Sudborough [11], bounds were given that only depend on the size n of the permutation, disregarding special properties of the given permutation. This reflects the concept of the diameter $f(n)$ of the *pancake network* of size n , i.e. the maximal number of prefix reversals that is needed to sort an (arbitrary) permutation of $1, \dots, n$. In summary, we now know [7, 11] that $(15/14)n \leq f(n) \leq (5n + 5)/3$.

In this article, we tackle the problem of finding a minimal sequence of prefix reversals that sorts a *given* permutation π (MIN-SBPR). It should be clear that for n fixed, some permutations of length n are easier to sort than others. This intuition can be formalized by the concept of *breakpoints* and *breakpoint graphs*, first introduced by Bafna and Pevzner [1]. They considered a different version of the problem, the task of sorting a permutation by (arbitrary) reversals (MIN-SBR). The problem was shown to be NP-complete by Caprara [4], but Hannenhalli and Pevzner [9] gave a polynomial time algorithm for the slightly restricted case of *signed* permutations which is highly relevant in computational biology. Despite the NP-completeness of MIN-SBR, substantial progress has been made in finding approximation algorithms, starting with Kececioglu and Sankoff's algorithm [14] which has a performance guarantee of 2. The currently best known algorithm is a 1.375-approximation due to Berman et al. [3].

A related problem is the one of *sorting by transpositions* (MIN-SBT), where one seeks to find the minimum number of transpositions needed to sort a permutation. In contrast to MIN-SBR, it is still unknown whether MIN-SBT is in P or NP-hard. However, several 1.5-approximation algorithms have been devised (e.g. [2], the most recent being [10]). Similar to the case of reversals, the problem of *prefix transpositions* has been considered. Here, the currently best result is a 2-approximation by Dias and Meidanis [6].

However, in the case of prefix *reversals*, little progress has been made. Although Heydari [12] has proven the NP-completeness of a *modified* version of MIN-SBPR, it remains unknown whether or not the original problem is in P. Similarly, although it is easy to come up with a 3-approximation for MIN-SBPR¹, no approximation algorithms with a performance guarantee less than 3 have been found. We give the first 2-approximation algorithm for MIN-SBPR.

The remainder of this article is organized as follows. Sect. 2 gives a formal definition of the problem and introduces the notion of breakpoint graphs and related concepts. It also states the lower bound on which our algorithm is based. Sect. 3 develops the 2-approximation algorithm. Sect. 4 shows by empirical tests that the actual performance of our approximation is much better than 2. Finally, Sect. 5 concludes and gives an outlook to future work.

2 Preliminaries

For a permutation $\pi = (\pi_1, \dots, \pi_n)$ of $\{1, \dots, n\}$, a *prefix reversal* $\phi(r)$ is defined as the operation that *flips* the first r elements of π for $2 \leq r \leq n$, i.e. $\pi \circ \phi(r) = (\pi_r, \dots, \pi_1, \pi_{r+1}, \dots, \pi_n)$. The *prefix reversal distance* $d(\pi)$ is the minimal number of prefix reversals that is needed to transform π into the identity permutation $\iota := (1, \dots, n)$. Determining d for a given permutation is known as the problem of *Sorting by Prefix Reversals* (MIN-SBPR) and is the issue of this article.

We now extend π by setting $\pi_0 := 0$ and $\pi_{n+1} := n + 1$, yielding $\pi' = (0, \pi_1, \dots, \pi_n, n + 1)$. For convenience, we will also write π for the extended permutation π' . We say that there is a *breakpoint* between π_i and π_{i+1} if $|\pi_i - \pi_{i+1}| \neq 1$ for $1 \leq i \leq n$.² That is, there is a breakpoint between 2 elements that are adjacent in π , but not adjacent in the identity permutation $\iota' := (1, \dots, n + 1)$. As an example, the breakpoints in the following permutation are marked with a horizontal bar: $(8|2, 1|9|5, 6|3, 4|7|)$. We define $b(\pi)$ to be the number of breakpoints in π .

¹ The obvious 3-approximation flips the strip with the highest unsorted element to the beginning, brings it in the correct direction and then flips it to its proper place at the end where it remains untouched. See Sec. 2 for the definition of strips.

² Because of the inherent asymmetry of prefix reversals, we never say that there is a breakpoint between π_0 and π_1 . However, in the *breakpoint graph* (to be defined after Lemma 1) we do have a red edge between π_0 and π_1 if $|\pi_0 - \pi_1| \neq 1$.

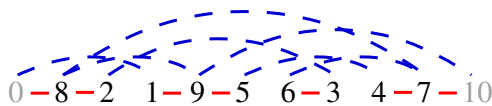


Fig. 1. The breakpoint graph of the permutation $(8|2, 1|9|5, 6|3, 4|7, 10)$

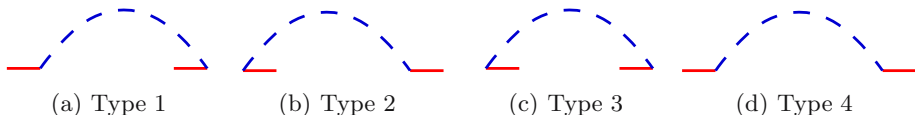


Fig. 2. Different types of blue edges

An immediate consequence is

Lemma 1. For a permutation $\pi = (\pi_1, \dots, \pi_n)$ of $\{1, \dots, n\}$ with $b(\pi)$ breakpoints, we have

$$d(\pi) \geq b(\pi) .$$

Proof. Let $\phi(r_1), \dots, \phi(r_{d(\pi)})$ be an optimal sequence of prefix reversals that sorts π , i.e. $\pi \circ \phi(r_1) \circ \dots \circ \phi(r_{d(\pi)}) = \iota$. Note that a reversal $\phi(r)$ can eliminate at most one breakpoint, namely the one between π_r and π_{r+1} (if any). Since ι is the only permutation having 0 breakpoints, the claim follows. \square

We note that this bound is not sharp for all cases. For example, the permutation $(3, 4|1, 2|)$ has two breakpoints, but needs at least *three* prefix reversals to be transformed into the identity permutation. (The prefix reversals are $\phi(2)$, $\phi(4)$ and $\phi(2)$.)

The *breakpoint graph* $G_\pi = (V, E)$ of a permutation π is defined as follows: the set of vertices in G_π is $V := \{\pi_0, \dots, \pi_{n+1}\}$, and the set of (directed) edges E is the union of so-called *red edges* R and *blue edges* B to be defined next. An edge e is in R if and only if $e = (\pi_i, \pi_{i+1})$ and there is a breakpoint between π_i and π_{i+1} , or $e = (\pi_0, \pi_1)$ and $|\pi_0 - \pi_1| \neq 1$. Further, an edge e is in B if and only if $e = (\pi_i, \pi_j)$ for some $0 \leq i < j \leq n + 1$, $\pi_j = \pi_i \pm 1$ and $\pi_i \in e'$ for some red edge e' . Note that the number of blue edges equals the number of red edges. See Fig. 1 for an example of a breakpoint graph, where the blue edges are drawn as dashed lines to distinguish them from the red edges. We stick to the convention that the breakpoint graph is drawn “from left to right”, and that red edges are drawn as straight lines, whereas blue edges are drawn as arcs. Whenever we talk of the *first* or *leftmost* blue edge we mean the blue edge $\operatorname{argmin}_{(\pi_i, \pi_j) \in B} i$. Likewise, the *last* or *rightmost* blue edge is defined as $\operatorname{argmax}_{(\pi_i, \pi_j) \in B} j$. For example, in Fig. 1 the first blue edge connects 0 and 1 and the last blue edge connects 9 and 10.

We say that π_i, \dots, π_j form a *strip* for $1 \leq i \leq j \leq n$ if $\pi_k = \pi_{k+1} \pm 1$ for all $i \leq k < j$, $\pi_i \neq \pi_{i-1} \pm 1$ and $\pi_j \neq \pi_{j+1} \pm 1$. A strip is called a *singleton* if it consists of only one element π_i , with the exception that $\pi_1 = 1$ and $\pi_n = n$

are never considered as singletons. A strip of length ≥ 2 is called *ascending* if $\pi_k = \pi_{k+1} - 1$ for all $i \leq k < j$ and *descending* otherwise.

Now consider a blue edge between 2 elements π_i and $\pi_j := \pi_i \pm 1$. Because of the definition of blue edges there is at least one adjacent red edge on each side of the blue edge; we can thus classify such a blue arc into one of the four different types shown in Fig. 2, depending on the directions of the adjacent red edges. Note that if π_i 's or π_j 's strip is a singleton, the blue arc may be classified into at least 2 different types. As an example, consider the blue arc (6, 7) in Fig. 1 which is of type 2 and 3.

Note that we will *not* use the breakpoint graph for theoretical considerations such as relating the maximal number of alternating cycles in it to the reversal distance as in [9]; it will rather be used for expository purposes. In the rest of this paper we will only consider permutations where there is a breakpoint between π_n and π_{n+1} . This is simply because if there are x ordered elements at the end of π , say $\pi = (\pi_1, \dots, \pi_{n-x}, n-x+1, \dots, n-1, n)$, we can reduce the problem of sorting π to sorting the permutation $\pi' := (\pi_1, \dots, \pi_{n-x})$. We will further restrict ourselves to prefix reversals that *act* on a breakpoint, which means that $\phi(r)$ is applied to π only if there is a breakpoint between π_r and π_{r+1} . We will see later that this restriction is sufficient for a 2-approximation.³

3 The 2-Approximation Algorithm

We are now ready to present the details of our 2-approximation. The next lemma gives a nice property of the breakpoint graph that allows us to eliminate a breakpoint by using at most two prefix reversals.

Lemma 2. *Let π be a permutation of $\{1, \dots, n\}$ and G_π be its associated breakpoint graph. Then there exists a sequence of at most two prefix reversals $\phi(r)$ and $\phi(s)$ that eliminates a breakpoint if at least one of the three conditions holds:*

1. G_π contains a blue arc (π_i, π_j) of type 1 with $i = 1$.
2. G_π contains a blue arc (π_i, π_j) of type 2, where $i \neq 0$.
3. G_π contains a blue arc (π_i, π_j) of type 3.

Proof. Because (π_i, π_j) is blue we must have $\pi_j = \pi_i \pm 1$. We show that in all cases we can create an adjacency between π_i and π_j without introducing any new breakpoints, thereby eliminating a breakpoint. The reader is encouraged to follow the examples shown in Fig. 3.

In case 1, the single prefix reversal $\phi(j - 1)$ creates the desired adjacency between π_1 and π_j without introducing a new breakpoint. See Fig. 3 (a) for an example.

In case 2, $\phi(j)$ and $\phi(j - i)$ suffice: we have $(\pi_1, \dots, \pi_n) \circ \phi(j) = (\pi_j, \dots, \pi_i, \dots, \pi_1, \pi_{j+1}, \dots, \pi_n) =: \pi'$; so $G_{\pi'}$ contains a blue arc (π'_1, π'_{j-i+1}) of type 1, and

³ In analogy to Hannenhalli and Pevzner's Theorem 3.1 in [8] we can in fact show that there is an optimal sorting of prefix reversals that does not "cut" a strip of length ≥ 3 . However, for strips of length 2 this is not the case.

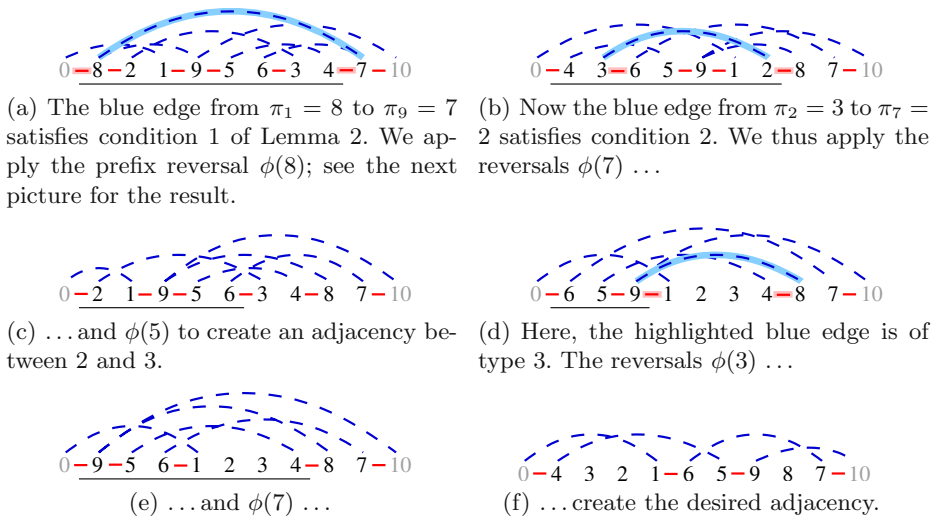


Fig. 3. An example. We want to sort $\pi = (8|2, 1|9|5, 6|3, 4|7|)$ (the six breakpoints are marked with a bar). The adjacencies that will be created next are highlighted. The example will be continued in figure 7, when the necessary tools have been introduced.

we are thus in the situation of case 1. The condition $i \neq 0$ is necessary because with $i = 0$ we have $\pi_0 = 0$, so $\pi_j = 1$ and we cannot create an adjacency by our definition of breakpoints. Note also that an arc of type 2 cannot be the last blue edge in G_π ; this implies in particular $j \neq n + 1$, so $\phi(j)$ is a valid reversal. See Fig. 3 (b)–(c) for an example.

In case 3, the adjacency is created by the prefix reversals $\phi(i)$ and $\phi(j - 1)$: again, with $(\pi_1, \dots, \pi_n) \circ \phi(i) = (\pi_i, \dots, \pi_1, \pi_{i+1}, \pi_j, \dots, \pi_n) =: \pi'$ we see that $G_{\pi'}$ contains a blue arc (π'_1, π'_j) of type 1. See Fig. 3 (d)–(f) for an example. \square

For the rest of this section, we will say that a blue edge of type 1,2 or 3 is a *good edge* if it satisfies one of the respective conditions given in Lemma 2. The next two lemmas are the key to our 2-approximation. They characterize those permutations that do *not* contain a good edge. In fact, we will show that these permutations all resemble a certain *prototype* permutation (given in Eq. (*) on p. 421) which can then be solved by the generic sorting sequence given in Lemma 6.

Lemma 3. *Let π be a permutation of $\{1, \dots, n\}$ that does not contain a good blue edge and let G_π be its associated breakpoint graph. Then π does not contain any singletons.*

Proof. Assume π contains a singleton, say at position $1 \leq i \leq n$. Then there are two blue edges beginning or ending at π_i . Further, there is a red edge ending at π_i and a red edge beginning at π_i . If one of the two blue edges had π_i as its left endpoint, then this edge would be of type 2 or 3 (or both), so at least

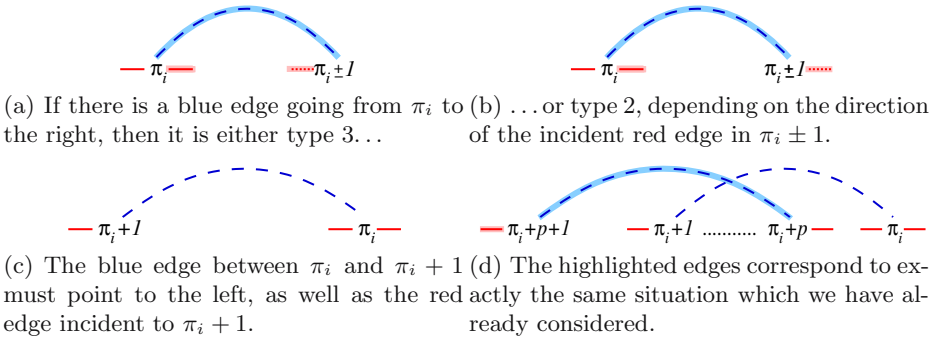


Fig. 4. Illustrations to the proof of Lemma 3

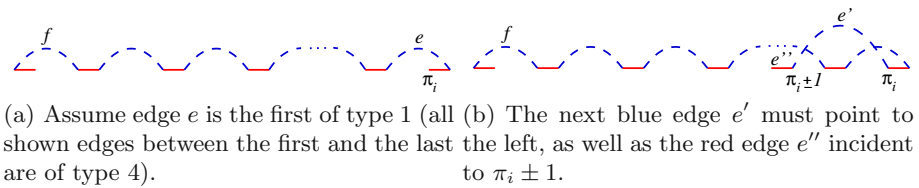


Fig. 5. Illustrations to the proof of Lemma 4

one of the conditions 2 or 3 in Lemma 2 would hold, a contradiction (see Fig. 4 (a)–(b)). So both blue edges have π_i as their right endpoint, in particular the edge starting at π_i+1 . There cannot be a red edge on the right side of π_i+1 , because otherwise the blue edge (π_i+1, π_i) would be of type 3 and thus be good. So the only red edge incident to π_i+1 ends there (Fig. 4 (c)). This means that there is an *ascending* strip from π_i+1 to π_i+p for some $p \geq 2$. Now we either have $\pi_i+p = n$, in which case there must be a blue edge from $\pi_i+p = n$ to $\pi_{n+1} = n+1$, which would then be of type 3, a contradiction. If, on the other hand, $\pi_i+p \neq n$, by the same reasoning as above we know that the blue edge incident to π_i+p must point to the left, and again the red edge incident to π_i+p+1 must also point to the left (Fig. 4 (d)). We are thus in the same situation as before and must eventually reach n which gives us a blue edge of type 3, a contradiction. This proves that there are no singletons in π . \square

Lemma 4. *Let $\pi \neq \iota$ be a permutation of $\{1, \dots, n\}$ that does not contain a good blue edge and let G_π be its associated breakpoint graph. Then all of G_π 's blue edges have a unique type, the first is of type 2, the last is of type 1, and all other blue edges are of type 4.*

Proof. The fact that all blue edges have a unique type follows immediately from Lemma 3. Since condition 3 of Lemma 2 is not satisfied, the first blue edge cannot be of type 3 and must thus be of type 2. (The first blue edge can never be of type 1 or 4.) We now prove that all other blue edges apart from the last are of type 4. In fact, all we need to show is that they are *not* of type 1, for if

there were an arc of type 2 or 3, one of the conditions 2 or 3 in Lemma 2 would hold. Note further that because π contains no singletons, there are either 0 or 2 edges incident to each vertex of G_π . So G_π forms a unique cycle. Following this cycle from the end of the leftmost blue edge f , look at the first blue edge e of type 1, see Fig. 5 (a). (We will actually show that this edge must be the last.) Define π_i as the element to the left of the right endpoint of e . By the same reasoning as in Lemma 3, the blue edge e' incident to π_i cannot point to the right, and likewise the next red edge e'' must point to the left (Fig. 5 (b)), because otherwise e' would be of type 2. Now either $e' = f$ (in which case we are done), or the argument can be continued inductively until we eventually reach edge f . The proof is finished by noting that there must be at least one blue edge of type 1, for otherwise G_π would not form a cycle. \square

For the proof of the following lemma we need another definition for blue edges [9]: two blue edges (π_i, π_j) and (π_k, π_l) are said to be *interleaving* if the intervals $[i, j]$ and $[k, l]$ overlap but neither of them contains the other.

Lemma 5. *Let $\pi \neq \iota$ be a permutation of $\{1, \dots, n\}$ that does not contain a good blue edge. Then π is of the form*

$$\pi = \underbrace{(p_1, \dots, 1)}_{l_1}, \underbrace{(p_2, \dots, p_1 + 1)}_{l_2}, \dots, \underbrace{(n, \dots, p_{b(\pi)-1} + 1)}_{l_{b(\pi)}}, \quad (*)$$

i.e. π consists of $b(\pi) \geq 2$ descending strips of length $l_i \geq 2$ for all $1 \leq i \leq b(\pi)$.

Proof. By Lemma 4, G_π consists of one blue edge of type 2 at the beginning, one blue edge of type 1 at the end, and all blue edges in between are of type 4. Because G_π forms a unique cycle and blue edges of type 4 must interleave with at least two other blue edges, the only possible arrangement of these edges looks as shown in Fig. 6, with (π_i, π_j) being its first blue edge.

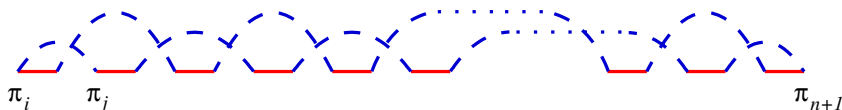


Fig. 6. The only possible breakpoint graph when we cannot apply two reversals that eliminate a breakpoint

First note that $i = 0$, for otherwise the blue edge (π_i, π_j) would be good. Following the first blue arc from $\pi_i = 0$, we see that $\pi_j = 1$. Therefore the first strip must be $p_1, \dots, 1$ with $l_1 = p_1 \geq 2$, for if $p_1 = 1$, the strip would be a singleton. Continuing this line of arguments we get that π must of the form $(*)$.

To see that $b(\pi) \geq 2$, assume that there is only one breakpoint in π . Then there would be no edge of type 4 in G_π , so the blue edge of type 1 would start at π_1 and would thus be good. \square

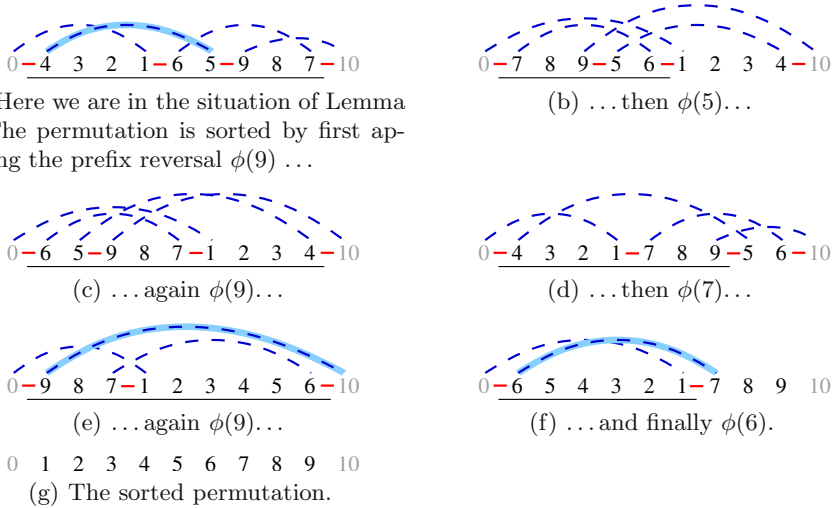


Fig. 7. Continuing the running example: This figure shows the effect of the generic sorting sequence

The previous lemma has characterized permutations that do not contain a good blue edge. We now show how to cope with these “hard” instances.

Lemma 6. *Assume π is of the form $(*)$, and let $l_1, \dots, l_{b(\pi)}$ be defined as in Lemma 5. Then the sequence of $2b(\pi)$ prefix reversals*

$$\phi(n) \circ \phi(n - l_1) \circ \phi(n) \circ \phi(n - l_2) \circ \dots \circ \phi(n) \circ \phi(n - l_{b(\pi)})$$

applied to π yields the identity permutation.

Proof. Applying the first two reversals to π yields

$$\pi^* := (p_2, \dots, p_1 + 1, p_3, \dots, p_2 + 1, \dots, n, \dots, p_{b(\pi)-1} + 1, 1, \dots, p_1) .$$

We now prove by induction on the number of breakpoints in π^* (equal to $b(\pi)$) that applying the next $2b(\pi) - 2$ prefix reversals yields the identity. The base is when $b(\pi^*) = 2$. So $\pi^* = (n, \dots, p_1 + 1, 1, \dots, p_1)$, and $\pi^* \circ \phi(n) \circ \phi(n - l_2)$ is clearly equal to ι .

For the induction step, let $\pi^* = (p_2, \dots, p_1 + 1, p_3, \dots, p_2 + 1, \dots, n, \dots, p_{b(\pi)-1} + 1, 1, \dots, p_1)$. Now $\pi^* \circ \phi(n) \circ \phi(n - l_2) = (p_3, \dots, p_2 + 1, \dots, n, \dots, p_{b(\pi)-1} + 1, 1, \dots, p_1, p_1 + 1, \dots, p_2)$, which is of the same form as π^* but has one breakpoint less. The claim follows. \square

See Fig. 7 for an example of the generic sorting sequence. We note that in the above lemma, the first two prefix reversals ($\phi(n)$ and $\phi(n - l_1)$) do *not* eliminate a breakpoint, whereas the last two prefix reversals ($\phi(n)$ and $\phi(n - l_{b(\pi)})$) both create an adjacency, and all other pairs of reversals create exactly one adjacency.

That is, the last two prefix reversals compensate for the first two that could not create any adjacency.

We now come to our main result:

Corollary 1. *MIN-SBPR is 2-approximable.*

Proof. While the breakpoint graph of π contains a good blue edge, choose on of these edges and apply at most 2 prefix reversals to eliminate a breakpoint. If this is impossible, by Lemma 5 π must be of the form (*), which can be transformed into the identity permutation ι by the generic sequence of prefix reversals given in Lemma 6. The number of prefix reversals used is at most $2b(\pi)$. The claim follows with the lower bound given in Lemma 1. \square

4 Empirical Results

We implemented the algorithm that drops out from the proof of Cor. 1. The obvious strategy used was that good arcs of type 1 were preferred over good arcs of type 2 or 3 because the former just need *one* prefix reversal to create an adjacency instead of *two*. This algorithm was compared to a branch-and-bound method to compute the optimal number of prefix reversals to sort a given permutation. Due to the enormous size of the group of permutations (and the even more numerous number of possible sorting sequences to be inspected by the

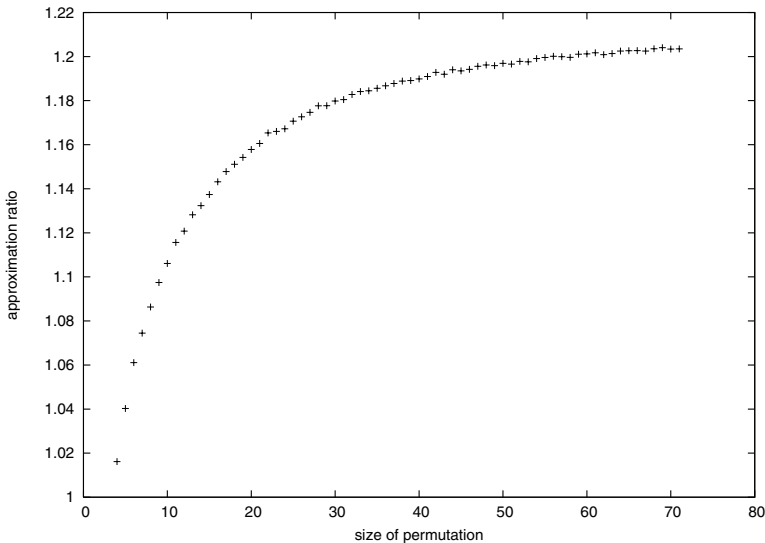


Fig. 8. The actual approximation ratios of our 2-approximation algorithm. The size of the permutations is plotted against the number of operations performed by our algorithm, divided by the minimum number of prefix reversals needed to sort these permutations. All numbers are averaged over 10,000 random permutations of the shown size.

branch-and-bound algorithm), we chose to select *at random* 10,000 permutations of length up to 71 and computed the actual approximation ratios of our 2-approximation. The results can be seen in Fig. 8.

It is interesting to see that the actual approximation ratio is much better than 2. This suggests that with a deeper analysis of the algorithm the theoretical approximation ratio could even be lowered. For example, we were able to prove that certain blue arcs of type 4 (similar to the ones in Fig. 6) contribute to $d(\pi)$ by an extra prefix reversal. Nevertheless this is *not* sufficient to lower the theoretical approximation ratio of the algorithm: To do so, one would have to make sure that, among other things, such situations are not “created” unless they are inevitable.

Another point to note on Fig. 8 is that the actual approximation ratio seems to level off at ≈ 1.2 . One possible explanation for this could be that the number of “hard” permutations for our method converges against a constant fraction of the size of the group. However, because we could only sample a *constant* number of permutations for every n (namely 10,000), it could also be that the really hard instances were not covered by our randomly chosen permutations and the true approximation ratio is worse than the graph shows.

5 Conclusions and Outlook

We have seen an algorithm to solve MIN-SBPR with an approximation ratio of 2. We note that Cohen and Blum [5] give a similar 2-approximation for the *signed* version of MIN-SBPR, parts of which could also have been used for our problem. While our result is rather of theoretical interest, empirical tests have shown that on average, our algorithm is within ≈ 1.2 of the optimal for permutations of length up to 71. This suggests that there is a *hidden* parameter in the prefix-reversal-distance $d(\pi)$, in a similar way as *hurdles* and *fortresses* account for the reversal-distance in MIN-SBR. Future work will be directed towards raising the lower bound by identifying the parameters influencing the prefix-reversal distance. From a theoretical standpoint, another interesting topic of research is to prove the theoretical computational complexity of both the signed and the unsigned version of the problem.

Acknowledgments

We wish to thank Volker Heun for fruitful discussions and helpful suggestions on this subject. Further thanks go to an anonymous reviewer who pointed out the connection to the signed version of MIN-SBPR [5].

References

1. V. Bafna, P. A. Pevzner: Genome Rearrangements and Sorting by Reversals. *SIAM J. on Computing*, 25(2): 272–289, 1996.
2. V. Bafna, P. A. Pevzner: Sorting by Transpositions *SIAM J. on Discrete Mathematics* 11(2), 224–240, 1998.

3. P. Berman, S. Hannenhalli, M. Karpinski: 1.375-Approximation Algorithm for Sorting by Reversals. *Proc. ESA'02, Lecture Notes in Computer Science* 2461: 200–210, 2002.
4. A. Caprara: Sorting Permutations by Reversals and Eulerian Cycle Decompositions. *SIAM J. on Discrete Mathematics* 12(1): 91–110, 1999.
5. D. S. Cohen, M. Blum: On the Problem of Sorting Burnt Pancakes. *Discrete Applied Mathematics* 61: 105–120, 1995.
6. Z. Dias, J. Meidanis: Sorting by Prefix Transpositions. *Proc. SPIRE'02, Lecture Notes in Computer Science* 2476: 65–76, 2002.
7. W. H. Gates, C. H. Papadimitriou: Bounds for Sorting by Prefix Reversals. *Discrete Mathematics* 27: 47–57, 1979.
8. S. Hannenhalli, P. Pevzner: TO CUT ... OR NOT TO CUT (Applications of Comparative Physical Maps in Molecular Evolution). Proceedings of the 7th ACM Symposium on Discrete Algorithms (SODA'96), 304–313, 1996.
9. S. Hannenhalli, P. A. Pevzner. Transforming Cabbage into Turnip: Polynomial Algorithm for Sorting Signed Permutations by Reversals. *J. of the ACM* 46(1): 1–27, 1999.
10. T. Hartman, R. Shamir: A Simpler 1.5-Approximation Algorithms for Sorting by Transpositions, *Proc. CPM'03, Lecture Notes in Computer Science* 2676, 156–169, 2003.
11. M. H. Heydari, I. H. Sudborough: On the Diameter of the Pancake Network. *J. of Algorithms* 25: 67–94, 1997.
12. M. H. Heydari. *The Pancake Problem*. PhD-thesis, University of Wisconsin at Whitewater, 1993.
13. J. Kececioglu, D. Sankoff. Efficient Bounds for Oriented Chromosome Inversion Distance. *Proc. CPM'94, Lecture Notes in Computer Science* 807: 307–325, 1994.
14. J. Kececioglu, D. Sankoff. Exact and Approximation Algorithms for Sorting by Reversals, with Application to Genome Rearrangements. *Algorithmica* 13: 180–210, 1995.

Approximating the 2-Interval Pattern Problem

Maxime Crochemore^{1,*}, Danny Hermelin², Gad M. Landau^{3,**},
and Stéphane Vialette⁴

¹ Institut Gaspard-Monge, Université de Marne-la-Vallée, France,
and Department of Computer Science, King's Collage, London, UK
`maxime.crochemore@univ-mlv.fr`

² Department of Computer Science, University of Haifa, Israel
`danny@cri.haifa.ac.il`

³ Department of Computer Science, University of Haifa, Israel,
and Department of Computer and Information Science,
Polytechnic University, NY, USA
`landau@cs.haifa.ac.il`

⁴ Laboratoire de Recherche en Informatique (LRI), Université Paris-Sud, France
`viallette@lri.fr`

Abstract. We address the problem of approximating the 2-INTERVAL PATTERN problem over its various models and restrictions. This problem, which is motivated by RNA secondary structure prediction, asks to find a maximum cardinality subset of a 2-interval set with respect to some prespecified model. For each such model, we give varying approximation quality depending on the different possible restrictions imposed on the input 2-interval set.

1 Introduction

In the context of RNA secondary structure prediction, Vialette [11] proposed a geometric representation of a helix in an RNA single stranded molecule by means of a natural generalization of an interval, namely a *2-interval*. A 2-interval is the union of two disjoint intervals defined over a single line. In [11], intervals and 2-intervals represent respectively sequences of contiguous bases and possible pairings between such sequences in the RNA secondary structure. The goal is to find a maximum disjoint subset of the given set of 2-intervals, restricted to prespecified geometrical constrains, so as to serve as a valid approximation of the actual secondary structure of the given RNA.

Throughout the paper, a 2-interval is denoted by $D = (I, J)$ where I and J are two (closed) intervals defined over a single line such that $I < J$, *i.e.*, I is completely to the left of J . Two 2-intervals $D_1 = (I_1, J_1)$ and $D_2 = (I_2, J_2)$ are *disjoint*, if both 2-intervals share no common point, *i.e.*, $(I_1 \cup J_1) \cap (I_2 \cup J_2) = \emptyset$. For such disjoint pairs of 2-intervals, three natural binary relations are of special interest.

* Partially supported by CNRS, France, and the French Ministry of Research through ACI NIM.

** Partially supported by the Israel Science Foundation grant 282/01.

Definition 1 (Relations between 2-intervals). Let $D_1 = (I_1, J_1)$ and $D_2 = (I_2, J_2)$ be two disjoint 2-intervals. Then

- $D_1 < D_2$ (D_1 precedes D_2), if $I_1 < J_1 < I_2 < J_2$.
- $D_1 \sqsubset D_2$ (D_1 is nested in D_2), if $I_2 < I_1 < J_1 < J_2$.
- $D_1 \bowtie D_2$ (D_1 crosses D_2), if $I_1 < I_2 < J_1 < J_2$.

A pair of 2-intervals D_1 and D_2 is R -comparable for some $R \in \{<, \sqsubset, \bowtie\}$, if either $(D_1, D_2) \in R$ or $(D_2, D_1) \in R$. A set of 2-intervals \mathcal{D} is \mathcal{R} -comparable for some $\mathcal{R} \subseteq \{<, \sqsubset, \bowtie\}$, $\mathcal{R} \neq \emptyset$, if any pair of distinct 2-intervals in \mathcal{D} is R -comparable for some $R \in \mathcal{R}$. The non-empty subset \mathcal{R} is called a *model*. Note that any two disjoint 2-intervals are R -comparable for some $R \in \{<, \sqsubset, \bowtie\}$. Equivalently, any pairwise disjoint subset of \mathcal{D} is $\{<, \sqsubset, \bowtie\}$ -comparable. In [3,11], the 2-INTERVAL PATTERN problem is defined as follows:

Definition 2 (The 2-INTERVAL PATTERN problem). Let \mathcal{D} be a set of 2-intervals and let $\mathcal{R} \subseteq \{<, \sqsubset, \bowtie\}$, $\mathcal{R} \neq \emptyset$, be a given model. The 2-INTERVAL PATTERN problem asks to find a maximum cardinality \mathcal{R} -comparable subset of \mathcal{D} .

By the above definition, any solution for the 2-INTERVAL PATTERN problem over a model \mathcal{R} corresponds to a secondary structure constrained by \mathcal{R} . Let \mathcal{D} be a set of 2-intervals and let $\mathcal{S}(\mathcal{D}) = \{I, J : D = (I, J) \in \mathcal{D}\}$ be the set of intervals involved in \mathcal{D} . Several biologically motivated restrictions on \mathcal{D} and $\mathcal{S}(\mathcal{D})$ are of interest.

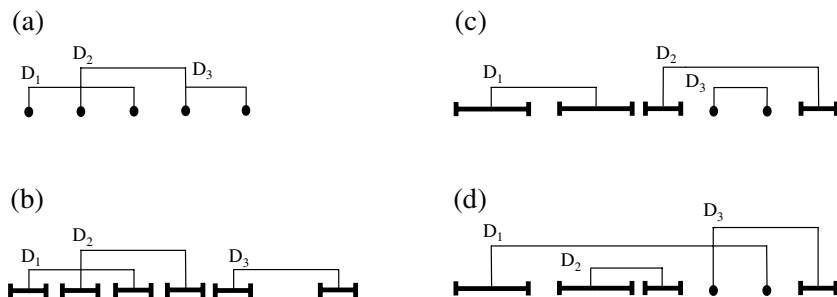


Fig. 1. Restrictions for 2-interval sets. Intervals are represented by dark lines or circles and 2-intervals are represented by a thin line connecting two intervals. (a) A point 2-interval set where $D_1 \bowtie D_2$ and $D_1 < D_3$. D_2 and D_3 are not disjoint and thus are not comparable by any relation. (b) A unitary 2-interval set where $D_1 \bowtie D_2$, $D_1 < D_3$, and $D_2 < D_3$. (c) A balanced 2-interval set where $D_3 \sqsubset D_2$. The entire set is $\{<, \sqsubset\}$ -comparable. (d) An unlimited $\{<, \sqsubset, \bowtie\}$ -comparable 2-interval set.

Definition 3. Let \mathcal{D} be a set of 2-intervals and let $\mathcal{S}(\mathcal{D})$ be the set of intervals involved in \mathcal{D} .

- \mathcal{D} is a point 2-interval set if all intervals in $\mathcal{S}(\mathcal{D})$ are pairwise disjoint (note that in this case, all intervals in $\mathcal{S}(\mathcal{D})$ may be considered as points).

- \mathcal{D} is a unitary 2-interval set if $\mathcal{S}(\mathcal{D})$ consists of intervals of unit length.
- \mathcal{D} is a balanced 2-interval set if any 2-interval in \mathcal{D} is a pair of two intervals of equal length.
- \mathcal{D} is an unlimited 2-interval set if none of the above restrictions are imposed.

The left part of Table 1 depicts the current state of the art for the 2-INTERVAL PATTERN problem in terms of exact algorithms. In [11], the 2-INTERVAL PATTERN problem over the $\{\sqsubset, \emptyset\}$ and $\{<, \sqsubset, \emptyset\}$ models is proved to be **NP**-hard even for unitary 2-interval sets. The proof for the $\{<, \sqsubset, \emptyset\}$ model is obtained as a direct consequence of the **APX**-hardness result for the MAXIMUM INDEPENDENT SET problem for t -interval graphs given in [2]. The results in [2] also provide approximation algorithms for this model. In [3], an **NP**-hardness result for the $\{<, \emptyset\}$ model restricted to unitary 2-interval sets is given. The time complexity for this same model when the input is restricted to point 2-interval sets is still unknown [11,3]. These results imply that in practical terms, secondary structures containing pseudoknots are hard to predict in our suggested mathematical model. This is consistent with previously known **NP**-hardness results for RNA secondary structures prediction in other models considering arbitrary pseudoknots [1,8,9].

Table 1. The 2-INTERVAL PATTERN problem over it’s various models and restrictions. Left part: Classical complexity results for the 2-INTERVAL PATTERN problem, where $n = |\mathcal{D}|$. Right part: The approximation factors we obtain in this paper.

Classical complexity				Approximation factors					
MODEL	UNL.	BAL.	UNI.	PNT.	MODEL	UNL.	BAL.	UNI.	PNT.
$\{<, \sqsubset, \emptyset\}$	NP-C [11,2]		$\mathcal{O}(n\sqrt{n})$ [11]		$\{<, \sqsubset, \emptyset\}$ (Section 2)	4^a	4^b	3^c	–
$\{\sqsubset, \emptyset\}$	NP-C [11]		$\mathcal{O}(n^2\sqrt{n})$ [3]		$\{\sqsubset, \emptyset\}$ (Section 3)	4^a	4^d	3^c	–
$\{<, \emptyset\}$	NP-C [3]		?		$\{<, \emptyset\}$ (Section 4)	6^b	5^b	3^c	2^c
$\{<, \sqsubset\}$	$\mathcal{O}(n^2)$ [11]								
$\{\emptyset\}$	$\mathcal{O}(n^2 \log n)$ [11]								
$\{\sqsubset\}$	$\mathcal{O}(n \log n)$ [3]								
$\{<\}$	$\mathcal{O}(n \log n)$ [11]								

^a Polynomial-time [2].
^b $\mathcal{O}(n^2)$ time algorithm.
^c $\mathcal{O}(n \lg n)$ time algorithm [2].
^d $\mathcal{O}(n^3)$ time algorithm.
^e $\mathcal{O}(n^2 \lg n)$ time algorithm.

In this paper we focus on the three **NP**-hard models of the 2-INTERVAL PATTERN problem. More specifically, we design constant factor approximation algorithms for the $\{<, \sqsubset, \emptyset\}$, $\{\sqsubset, \emptyset\}$, and $\{<, \emptyset\}$ models. The approximation factors obtained by all our algorithms vary depending on the restriction imposed on the input set of 2-intervals (see Table 1). Furthermore we suggest a new restriction, namely balanced 2-interval sets. By definition, unitary 2-interval sets are also balanced but the converse is not necessarily true. Consequently, the above mentioned **NP**-hardness results also hold for the balanced case, and moreover, balanced 2-interval sets introduce a new combinatorial object which requires

particular consideration. Furthermore, the balanced restriction is very natural in the biological setting of the problem.

This paper is organized as follows. In Section 2, we consider the 2-INTERVAL PATTERN problem over the general model, *i.e.*, the $\{<, \sqsubset, \boxminus\}$ model. We describe in Section 3 an approximation algorithm for the problem over the $\{\sqsubset, \boxminus\}$ model. Finally, in Section 4, the $\{<, \boxminus\}$ model is considered, and different approximation algorithms are introduced for all possible restrictions imposed on the input.

2 Approximation Algorithms for the $\{<, \sqsubset, \boxminus\}$ Model

We begin by considering the 2-INTERVAL PATTERN problem over the general model, *i.e.*, the $\{<, \sqsubset, \boxminus\}$ model. Recall that in this case, given an input set of 2-intervals \mathcal{D} , the problem asks to find a maximum $\{<, \sqsubset, \boxminus\}$ -comparable subset of \mathcal{D} , which is equivalent to finding a maximum pairwise disjoint subset of \mathcal{D} .

For point 2-intervals sets, 2-INTERVAL PATTERN can be solved in polynomial time by maximum matching [11]. For unitary 2-interval sets, the problem is already **APX**-hard [2], and therefore is **APX**-hard also for balanced and unlimited 2-interval sets. Furthermore, the results in [2] also yield approximation algorithms for our case which directly imply the following.

Proposition 1 ([2]). *The 2-INTERVAL PATTERN problem over the $\{<, \sqsubset, \boxminus\}$ model can be approximated within a factor of 4 when restricted to unlimited 2-interval sets, and a factor of 3 when restricted to unitary interval sets.*

The algorithm given in [2] that solves the case of unitary 2-interval sets can be executed in $\mathcal{O}(n \lg n)$ time, where n is the size of the input set of 2-intervals. However, the algorithm for unlimited 2-interval sets uses linear programming techniques, which in practice are very often too time costly. Clearly, balanced 2-interval sets lie between the two cases and are arguably the most biologically important case. In the rest of this section we describe a quadratic time 4-approximation algorithm for balanced 2-intervals sets.

Given any balanced 2-interval set \mathcal{D} , let the *smallest* 2-interval in \mathcal{D} be the 2-interval with the shortest left (or right, as they are both of equal length) interval among all left intervals involved in \mathcal{D} . The algorithm we suggest is a simple greedy strategy that repeatedly picks the smallest 2-interval in the input, adds it to the solution, and omits all other 2-intervals in the input which intersect it. A schematic description of this algorithm, which we call Bal- $\{<, \sqsubset, \boxminus\}$ -Approx, is given in Figure 2.

Lemma 1. *Algorithm Bal- $\{<, \sqsubset, \boxminus\}$ -Approx achieves an approximation factor guarantee of 4 for the 2-INTERVAL PATTERN problem over the general model, restricted to balanced 2-interval sets.*

Proof. Let \mathcal{D} be the set of remaining 2-intervals at any arbitrary iteration of Bal- $\{<, \sqsubset, \boxminus\}$ -Approx, and let $D_0 \in \mathcal{D}$ be the smallest 2-interval at this iteration. Since D_0 is the smallest 2-interval in \mathcal{D} , no interval involved in \mathcal{D} can be properly

```

Algorithm Bal- $\{\prec, \sqsubset, \emptyset\}$ -Approx( $\mathcal{D}$ )


---


Data : A set of balanced 2-intervals  $\mathcal{D}$ .
Result : A  $\{\prec, \sqsubset, \emptyset\}$ -comparable subset of  $\mathcal{D}$ .
begin
  while  $\mathcal{D} \neq \emptyset$  do
    1. Let  $D_0$  be the smallest 2-interval in  $\mathcal{D}$ .
    2. Add  $D_0$  to the solution.
    3. Omit  $D_0$  and all 2-intervals intersecting  $D_0$  from  $\mathcal{D}$ .
  end
  return the 2-intervals chosen for the solution.
end

```

Fig. 2. A schematic description of algorithm Bal- $\{\prec, \sqsubset, \emptyset\}$ -Approx

contained in the left or right interval of D_0 . Thus, there can be at most four disjoint intervals involved in \mathcal{D} , which intersect D_0 at this given iteration. It follows that at this iteration, at most four 2-intervals in the optimal solution are omitted from \mathcal{D} . Applying this argument for all iterations of the algorithm yields the desired approximation factor guarantee. \square

Time Complexity. Given an input set of 2-intervals \mathcal{D} of size n , algorithm Bal- $\{\prec, \sqsubset, \emptyset\}$ -Approx can be implemented straightforwardly to run in $\mathcal{O}(n^2)$ time.

3 An Approximation Algorithm for the $\{\sqsubset, \emptyset\}$ Model

We next consider the 2-INTERVAL PATTERN problem over the $\{\sqsubset, \emptyset\}$ model. Recall that the 2-INTERVAL PATTERN problem over this model is **NP**-complete even for unitary 2-interval sets [11]. In the following we introduce a single algorithm which achieves different constant approximation factors for unitary, balanced and unlimited 2-interval sets. More specifically, we describe an algorithm which uses the algorithms described in the previous section as sub-procedures, choosing the specific algorithm according to the restriction imposed on the input. Our algorithm is a direct generalization of the algorithm devised in [3] for the 2-INTERVAL PATTERN problem over the $\{\sqsubset, \emptyset\}$ model, restricted to point 2-interval sets. As in [3], the notion of *interval graphs* is used extensively throughout the section. An interval graph is an intersection graph of a finite family of intervals, all defined over a single line [7,10].

Given a 2-interval $D = (I, J)$, let $C(D)$ denote the smallest interval that covers \mathcal{D} , *i.e.*, $C(D) = [l(I) : r(J)]$ where $l(I)$ and $r(J)$ are the left and right endpoints of I and J , respectively. Blin *et al.* [3] called $C(D)$ the *covering interval* of D . They also observed that any pair of disjoint 2-intervals are $\{\sqsubset, \emptyset\}$ -comparable if and only if their corresponding covering intervals intersect. Thus, given a set of 2-intervals \mathcal{D} , and the set $\mathcal{C}(\mathcal{D})$ of all covering intervals of 2-intervals in \mathcal{D} , any $\{\sqsubset, \emptyset\}$ -comparable subset $\mathcal{D}' \subseteq \mathcal{D}$ corresponds to a pairwise intersecting subset of $\mathcal{C}' \subseteq \mathcal{C}(\mathcal{D})$. However, the converse is not true as a pair of non-disjoint

2-intervals have corresponding intersecting covering intervals as well. Hence, a pairwise intersecting subset of $\mathcal{C}(\mathcal{D})$ can contain corresponding 2-intervals which are non-disjoint in \mathcal{D} .

Let \mathcal{D} be the input set of 2-intervals and $\mathcal{C}(\mathcal{D})$ be the set of covering intervals of all 2-intervals in \mathcal{D} . First, we construct the interval graph $\Omega_{\mathcal{C}(\mathcal{D})}$ of $\mathcal{C}(\mathcal{D})$. Since $\Omega_{\mathcal{C}(\mathcal{D})}$ is an interval graph, it has at most $|V(\Omega_{\mathcal{C}(\mathcal{D})})| = |\mathcal{D}|$ maximal (in containment order) cliques, and all these maximal cliques can be computed in polynomial time [6]. Note that any pair of 2-intervals with covering intervals in a maximal clique, are either nesting or crossing (but not preceding), or they are non-disjoint. Now, let OPT denote a maximum cardinality $\{\sqsubset, \boxtimes\}$ -comparable subset of \mathcal{D} and let $\mathcal{C}(OPT)$ be the set of covering intervals of OPT . The subgraph of $\Omega_{\mathcal{C}(\mathcal{D})}$ which corresponds to $\mathcal{C}(OPT)$ is a clique, and is thus a subset of a maximal clique in $\Omega_{\mathcal{C}(\mathcal{D})}$. Furthermore, any 2-interval with a covering interval in this clique and not in OPT is necessarily non-disjoint with at least one of the 2-intervals in OPT .

Observation 1. *Let OPT denote the maximum $\{\sqsubset, \boxtimes\}$ -comparable subset of \mathcal{D} . Then OPT is a maximum pairwise disjoint subset of a set of 2-intervals \mathcal{D}' , $OPT \subseteq \mathcal{D}' \subseteq \mathcal{D}$, such that $\mathcal{C}(\mathcal{D}')$, the covering intervals of OPT , corresponds to a maximal clique in $\Omega_{\mathcal{C}(\mathcal{D})}$.*

Given the 2-intervals which corresponds to a maximal clique in $\Omega_{\mathcal{C}(\mathcal{D})}$, one can use the algorithms in Section 2 to find an approximation of the maximum pairwise disjoint subset of these 2-intervals. A detailed schematic description of our algorithm, which is called $\{\sqsubset, \boxtimes\}$ -Approx, is given in Figure 3.

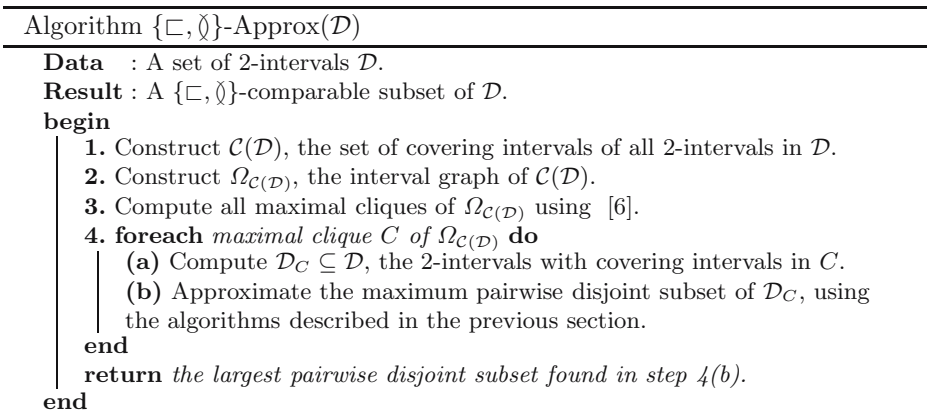


Fig. 3. A schematic description of algorithm $\{\sqsubset, \boxtimes\}$ -Approx

Lemma 2. *Algorithm $\{\sqsubset, \boxtimes\}$ -Approx is a 4-approximation (3-approximation) algorithm for the 2-INTERVAL PATTERN problem for unlimited (unitary) 2-interval sets.*

Proof. Immediate from the above discussion and from Proposition 1 and Lemma 1. \square

Time Complexity. The number of sub-procedure invocations in step 4(b) of $\{\square, \emptyset\}$ -Approx is bounded by $\mathcal{O}(n)$ where n denotes the size of the input set. Also, generating all maximal cliques of $\Omega_{\mathcal{C}(\mathcal{D})}$ can be done in $\mathcal{O}(n^2)$ time. Hence, we have a super-quadratic running time of $\mathcal{O}(n^2 \lg n)$ for unitary 2-interval sets and a $\mathcal{O}(n^3)$ running time for balanced 2-interval sets. For unlimited 2-interval sets, the running time of $\{\square, \emptyset\}$ -Approx is polynomial [2].

4 Approximation Algorithms for the $\{\prec, \emptyset\}$ Model

We now turn to considering the 2-INTERVAL PATTERN problem over the $\{\prec, \emptyset\}$ model. Recall that the problem is known to be **NP**-hard for unitary 2-interval sets, while for point 2-interval sets the problem is not known to be in **P** [3]. Thus, in the following section we consider all possible restrictions for the $\{\prec, \emptyset\}$ model. More specifically, we design a 3-approximation algorithm for unitary 2-interval sets which is also a 2-approximation algorithm for point 2-interval sets. We later slightly modify this algorithm to obtain a 5-approximation algorithm for balanced 2-interval sets. Finally, we introduce a different more complex modification which yields a 6-approximation algorithm for unlimited 2-interval sets.

Throughout the section, we will use the notion of *trapezoid graph* [4,5]. Consider two intervals, I' and J' , defined over two distinct horizontal lines. The trapezoid $T = (I', J')$ is the convex set of points bounded by I' and J' , and the two arcs connecting the right and left endpoints of I' and J' . We call I' and J' the *bottom interval* and *top interval* of T respectively. A family of trapezoids is a finite set of trapezoids which are all defined over the same two horizontal lines. The above definitions imply, that two distinct trapezoids $T_1 = (I'_1, J'_1)$ and $T_2 = (I'_2, J'_2)$ in a family of trapezoids are disjoint, *i.e.*, they contain no common point, if and only if $(I'_1 < I'_2 \text{ and } J'_1 < J'_2)$ or $(I'_2 < I'_1 \text{ and } J'_2 < J'_1)$ holds. If T_1 and T_2 are indeed disjoint, then one trapezoid is completely to left of the other, say for instance T_1 , and this is denoted by $T_1 < T_2$. Finally, a trapezoid graph is an intersection graph of a family of trapezoids.

4.1 Point and Unitary 2-Interval Sets

We begin the discussion in this section by first describing an approximation algorithm for point and unitary 2-interval sets. We call this initial algorithm $\{\prec, \emptyset\}$ -Approx. The general outline of $\{\prec, \emptyset\}$ -Approx consists of the following stages: First $\mathcal{T}(\mathcal{D})$, a family of trapezoids representing each 2-interval in \mathcal{D} is constructed. Next, the maximum pairwise disjoint subset of $\mathcal{T}(\mathcal{D})$ is computed using the algorithm proposed in [5]. Finally, trapezoids in this subset which correspond to non-disjoint 2-intervals in \mathcal{D} are omitted, and the filtered solution is outputted.

Definition 4 (Corresponding trapezoid family). Let \mathcal{D} be a set of 2-intervals, and let α and β be two distinct horizontal lines such that α is below β . The corresponding trapezoid family of \mathcal{D} , denoted $\mathcal{T}(\mathcal{D})$, is defined as the family containing a single trapezoid $T = (I', J') \in \mathcal{D}$ for each 2-interval $D = (I, J) \in \mathcal{D}$, where

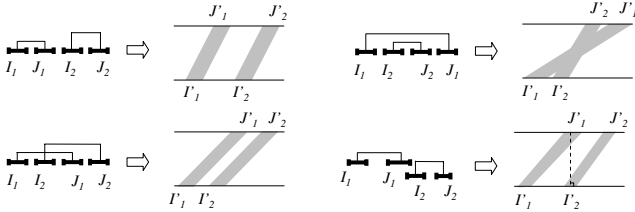


Fig. 4. $\{<, \checkmark\}$ -comparable 2-intervals correspond to disjoint trapezoids but the converse is not necessarily true

Let \mathcal{D} be a set of 2-intervals and let $\mathcal{T}(\mathcal{D})$ be the corresponding trapezoid family of \mathcal{D} . It is not difficult to see that $\{<, \checkmark\}$ -comparable 2-intervals in \mathcal{D} correspond to disjoint trapezoids in $\mathcal{T}(\mathcal{D})$, while $\{\square\}$ -comparable 2-intervals in \mathcal{D} correspond to intersecting trapezoids in $\mathcal{T}(\mathcal{D})$ (see Figure 4).

Observation 2. Any two disjoint 2-intervals in \mathcal{D} are $\{<, \checkmark\}$ -comparable if and only if their corresponding trapezoids in $\mathcal{T}(\mathcal{D})$ are disjoint.

Felsner *et al.* [5] presented an $\mathcal{O}(n \lg n)$ algorithm for finding a maximum disjoint subset in a family of n trapezoids. Unfortunately, this alone does not suffice in our case since there may be disjoint trapezoids in $\mathcal{T}(\mathcal{D})$ which correspond to non-disjoint 2-intervals in \mathcal{D} . (see Figure 4).

Definition 5 (Clashing intervals). Let $I' = [l(I'), r(I')]$ and $J' = [l(J'), r(J')]$ be two distinct intervals defined over two distinct horizontal lines such that $l(I') \leq l(J')$. The two intervals I' and J' clash, if either $l(I') \leq l(J') \leq r(J') \leq r(I')$ or $l(I') \leq l(J') \leq r(I') \leq r(J')$.

Definition 6 (Clashing trapezoids). Let $T_1 = (I'_1, J'_1)$ and $T_2 = (I'_2, J'_2)$ be two distinct trapezoids in a family of trapezoids. The two trapezoids T_1 and T_2 clash, if either I'_1 and J'_2 clash or I'_2 and J'_1 clash.

Observation 3. Any pair of 2-intervals in \mathcal{D} are $\{<, \checkmark\}$ -comparable if and only if their corresponding trapezoids in $\mathcal{T}(\mathcal{D})$ are disjoint and do not clash.

Observation 3 is the heart of algorithm $\{<, \checkmark\}$ -Approx. Note that the number of maximal (in containment order) pairwise disjoint subsets of $\mathcal{T}(\mathcal{D})$ can be exponential, so exhaustively searching through all such subsets for a maximum non-clashing subset is unfeasible. Let \mathcal{T}' be the maximum pairwise disjoint subset of $\mathcal{T}(\mathcal{D})$. Since the optimal solution $OPT \subseteq \mathcal{D}$ also corresponds to a pairwise

disjoint non-clashing subset of trapezoids, we must have $|OPT| \leq |T'|$. Next we show how to obtain a pairwise non-clashing subset of T' which is no more than a constant factor smaller than OPT , in case \mathcal{D} is either a point or unitary 2-interval set. Namely, we find a subset of T' which is an approximation of OPT .

Consider the leftmost trapezoid T_0 of T' and let D_0 be its corresponding 2-interval in \mathcal{D} . By our definition of a 2-interval and of $\mathcal{T}(\mathcal{D})$, any trapezoid in $\mathcal{T}(\mathcal{D})$, has a bottom interval which is completely to the left of its top interval. Thus, T_0 can only clash with trapezoids on its right in T' . Now, if \mathcal{D} is a point 2-interval set, then all 2-intervals with left intervals intersecting the right interval of D_0 have the same left interval, and as T' is pairwise disjoint, at most one of these has a corresponding trapezoid in T' . Furthermore, if \mathcal{D} is a unitary 2-interval set, intersecting intervals involved in \mathcal{D} must overlap. Thus, any trapezoid in T' clashing with T_0 corresponds to a 2-interval with a left interval which contains either endpoints, but not both, of the right interval of D_0 . Since T' is pairwise disjoint, there can be at most two such trapezoids in T' .

Algorithm $\{<, \emptyset\}$ -Approx first computes T' the maximum pairwise disjoint subset of $\mathcal{T}(\mathcal{D})$, and then repeatedly adds the leftmost trapezoids in T' to the solution while omitting all trapezoids which clash with this trapezoid in T' . A schematic description of algorithm $\{<, \emptyset\}$ -Approx is given in Figure 5.

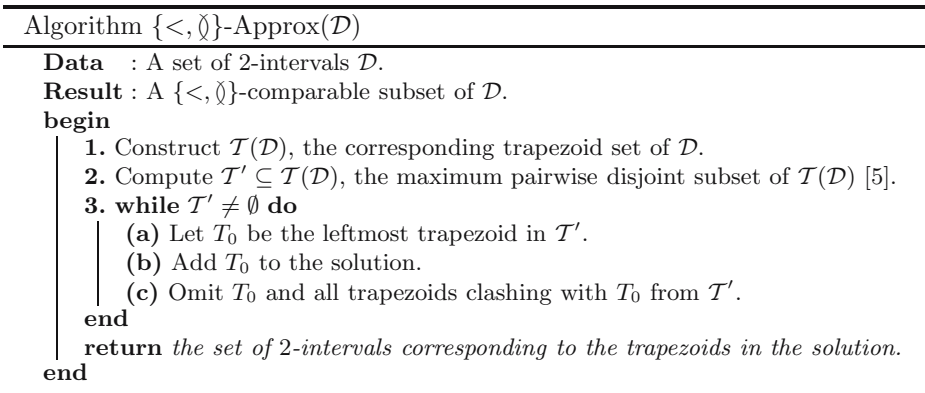


Fig. 5. A schematic description of algorithm $\{<, \emptyset\}$ -Approx

Lemma 3. *Algorithm $\{<, \emptyset\}$ -Approx is a 3-approximation algorithm (2-approximation algorithm) for the 2-INTERVAL PATTERN problem over the $\{<, \emptyset\}$ model restricted to unitary 2-interval sets (point 2-interval sets).*

Time Complexity. Let $|\mathcal{D}| = n$. The family of trapezoids $\mathcal{T}(\mathcal{D})$ can be constructed in $\mathcal{O}(n)$ time, and according to [5], $T' \subseteq \mathcal{T}(\mathcal{D})$ can be computed in $\mathcal{O}(n \lg n)$ time. In addition, each iteration in step 3 of the algorithm can easily be computed by scanning T' a constant number of times. As there are $\mathcal{O}(n)$ iterations all together, it follows that step 3, and consequently algorithm $\{<, \emptyset\}$ -Approx, can be computed in $\mathcal{O}(n^2)$ time. In fact, if we sort all the right endpoints

of intervals involved in \mathcal{D} in an $\mathcal{O}(n \lg n)$ preprocessing stage, we can compute each iteration of step 3 in linear time with respect to the number of trapezoids omitted. As there is only a constant number of such trapezoids in each iteration, step 3 can be computed in $\mathcal{O}(n)$ time. This yields a total of $\mathcal{O}(n \lg n)$ running time.

4.2 Balanced 2-Interval Sets

We next consider balanced 2-interval sets. We show that a slight modification to algorithm $\{<, \emptyset\}$ -Approx yields a 5-approximation algorithm for this case. We call this new algorithm Bal- $\{<, \emptyset\}$ -Approx. Algorithm Bal- $\{<, \emptyset\}$ -Approx differs from $\{<, \emptyset\}$ -Approx only by the fact that at each iteration of step 3, instead of choosing the leftmost trapezoid in \mathcal{T}' , the smallest trapezoid (*i.e.*, the trapezoid corresponding to the smallest 2-interval) amongst all trapezoids in \mathcal{T}' is chosen for the solution.

Lemma 4. *Algorithm Bal- $\{<, \emptyset\}$ -Approx is a 5-approximation factor the 2-INTERVAL PATTERN problem over the $\{<, \emptyset\}$ model restricted to balanced 2-interval sets.*

Proof. Consider \mathcal{T}' at an arbitrary iteration of step 3 in Bal- $\{<, \emptyset\}$ -Approx, and let T_0 be the smallest trapezoid of \mathcal{T}' at this iteration. Also let OPT denote the maximum $\{<, \emptyset\}$ -comparable subset of \mathcal{D} . Since T_0 is the smallest trapezoid, by a similar argument used in Lemma 1, T_0 clashes at most 4 other trapezoids in \mathcal{T}' at this iteration. Hence, since $|OPT| \leq |\mathcal{T}'|$ prior to step 3, the promised approximation factor is obtained and the above lemma holds. \square

Time Complexity. Algorithm Bal- $\{<, \emptyset\}$ -Approx can be implemented straightforwardly to run in $\mathcal{O}(n^2)$ time, where $n = |\mathcal{D}|$.

4.3 Unlimited 2-Interval Sets

The rest of this section is devoted to the 2-INTERVAL PATTERN problem over the $\{<, \emptyset\}$ model for unlimited 2-interval sets. We introduce a slightly more delicate modification of $\{<, \emptyset\}$ -Approx to obtain a 6-approximation algorithm for the unlimited case. For this, we consider special trapezoid families which have structures that are convenient for our purposes.

Definition 7 (Proper trapezoid family). *A family of trapezoids \mathcal{T} is proper if for any two distinct trapezoids $T_1 = (I'_1, J'_1), T_2 = (I'_2, J'_2)$ in \mathcal{T} , $I'_1 \cap I'_2 = \emptyset$ and $J'_1 \cap J'_2 = \emptyset$ holds.*

Definition 8 (Strongly proper trapezoid family). *A proper family of trapezoids \mathcal{T} is strongly proper if for any two distinct trapezoids $T_1 = (I'_1, J'_1), T_2 = (I'_2, J'_2)$ in \mathcal{T} , if J'_1 and I'_2 clash then $l(J'_1) \leq l(I'_2) < r(I'_2) \leq r(J'_1)$, where $l(J'_1), r(J'_1)$ and $l(I'_2), r(I'_2)$ are the left and right endpoints of J'_1 and I'_2 respectively.*

Note that by the above definition, any pairwise disjoint family of trapezoids is proper, but the converse is not true. Thus, $\mathcal{T}' \subseteq \mathcal{T}$ computed at step 2 of $\{\prec, \emptyset\}$ -Approx is a proper trapezoid family. Also, computing a strongly proper subset $\mathcal{T}'' \subseteq \mathcal{T}'$ can be done easily by adjusting step 3 of $\{\prec, \emptyset\}$ -Approx. Instead of omitting all trapezoids clashing with the leftmost trapezoid in this iteration, we need only to omit a small subset of these trapezoids. More specifically, let $T_0 = (I'_0, J'_0)$ be the leftmost trapezoid in \mathcal{T}' . We only omit trapezoids $T_\alpha = (I'_\alpha, J'_\alpha) \in \mathcal{T}'$ with either $l(I'_\alpha) \leq l(J'_0) \leq r(I'_\alpha)$ or $l(I'_\alpha) \leq r(J'_0) \leq r(I'_\alpha)$ (or both). It is not difficult to see that we obtain a strongly proper trapezoid family $\mathcal{T}'' \subseteq \mathcal{T}'$ if we proceed in this fashion and that $|\mathcal{T}''| \geq \frac{1}{3}|\mathcal{T}'|$.

Definition 9 (Clashing trapezoid graph). *Given a family \mathcal{T} of trapezoids, the clashing trapezoid graph of \mathcal{T} , denoted by $G_{\mathcal{T}}$, is the graph such that each vertex in $V(G_{\mathcal{T}})$ corresponds to a distinct trapezoid in \mathcal{T} , and two vertices are connected by an edge if and only if their corresponding trapezoids clash.*

Lemma 5. *Let \mathcal{T} be a family of trapezoids. If \mathcal{T} is strongly proper then $G_{\mathcal{T}}$ is a forest.*

Proof. Let \mathcal{T} be a strongly proper family of trapezoids and let $G_{\mathcal{T}}$ be its corresponding clashing trapezoid graph. Define $G_{\mathcal{T}}^*$ as the directed graph obtained by orientating the edges of $G_{\mathcal{T}}$ according to the precedence relation of \mathcal{T} . In other words, $V(G_{\mathcal{T}}^*) = V(G_{\mathcal{T}})$ and $(T_1, T_2) \in E(G_{\mathcal{T}}^*)$ if and only if $\{T_1, T_2\} \in E(G_{\mathcal{T}})$ and $T_1 < T_2$ in \mathcal{T} . Since \mathcal{T} is strongly proper, every trapezoid in \mathcal{T} clashes with at most one trapezoid on its left, and so the in-degree of every vertex $v \in V(G_{\mathcal{T}}^*)$ is at most one. Hence, any cycle (v_0, \dots, v_t, v_0) in $G_{\mathcal{T}}$ is a (directed) cycle in $G_{\mathcal{T}}^*$. However, in such a case we must have $T_0 < T_t < T_0$ by definition of $G_{\mathcal{T}}^*$, which is clearly a contradiction. Hence, we conclude that $G_{\mathcal{T}}^*$, and consequently $G_{\mathcal{T}}$, contain no cycles, and the above lemma holds. \square

It is well known that the maximum independent set in any forest G is of size at least $\frac{1}{2}|V(G)|$ and that this set can be found in linear time with respect to $|V(G)|$. Also, by definition, if \mathcal{T}'' is a pairwise disjoint family of trapezoids, then any independent set of $G_{\mathcal{T}''}$ corresponds to a pairwise disjoint non-clashing set of trapezoids, which by Observation 3, corresponds to a $\{\prec, \emptyset\}$ -comparable set of 2-intervals. A schematic description of our algorithm for unlimited 2-intervals sets, called $\text{Unl-}\{\prec, \emptyset\}$ -Approx, is given in Figure 6.

Lemma 6. *Algorithm $\text{Unl-}\{\prec, \emptyset\}$ -Approx is a 6-approximation algorithm for the 2-INTERVAL PATTERN problem over the $\{\prec, \emptyset\}$ model.*

Proof. Let \mathcal{D} be the input set of 2-intervals and let $\mathcal{T}(\mathcal{D})$, \mathcal{T}' and \mathcal{T}'' be the trapezoid families as described in the above description of $\text{Unl-}\{\prec, \emptyset\}$ -Approx. Also, denote by OPT the maximum $\{\prec, \emptyset\}$ -comparable subset of \mathcal{D} . We have $|\text{OPT}| \leq |\mathcal{T}'|$ and $|\mathcal{T}'| \leq 3|\mathcal{T}''|$. Let $\alpha(G_{\mathcal{T}''})$ denote the size of the maximal independent set of $G_{\mathcal{T}''}$. Since $G_{\mathcal{T}''}$ is a forest, we have $|V(G_{\mathcal{T}''})| \leq 2\alpha(G_{\mathcal{T}''})$. Accumulating all these inequalities together we get: $|\text{OPT}| \leq |\mathcal{T}'| \leq 3|\mathcal{T}''| = 3|V(G_{\mathcal{T}''})| \leq 6\alpha(G_{\mathcal{T}''})$. Thus, the maximum independent set of $G_{\mathcal{T}''}$ is at least of size $\frac{1}{6}|\text{OPT}|$, and the promised approximation factor holds. \square

Algorithm Unl- $\{<, \bar{\}\}$ -Approx(\mathcal{D})

Data : A set of 2-intervals \mathcal{D} .

Result : A $\{<, \bar{\}\}$ -comparable subset of \mathcal{D} .

begin

1. Construct $\mathcal{T}(\mathcal{D})$, the corresponding trapezoid set of \mathcal{D} .

2. Compute \mathcal{T}' , the maximum pairwise disjoint subset of $\mathcal{T}(\mathcal{D})$.

3. Compute \mathcal{T}'' , a strongly proper subset of \mathcal{T}' , such that $|\mathcal{T}''| \geq \frac{1}{3}|\mathcal{T}'|$.

4. Compute $G_{\mathcal{T}''}$ and the maximum independent set of $G_{\mathcal{T}''}$.

return the set of 2-intervals corresponding to the maximum independent set of $G_{\mathcal{T}''}$.

end

Fig. 6. A schematic description of algorithm Unl- $\{<, \bar{\}\}$ -Approx

Time Complexity. Let $|\mathcal{D}| = n$. Steps 1-2 in Unl- $\{<, \bar{\}\}$ -Approx can be computed in $\mathcal{O}(n \lg n)$ time by a similar analysis of the time complexity of $\{<, \bar{\}\}$ -Approx. Step 3 can be computed straightforwardly in $\mathcal{O}(n^2)$ time. Finally, step 4 can be computed in $\mathcal{O}(n)$ time since $G_{\mathcal{T}''}$ is a forest. Thus, the whole algorithm can be implemented to run in $\mathcal{O}(n^2)$ time.

References

1. T. Akutsu. Dynamic programming algorithms for RNA secondary structure prediction with pseudoknots. *Discrete Applied Mathematics*, 104:45-62, 2000.
2. R. Bar-Yehuda, M.M. Halldorsson, J. Naor, H. Shachnai and I. Shapira. Scheduling spit intervals. *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2002)*, 732-741.
3. G. Blin, G. Fertin and S. Vialette. New results for the 2-interval pattern problem. *Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching (CPM 2004)*, Lecture Notes in Computer Science 3109, Springer-Verlag, 311-322.
4. I. Dagan, M.C. Golumbic and R.Y. Pinter. Trapezoid graphs and their coloring. *Discrete Applied Mathematics*, 21:35-46, 1988.
5. S. Felsner, R. Müller and L. Wernisch. Trapezoid graphs and generalizations: Geometry and algorithms. *Discrete Applied Mathematics*, 74:13-32, 1997.
6. F. Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques and maximum independent set of a chordal graph. *SIAM Journal on Computing*, 1:180-187, 1972.
7. M.C. Golumbic. *Algorithmic graph theory and perfect graphs*. Academic Press, New York, 1980.
8. S. Jeong, M.Y. Kao, T.W. Lam, W.K. Sung and S.M. Yiu. Predicting RNA secondary structures with arbitrary pseudoknots by maximizing the number of stacking pairs. *Proceedings of the 2nd Symposium on Bioinformatics and Bioengineering (BIBE 2002)*, 183-190.
9. R.B. Lyngsø, C.N.S. Pedersen. RNA pseudoknot prediction in energy based models. *Journal of Computational Biology*, 7:409-428, 2000.
10. T.A. McKee, F.R. McMorris. *Topics in intersection graph theory*. SIAM monographs on discrete mathematics and applications, 1999.
11. S. Vialette. On the computational complexity of 2-interval pattern matching problems. *Theoretical Computer Science*, 312:335-379, 2004.

A Loopless Gray Code for Minimal Signed-Binary Representations

Gurmeet Singh Manku¹ and Joe Sawada²

¹ Google Inc., USA

manku@cs.stanford.edu

<http://www.cs.stanford.edu/~manku>

² University of Guelph, Canada

sawada@cis.uoguelph.ca

<http://www.cis.uoguelph.ca/~sawada>

Abstract. A string $\dots a_2 a_1 a_0$ over the alphabet $\{-1, 0, 1\}$ is said to be a minimal signed-binary representation of an integer n if $n = \sum_{k \geq 0} a_k 2^k$ and the number of non-zero digits is minimal. We present a loopless (and hence a Gray code) algorithm for generating all minimal signed binary representations of a given integer n .

1 Introduction

A string $\dots a_2 a_1 a_0$ is said to be a signed-binary representation (SBR) of an integer n if $n = \sum_{k \geq 0} a_k 2^k$ and $a_k \in \{-1, 0, 1\}$ for all k . A *minimal* SBR has the least number of non-zero digits. For example, 45 has five minimal SBRs: 101101, 110 $\bar{1}$ 01, 10 $\bar{1}$ 0 $\bar{1}$ 01, 10 $\bar{1}$ 00 $\bar{1}$ $\bar{1}$ and 1100 $\bar{1}$ $\bar{1}$, where $\bar{1}$ denotes -1 . Our main result is a *loopless* algorithm that generates *all* minimal SBRs for an integer n in Gray code order. See Fig. 1 for an example. Our algorithm requires linear time for generating the first string. Thereafter, only $O(1)$ time is required in the worst-case for identifying the portion of the current string to be modified for generating the next string¹.

Volumes 3 and 4 of Knuth's *The Art of Computer Programming* are devoted entirely to algorithms for generation of combinatorial objects. For the output of such an algorithm to be considered a Gray code, successive objects must differ by a constant amount. However, the time required to obtain each new object may be $\omega(1)$. A generation algorithm is said to be loopless if after the initial object is generated, successive objects may be obtained in $O(1)$ time in the worst-case. For a survey of Gray code generation algorithms, see Savage [20].

The earliest algorithm for listing all minimal SBRs is due to Ganesan and Manku [8]; however they did not consider the efficiency of implementing their algorithm. By modifying their technique, Sawada [21] was able

```
110100 $\bar{1}$ 10 $\bar{1}$ 
10 $\bar{1}$ 0100 $\bar{1}$ 10 $\bar{1}$ 
10 $\bar{1}$ 100 $\bar{1}$ 10 $\bar{1}$ 
10 $\bar{1}$ 10 $\bar{1}$ 010 $\bar{1}$ 
10 $\bar{1}$ 010 $\bar{1}$ 010 $\bar{1}$ 
11010 $\bar{1}$ 010 $\bar{1}$ 
110011010 $\bar{1}$ 
10 $\bar{1}$ 0011010 $\bar{1}$ 
10 $\bar{1}$ 00110011
1100110011
11010 $\bar{1}$ 0011
10 $\bar{1}$ 010 $\bar{1}$ 0011
10 $\bar{1}$ 10 $\bar{1}$ 0011
```

Fig. 1. A Gray code listing of minimal SBRs for 819. Successive strings differ in three adjacent positions.

¹ See <http://www.cs.stanford.edu/~manku/projects/graycode/index.html> for source code in C.

generate all minimal SBRs in constant amortized time. Additionally, the output constitutes a Gray code. However, the algorithm is not loopless, since successive strings require linear time in the worst case.

Our approach is novel — we first identify the *canonical* minimal SBR (see §2 for its definition). The canonical SBR is split into disjoint “chains”. Individual chains are handled by a Gray code algorithm which never outputs certain forbidden strings (see §4). The cross-product of all the chains is handled by a generalization of the Binary Reflected Gray Code (BRGC) [10, 3] (see §3 and §5). A detailed history of SBRs is presented in §6.

2 A Loopless Gray Code for Minimal SBRs

From earlier work by Sawada [21], we know that any minimal signed binary representation (SBR) for an integer n can be transformed into another minimal SBR for the same integer by repeated application of the following re-write rules: $10\bar{1} \rightarrow 011$, $011 \rightarrow 10\bar{1}$, $\bar{1}01 \rightarrow 0\bar{1}\bar{1}$, and $0\bar{1}\bar{1} \rightarrow \bar{1}01$. Our strategy for listing minimal SBRs in Gray code order is the following. We study the structural properties of a specific minimal SBR, popularly known as the *canonical* SBR. We then develop a procedure for listing all strings that result from repeated application of the four re-write rules to the canonical SBR.

DEFINITION (Canonical SBR). *Let S denote the binary representation of a given integer, padded with two leading zeros. For instance, integer 45 would correspond to the string $S = 00101101$. $S_{canonical}$ is the unique minimal SBR for S such that the product of any two adjacent digits is 0. Thus we never have 11, $1\bar{1}$, $\bar{1}1$ or $\bar{1}\bar{1}$ as a substring. For example,*

$$S = 00101011111010000001010110101000010100$$

$$S_{canonical} = 010\bar{1}0\bar{1}0000\bar{1}010000010\bar{1}0\bar{1}0\bar{1}0101000010100$$

$S_{canonical}$ has been used by previous authors (Reitwiesner [19], Chang and Tsao-Wu [6], Jedwab and Mitchell [12] and Prodinger [18]). In fact, $S_{canonical}$ for integer n can be obtained by “bit-wise subtracting $n/2$ from $3n/2$ ” (Prodinger [18]). Starting with $S_{canonical}$ is critical to the simplicity of our approach.

DEFINITION (Blocks). *A maximally long bit-sequence of $(01)^+$ and $(0\bar{1})^+$ in $S_{canonical}$ is called a block. The following string has eight blocks (each block has been underlined):*

$$\underline{01} \underline{0\bar{1}0\bar{1}} \ 000 \underline{0\bar{1}} \ \underline{01} \ 0000 \ \underline{01} \ \underline{0\bar{1}0\bar{1}0\bar{1}} \ \underline{0101} \ 000 \ \underline{0101} \ 00$$

DEFINITION (Chains). *A chain is a maximally long sequence of two or more adjacent blocks. The following string has three chains (each chain has been underlined):*

$$\underline{\underline{01} \underline{0\bar{1}0\bar{1}} \ 000} \ \underline{\underline{0\bar{1}} \ \underline{01}} \ 0000 \ \underline{\underline{01} \ \underline{0\bar{1}0\bar{1}0\bar{1}} \ \underline{0101}} \ 000 \ 0101 \ 00$$

Two chains are separated by one or more 0s. Therefore, none of the four rewrite rules, when applied to one chain, affects another chain. This proves the following:

Theorem 1. *The set of minimal SBRs of S corresponds to the cross product of the sets of minimal SBRs for individual chains of $S_{canonical}$.*

We now develop two loopless algorithms: one for generating the minimal SBRs of a chain in Gray code order (see §5), and another for generating the cross-product of Gray codes (see §3).

3 Gray Codes for Cross-Products

Consider the cross product of m combinatorial objects: $X_m \times X_{m-1} \times \dots \times X_1$, where object X_i has $t_i \geq 2$ members which can be listed in Gray code order. Clearly, there is a 1-1 correspondence between members of the cross product and tuples of the form $(a_m, a_{m-1}, \dots, a_1)$, where $a_i \in [1, t_i]$ represents the a_i -th object in the Gray code of X_i . When each $t_i = 2$, one possible Gray code for the set of tuples is the Binary Reflected Gray Code (BRGC) [10]. A generalization of the BRGC, developed by Bitner, Ehrlich, and Reingold [3], handles arbitrary values of $t_i \geq 2$. Procedure BRGC (displayed in Fig. 2) is such an algorithm.

Procedure BRGC maintains three tuples:

$(a_m, a_{m-1}, \dots, a_1)$ is the *current-tuple*,
 $(d_m, d_{m-1}, \dots, d_1)$ is the *direction-tuple*, and
 $(p_{m+1}, p_m, \dots, p_1)$ is the *pointer-tuple*.

INITIALIZE initializes the three tuples.

The current-tuple has $a_i = 1$ or $a_i = t_i$, chosen arbitrarily. The direction-tuple has initial value $d_i = 1$ if $a_i = 1$; otherwise $d_i = -1$. The pointer-tuple has initial value $(m + 1, m, m - 1, \dots, 1)$.

NEXT(i) updates $a_i \leftarrow a_i + d_i$.

IS_TERMINAL(i) returns TRUE iff $(a_i = t_i$ and $d_i = 1)$ or $(a_i = 1$ and $d_i = -1)$.

The pointer-tuple lies at the heart of procedure BRGC. If $p_1 = m + 1$, procedure BRGC terminates. Otherwise, let $i = p_1$. Then a_i , the i -th member of the current-tuple, is modified. The direction-tuple indicates whether to increment ($d_i = 1$) or decrement ($d_i = -1$) the value of a_i .

Sample output produced by the algorithm is shown in Table 1(A).

Procedure BRGC can easily be adapted to generate members of $X_m \times X_{m-1} \times \dots \times X_1$ in Gray code order. Clearly, such an algorithm is loopless if the algorithm that generates members of each X_i in Gray code order is loopless.

4 Gray Codes for Cross-Products with Forbidden Tuples

Let R_m denote the set of m -tuples $(a_m, a_{m-1}, \dots, a_1)$ satisfying

- 1) $\forall m \geq i \geq 1 : a_i \in [1, t_i]$, with $t_i \geq 2$
- 2) $\forall m \geq i > 1 : (a_i = t_i) \Rightarrow (a_{i-1} = 1)$

```

BRGC

INITIALIZE

WHILE TRUE DO
  last ← 1
  i ← plast
  IF (i = m + 1) THEN exit

  NEXT(i)

  IF (IS_TERMINAL(i)) THEN
    di ← -di
    j ← i + 1
    pi ← pj
    pj ← j

  IF (i ≠ last) THEN plast ← last
    
```

Fig. 2. A generalization of the Binary Reflected Gray Code [10,3]. See Table 1(A) for sample output.

Table 1. Output of BRGC (Fig 2) and BRGC-RESTRICT (Fig 3) for $t_3 = 2, t_2 = 3, t_1 = 3$. The initial tuple is $(a_3, a_2, a_1) = (1, 3, 1)$. The output is generated after each iteration of the WHILE loop. For simplicity we use ‘-’ to represent -1.

(A) With BRGC								(B) With BRGC-RESTRICT											
a_3	a_2	a_1	p_4	p_3	p_2	p_1	d_3	d_2	d_1	a_3	a_2	a_1	p_4	p_3	p_2	p_1	d_3	d_2	d_1
1	3	1	4	3	2	1	1	-	1	1	3	1	4	3	2	1	1	-	1
1	3	2	4	3	2	1	1	-	1	1	2	1	4	3	2	1	1	-	1
1	3	3	4	3	2	2	1	-	-	1	2	2	4	3	2	1	1	-	1
1	2	3	4	3	2	1	1	-	-	1	2	3	4	3	2	2	1	-	-
1	2	2	4	3	2	1	1	-	-	1	1	3	4	3	3	1	1	1	-
1	2	1	4	3	2	2	1	-	1	1	1	2	4	3	3	1	1	1	-
1	1	1	4	3	3	1	1	1	1	1	1	1	4	3	2	3	1	1	1
1	1	2	4	3	3	1	1	1	1	2	1	1	4	4	2	1	-	1	1
1	1	3	4	3	2	3	1	1	-	2	1	2	4	4	2	1	-	1	1
2	1	3	4	4	2	1	-	1	-	2	1	3	4	3	2	4	-	1	-
2	1	2	4	4	2	1	-	1	-										
2	1	1	4	4	2	2	-	1	1										
2	2	1	4	4	2	1	-	1	1										
2	2	2	4	4	2	1	-	1	1										
2	2	3	4	4	2	2	-	1	-										
2	3	3	4	3	4	1	-	-	-										
2	3	2	4	3	4	1	-	-	-										
2	3	1	4	3	2	4	-	-	1										

For example, with $t_3 = 2, t_2 = 3,$ and $t_1 = 3, R_m$ consists of 3-tuples listed in Table 1(B). We now develop a loop-free algorithm for listing R_m in Gray code order. This algorithm will be used in §5 for listing minimal SBRs of chains.

Let G_m denote a Gray code for R_m . Then the reversal of G_m , denoted \overline{G}_m , is also a Gray code. We define G_m recursively as follows. The base cases are $G_0 = ()$, the empty tuple, and $G_1 = (1), (2), \dots, (t_1)$. For $m \geq 1, G_{m+1}$ depends upon the parity (odd/even) of both t_{m+1} and t_m . Four cases arise; the sequence of $(m + 1)$ -tuples for G_{m+1} for the four cases is defined below.

(even, even)	(even, odd)	(odd, even)	(odd, odd)
$1\overline{G}_m,$	$1\overline{G}_m,$	$1G_m,$	$1G_m,$
$2\overline{G}_m,$	$2\overline{G}_m,$	$2\overline{G}_m,$	$2\overline{G}_m,$
$3\overline{G}_m,$	$3\overline{G}_m,$	$3G_m,$	$3G_m,$
$4\overline{G}_m,$	$4\overline{G}_m,$	$4\overline{G}_m,$	$4\overline{G}_m,$
\dots	\dots	\dots	\dots
$t_m\overline{G}_m,$	$t_m\overline{G}_m,$	$t_m\overline{G}_m,$	$t_m\overline{G}_m,$
$t_{m+1}1\overline{G}_{m-1}$	$t_{m+1}1G_{m-1}$	$t_{m+1}1\overline{G}_{m-1}$	$t_{m+1}1G_{m-1}$

The notation xG_i denotes a sequence of tuples with $i + 1$ members: the first member of each tuple is x ; the remaining members of the tuple constitute G_i . The last tuple in G_m is the same as the first tuple in \overline{G}_m and *vice versa*. Thus,

<u>BRGC-RESTRICT</u>	Procedure INITIALIZE:
INITIALIZE	FOR $i \leftarrow m + 1$ DOWNTO 1 DO $p_i \leftarrow i$
WHILE TRUE DO	$a_m \leftarrow d_m \leftarrow 1$
last \leftarrow MAP(1)	IF (EVEN(t_m)) THEN $rev \leftarrow$ true
$i \leftarrow$ MAP(p_{last})	ELSE $rev \leftarrow$ false
IF ($i = m + 1$) THEN exit	FOR $i \leftarrow m - 1$ DOWNTO 1 DO
NEXT(i)	IF $rev =$ false THEN
IF (IS_TERMINAL(i)) THEN	$a_i \leftarrow d_i \leftarrow 1$
$d_i \leftarrow -d_i$	IF (EVEN(t_i)) THEN $rev \leftarrow$ true
$j \leftarrow$ MAP($i + 1$)	ELSE
$p_i \leftarrow p_j$	$a_i \leftarrow t_i$
$p_j \leftarrow j$	$d_i \leftarrow -1$
IF ($i \neq last$) THEN $p_{last} \leftarrow last$	$i \leftarrow i - 1$
	$a_i \leftarrow d_i \leftarrow 1$
	IF (EVEN(t_i)) THEN $rev \leftarrow$ false

Fig. 3. A loopless algorithm for listing restricted cross products. See Table 1(B) for sample output.

since the first tuple in each listing begins with a one, G_{m+1} for $m \geq 1$ is indeed a Gray code for R_{m+1} .

Theorem 2. *Procedure BRGC-RESTRICT in Fig. 3 is a loopless algorithm for producing the Gray code G_m .*

BRGC-RESTRICT (Fig. 3) differs from BRGC (Fig. 2) in two important aspects:

1. The initial string $(a_m, a_{m-1}, \dots, a_1)$ has to be initialized appropriately (see procedure INITIALIZE). We begin by assigning $a_m \leftarrow 1$. The recursive definition of G_m then helps us determine the initial values for each a_i , where $m - 1 \geq i \geq 1$. To do this we need only keep track of whether or not a_i is the first member in the first i -tuple of G_i or \overline{G}_i . The variable rev is used determine the list. Recall that the direction d_i is initialized to 1 if $a_i = 1$. If $a_i = t_i$, then d_i is initialized to -1 . The initialization for the “pointer-tuple” p is the same as before: $(m + 1, m, m - 1, \dots, 1)$.
2. We employ a function MAP which is defined as follows:

$$\text{MAP}(i) = \begin{cases} i + 1 & \text{if } (m > i \geq 1) \text{ and } (a_i = 1) \text{ and } (a_{i+1} = t_{i+1}) \\ i & \text{otherwise} \end{cases}$$

If MAP(i) always returns i , then BRGC-RESTRICT would be identical to BRGC.

An interesting special case corresponds to $t_i = 2$ for all i . Then G_m consists of m -digit strings using the digits $\{1, 2\}$ in which 22 is a forbidden substring. The total number of such strings equals the $(m + 1)^{\text{st}}$ Fibonacci number.

5 A Loopless Gray Code for Chains

We begin with two examples for illustration of our approach.

EXAMPLE (Chain with 2 Blocks). Let $B_2B_1 = (0\bar{1})^s(01)^t$. A rewrite rule is applicable only where the two blocks join: $\bar{1}01 \rightarrow 0\bar{1}\bar{1}$, to obtain $(0\bar{1})^{s-1}00\bar{1}\bar{1}(01)^{t-1}$. Now, we could apply the inverse rule ($0\bar{1}\bar{1} \rightarrow \bar{1}01$) to obtain the previous string, or we can apply the same rule again to the unique substring $\bar{1}01$ in the new representation. This pattern will repeat until we reach the end of the chain. The number of minimal SBRs for this chain is $t + 1$ and is independent of s . As an example, if $s = 2$ and $t = 3$, then the 4 minimal SBRs of $0\bar{1}0\bar{1}010101$ will be: $0\bar{1}0\bar{1}010101, 0\bar{1}00\bar{1}\bar{1}0101, 0\bar{1}00\bar{1}0\bar{1}\bar{1}01$ and $0\bar{1}00\bar{1}0\bar{1}0\bar{1}\bar{1}$. Only B_1 is changing, except for the rightmost digit of B_2 that changes after the first rewrite. \square

EXAMPLE (Chain with 3 Blocks). Without loss of generality, let $B_3B_2B_1 = (0\bar{1})^s(01)^t(0\bar{1})^u$. In this case, we can again apply the rewrite rules between B_3 and B_2 as with the two block case, but now we can also apply similar rewrite rules between B_2 and B_1 . The only difference is that the rewrite rules between B_2 and B_1 can only be applied if the state of B_2 has not been altered to its final state where it ends with $\bar{1}\bar{1}$. In that case, no rewrite rules are possible between the two blocks (block B_1 must remain in its initial form: $(0\bar{1})^u$). If we ignore the leftmost block, observe that this problem is an instance of the restricted cross products (where $m = 2$) described in §4. \square

To generalize the above observations, we define

$$s(k, \ell) = \begin{cases} (01)^k & \text{if } \ell = 1 \\ (\bar{1}0)^{\ell-2}\bar{1}\bar{1}(01)^{k-\ell+1} & \text{if } 1 < \ell \leq k + 1 \end{cases}$$

For block $B_i = (01)^k$ (that is not the leftmost block of a chain), the sequence $s(k, 1), s(k, 2), \dots, s(k, k + 1)$ corresponds to the $k + 1$ different strings that the block B_i may cycle through. The string $\bar{s}(k, \ell)$ is defined similarly, with 1 and $\bar{1}$ interchanged throughout the string. Examples:

		$s(4, 1) = 01010101$	$\bar{s}(4, 1) = 0\bar{1}0\bar{1}0\bar{1}0\bar{1}$
$s(1, 1) = 01$	$\bar{s}(1, 1) = 0\bar{1}$	$s(4, 2) = \bar{1}\bar{1}010101$	$\bar{s}(4, 2) = 110\bar{1}0\bar{1}0\bar{1}$
$s(1, 2) = \bar{1}\bar{1}$	$\bar{s}(1, 2) = 11$	$s(4, 3) = \bar{1}0\bar{1}\bar{1}0101$	$\bar{s}(4, 3) = 10110\bar{1}0\bar{1}$
		$s(4, 4) = \bar{1}0\bar{1}0\bar{1}\bar{1}01$	$\bar{s}(4, 4) = 1010110\bar{1}$
		$s(4, 5) = \bar{1}0\bar{1}0\bar{1}0\bar{1}\bar{1}$	$\bar{s}(4, 5) = 10101011$

Using these strings we can now formally map the problem of cycling through all minimal SBRs of a chain $B_{m+1}B_m \dots B_1$ to the problem of generating restricted m -tuples. Without loss of generality assume that m is odd and that each B_i is initially defined as follows:

$$\begin{aligned} B_{m+1} &= \bar{s}(k_{m+1}, 1) = (0\bar{1})^{k_{m+1}}, \\ B_m &= s(k_m, 1) = (01)^{k_m}, \\ B_{m-1} &= \bar{s}(k_{m-1}, 1) = (0\bar{1})^{k_{m-1}}, \\ \dots & \quad \dots \quad \dots \\ B_2 &= s(k_2, 1) = (01)^{k_2}, \\ B_1 &= \bar{s}(k_1, 1) = (0\bar{1})^{k_1}. \end{aligned}$$

Then a listing of all minimal SBRs for the chain is a subset of the cross-product of strings in blocks B_m, B_{m-1}, \dots, B_1 , satisfying two constraints for $m \geq i > 1$:

- (1) If the string in block B_i equals $s(k_i, k_i + 1)$, then the string in block B_{i-1} must equal $\bar{s}(k_{i-1}, 1)$.
- (2) If the string in block B_i equals $\bar{s}(k_i, k_i + 1)$, then the string in block B_{i-1} must equal $s(k_{i-1}, 1)$.

A Gray code for the chain can be obtained by setting $t_i = k_i + 1$ for $m \geq i \geq 1$ and using BRGC-RESTRICT outlined in §4. There is a 1-1 correspondence between tuples generated by BRGC-RESTRICT and strings assigned to blocks. A tuple $(a_m, a_{m-1}, a_{m-2}, \dots, a_1)$ generated by BRGC-RESTRICT corresponds to the following configuration: string $s(k_m, a_m)$ in block B_m , string $\bar{s}(k_{m-1}, a_{m-1})$ in block B_{m-1} , string $s(k_{m-2}, a_{m-2})$ in block B_{m-2} , and so on. The only special consideration is that rightmost bit in the leftmost block B_{m+1} must be changed to 0 iff B_m is not in its original state. This is a trivial constant time operation.

Since BRGC-RESTRICT (Fig. 3) is loopless, we have a loopless algorithm to list all minimal SBRs for a given chain. For cross-product of chains (see Theorem 1) we apply procedure BRGC (Fig. 2).

Theorem 3. *A listing of all minimal SBRs for a given integer n can be generated by a loopless algorithm.*

6 A Brief History of Signed Binary Representations

Signed-digit representations have been investigated by both mathematicians and computer scientists (see Hwang [11], Parhami [16] and Knuth [13]). Signed-binary representations using the digits $\{-1, 0, 1\}$ were first investigated by Reitwiesner [19] and Avizienis [2] in the context of digital hardware. Reitwiesner presented an algorithm for identifying the *canonical* signed-binary representation, which is that representation in which no two adjacent digits are non-zero. Over the years, similar algorithms have been re-discovered by several authors (Chang and Tsao-Wu [6], Jedwab and Mitchell [12] and Prodinger [18]). A technique for identifying *all* minimal signed-binary representations, not just the canonical, was discovered by Ganesan and Manku [8]. Sawada [21] adapted this technique to list all minimal SBRs in Gray code order in constant amortized time.

The average weight of minimal signed-binary representations of b -bit numbers approaches $b/3$ for large b . This result has been re-discovered several times, using different proof techniques (Reitwiesner [19], Arno and Wheeler [1], Prodinger [18] and Ganesan and Manku [8]).

Sloane and Plouffe's sequence M0103 and Sloane's sequence A007302 correspond to the weights of minimal signed-binary representations of natural numbers. Sloane's Sequence A005578 are numbers n at which the weight of minimal signed-binary representations of n increases. Sloane's sequence A057526 is the number of zeros in minimal signed-binary representations of natural numbers.

For $m \geq 2$, $(\dots a_2 a_1 a_0)_m$ is said to be a "signed-digit representation" of n if $n = \sum_{k \geq 0} a_k m^k$ and $m_k \in \{0, \pm 1, \pm 2, \dots, \pm (m-1)\}$. A minimal representation

has the least number of non-zero digits. The general case $m \geq 2$ has appeared in early work by Avizienis [2]. Clark and Liang [7] defined a *canonical* representation as one satisfying two additional constraints: (a) $|a_{i+1} + a_i| < m$ for all i , and (b) $|a_i| < |a_{i+1}|$, if $a_{i+1}a_i < 0$, where $|a_i|$ denotes the absolute value of a_i . Such a representation is also known as a *generalized non-adjacent form* (GNAF) since it possesses the property that no two consecutive digits are non-zero for $m = 2$. The GNAF for any integer is minimal and unique. An algorithm for identifying the GNAF was presented in [7]. The average weight for b -digit numbers was shown to be asymptotically $\frac{m-1}{m+1}b$ by Arno and Wheeler [1]. Wu and Hasan [26] derive closed-form formulae for the same. These results were re-discovered by Ganesan and Manku [8].

6.1 Fast Exponentiation

Fast computation of $x^n \bmod r$ is very valuable in cryptography (see surveys by Koç [14] and Gordon [9]). Exponentiation can be studied in terms of addition chains and addition-subtraction chains.

An addition chain for integer n is a sequence of values $a_0 = 1, a_1, a_2, \dots, a_r = n$ with the property that for each $i > 0$, there exist j and k such that $a_i = a_j + a_k$. Then x^n can be computed with r multiplications. See Knuth [13] for a survey of addition chains. The best known lower-bound is $\log_2 n + \log_2 H(n) - 2.13$ by Schönhage [22]. An upper bound for the length of addition chains is $\lfloor \log_2 n \rfloor + H(n)$, where $H(n)$ denotes the Hamming weight of n (the number of 1-bits in binary representation of n). The upper bound is realized by the folklore “fast-multiplication algorithm”. For a randomly chosen b -bit exponent, $b/2$ bits are 1 on average; so the expected number of multiplications is $3b/2$. Several papers propose heuristics for reducing the average by discovering short addition chains (see Bos and Coster [4] and Yacobi [27], for example).

For evaluating $x^n \bmod r$ when x and r are fixed *a priori*, we can pre-compute $x^{-1} \bmod r$, enabling efficient “division” as well. Further, in elliptic curve cryptography, computing $x^{-1} \bmod r$ is as costly as computing $x \bmod r$. This leads us to the idea of addition-subtraction chains (described below), which reduces the average number of multiplications far below $3b/2$.

An addition-subtraction chain for integer n is a sequence of values $a_0 = 1, a_1, a_2, \dots, a_r = n$ with the property that for each $i > 0$, there exist j and k such that $a_i = \pm a_j \pm a_k$. Then x^n can be computed with r multiplications/divisions. Signed-binary representations correspond to addition-subtraction chains. For b -bit exponents, approximately $b/3$ bits are ± 1 ; so the average number of multiplications/divisions is roughly $4b/3$. Higher bases lead to further savings.

Addition-subtraction chains are useful for fast exponentiation in groups (Wu and Hasan [25], Brickell *et al* [5]). Their usefulness in elliptic curve cryptography was first pointed out by Morain and Olivos [15]. Conversion of an integer in binary to its minimal signed-digit representation is popularly known as *recoding*. Efficient software/hardware implementation of recoding presents its own unique challenges. This has led to a variety of recoding algorithms and generalizations of signed-digit representations by the cryptography community. For a good overview of recoding literature, see Phillips and Burgess [17].

6.2 Routing in Chord and CM-2

Weitzman [24] studied routing in the Connection Machine CM-2, developed by Thinking Machines in 1980s. CM-2 was a massively parallel computer using a hypercube-based inter-connection network for routing. Every processor could send a message to another processor a fixed distance $\pm 2^i$ away for any $i \geq 0$. Weitzman discovered that $F(n)$, the optimal cost of communication between two processors distance n away, was given by $F(0) = 0$, $F(2^k) = 1$ and $F(n) = 1 + \min(F(n - 2^k), F(2^{k+1} - n))$, for $2^k < n < 2^{k+1}$. The relationship between $F(n)$ and signed-binary representations was exposed by Ganesan and Manku [8]. They studied a peer-to-peer routing network called Chord [23]. In its simplest form, Chord is an undirected graph on 2^b nodes arranged in a circle, with edges connecting pairs of nodes that are 2^k positions apart for any $k \geq 0$. The shortest path for clockwise distance d can be identified by computing a minimal signed-binary representation of d' defined as follows [8]:

$$d' = \begin{cases} d & \text{if } d \leq \lfloor 2^b/3 \rfloor \\ 2^b - d & \text{if } d > \lfloor 2^b/3 \rfloor \\ d \text{ or } 2^b - d & \text{otherwise} \end{cases}$$

1 and $\bar{1}$ in the signed-binary representation correspond to clockwise and anti-clockwise traversals of Chord edges respectively. A variety of algorithms for solving the problem are presented in [8]. One of them is “LEFT-TO-RIGHT BIDIRECTIONAL GREEDY”, which is identical to Weitzman’s algorithm.

References

1. Steven Arno and Ferrell S Wheeler. Signed digit representations of minimal hamming weight. *IEEE Transactions on Computers*, 42(8):1007–1010, August 1993.
2. Algirdas A Avizienis. Signed-digit number representations for fast parallel arithmetic. *IRE Transactions on Electronic Computers*, 10:389–400, 1961.
3. James R Bitner, Gideon Ehrlich, and Edward M Reingold. Efficient generation of the binary reflected Gray code and its applications. *Communications of the ACM*, 19(9):517–521, September 1976.
4. J Bos and M Coster. Addition chain heuristics. In *Advances in Cryptology: CRYPTO 89 (LCNS No 435)*, pages 400–407, 1989.
5. E F Brickell, D M Gordon, K S McCurley, and D B Wilson. Fast exponentiation with precomputation. In *Proc. EUROCRYPT '92*, pages 200–207, 1992.
6. S H Chang and N Tsao-Wu. Distance and structure of cyclic arithmetic codes. In *Proc. Hawaii International Conference on System Sciences*, volume 1, pages 463–466, 1968.
7. W E Clark and J J Liang. On arithmetic weight for a general radix representation of integers. *IEEE Transactions on Information Theory*, 19:823–826, November 1973.
8. Prasanna Ganesan and Gurmeet Singh Manku. Optimal routing in Chord. In *Proc. 15th ACM-SIAM Symposium on Discrete Algorithms (SODA 2004)*, pages 169–178, January 2004.
9. Daniel M Gordon. A survey of fast exponentiation methods. *J of Algorithms*, 27(1):129–146, April 1998.

10. F Gray. Pulse code communications. U S Patent 2,632,058 (March 17, 1953), 1953.
11. Kai Hwang. *Computer Arithmetic: Principles, Architecture and Design*. John Wiley and Sons, Inc., 1979.
12. J Jedwab and C J Mitchell. Minimum weight modified signed-digit representations and fast exponentiation. *Electronic Letters*, 25(17):1171–1172, 1989.
13. Donald E Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, 3 edition, 1997.
14. Çetin Kaya Koç. High-speed RSA implementation. RSA Labs, November 1994.
15. François Morain and Jorge Olivós. Speeding up the computations on an elliptic curve using addition-subtraction chains. *RAIRO Informatique Théorique et Applications*, 24(6), 1990.
16. B Parhami. Generalized signed-digit number systems: A unifying framework for redundant number representations. *IEEE Transactions on Computers*, 39:89–98, 1990.
17. Braden Phillips and Neil Burgess. Minimal weight digit set conversions. *IEEE Transactions on Computers*, 53(6):666–677, June 2004.
18. Helmut Prodinger. On binary representations of integers with digits $-1, 0, 1$. *INTEGER: The Electronic Journal of Combinatorial Number Theory*, 0, 2000.
19. G W Reitwiesner. Binary arithmetic. *Advances in Computers*, 1:231–308, 1960.
20. Carla Savage. A survey of combinatorial Gray codes. *SIAM Review*, 39(4):609–625, 1997.
21. Joe Sawada. A Gray code for binary subtraction. In *2nd Brazilian Symposium on Graphs, Algorithms and Combinatorics (GRACO 2005)*, 2005.
22. Schönhage. A lower bound for the length of addition chains. *Theoretical Computer Science*, 1:1–12, 1975.
23. Ion Stoica, Robert Morris, D Liben-Lowell, David R Karger, M Frans Kaashoek, F Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, 2003.
24. A Weitzman. Transformation of parallel programs guided by micro-analysis. In B Salvy, editor, *Algorithms Seminar, 1992-1993*, pp. 155–159, Institut National de Recherche en Informatique et en Automatique, France, Rapport de Recherche, No. 2130 (Summarized by Paul Zimmermann), 1993.
25. H Wu and M A Hasan. Efficient exponentiation of a primitive root in $\text{GF}(2^m)$. *IEEE Transactions on Computers*, 46(2):162–172, February 1997.
26. Huapeng Wu and M Anwar Hasan. Closed-form expression for the average weight of signed-digit representations. *IEEE Transactions on Computers*, 48(8):848–851, August 1999.
27. Yacov Yacobi. Exponentiating faster with addition chains. In *Advances in Cryptography – EUROCRYPT 90: Workshop on the Theory and Application of Cryptographic Techniques*, page 222, 1990.

Efficient Approximation Schemes for Geometric Problems?*

Dániel Marx

Department of Computer Science and Information Theory,
Budapest University of Technology and Economics,
Budapest, H-1521, Hungary
dmarx@cs.bme.hu

Abstract. An EPTAS (efficient PTAS) is an approximation scheme where ϵ does not appear in the exponent of n , i.e., the running time is $f(\epsilon) \cdot n^c$. We use parameterized complexity to investigate the possibility of improving the known approximation schemes for certain geometric problems to EPTAS. Answering an open question of Alber and Fiala [2], we show that MAXIMUM INDEPENDENT SET is W[1]-complete for the intersection graphs of unit disks and axis-parallel unit squares in the plane. A standard consequence of this result is that the $n^{O(1/\epsilon)}$ time PTAS of Hunt et al. [11] for MAXIMUM INDEPENDENT SET on unit disk graphs cannot be improved to an EPTAS. Similar results are obtained for the problem of covering points with squares.

1 Introduction

We say that an optimization problem admits a polynomial-time approximation scheme (PTAS) if for every $\epsilon > 0$ there is a polynomial-time algorithm with relative error at most ϵ . A PTAS is an *efficient polynomial-time approximation scheme* (EPTAS) if this family of approximation algorithms is uniformly polynomial: for every $\epsilon > 0$, the running-time is $f(\epsilon) \cdot n^c$, where f is an arbitrary function of ϵ , and c is a constant independent of ϵ . For example, Arora [3] presented an $n^{O(1/\epsilon)}$ time PTAS for Euclidean TSP, which is not an EPTAS. However, in the journal version of the paper [4], the running-time of the algorithm is improved to $n \cdot \log^{O(1/\epsilon)} n = 2^{O(1/\epsilon^2)} \cdot n^2$, hence the problem admits an EPTAS.

Whenever a problem admits a PTAS, it should be examined if the algorithm can be improved to an EPTAS. The motivation comes from the observation that a polynomial-time algorithm is not really practical if the degree is larger than 3. Therefore, a $O(n^{2/\epsilon})$ time PTAS is not practical even for 20% error. In fact, the situation is much worse than that: as pointed out in [7], most approximation schemes in the literature have very high degrees even for 20% error. For example, the running time of the PTAS of [9] for finding the maximum weighted independent set in the intersection graph of disks is $O(n^{523804})$ for 20% error.

* Research is supported in part by grants OTKA 44733, 42559 and 42706 of the Hungarian National Science Fund.

Parameterized complexity gives us useful tools to investigate the question whether a PTAS can be improved to an EPTAS. Parameterized complexity deals with problems where the input instances have a distinguished part k called the parameter. For example, in the MAXIMUM CLIQUE problem the input is of the form (G, k) , where k is the size of the clique to be found. Usually we are considering problems that are polynomial-time solvable for every fixed value of k : for example, MAXIMUM CLIQUE can be solved by checking all the $O(n^k)$ size k sets. However, in parameterized complexity we are interested in the question whether there is a *uniformly polynomial-time* algorithm for the problem, that is, whether it can be solved in $f(k)n^c$ time, where f depends only on k and c is a constant independent of k . It turns out that several NP-hard problems, such as MINIMUM VERTEX COVER, LONGEST PATH, and TRIANGLE PACKING can be solved in uniformly polynomial time. Such problems are called *fixed-parameter tractable* (FPT). On the other hand, for MAXIMUM CLIQUE, MINIMUM DOMINATING SET, and several other problems, no uniformly polynomial algorithm is known. W[1]-hardness is the parameterized complexity analog of NP-hardness: by showing that a problem is W[1]-hard, we prove that the problem is not fixed-parameter tractable (under standard complexity-theoretic assumptions). For more background, the reader referred to the monograph of Downey and Fellows [8].

In [5] and [6], it is noted that an approximation algorithm with relative error $1/(2k)$ can decide whether the optimum is k . Therefore, an EPTAS with running time $f(\epsilon)n^c$ immediately implies that the parameterized version of the problem is fixed-parameter tractable: the EPTAS gives an $f(1/(2k))n^c$ time algorithm for the problem. This means that by proving that a problem is W[1]-hard, we can show that the problem is unlikely to have an EPTAS.

There are many optimization problems involving geometric objects in the plane that admit a PTAS. For example, [11] presents a PTAS for finding the maximum number of pairwise independent disks in a collection of unit disks. The reason for the abundance of approximation schemes for geometric problems is that in many cases, shifting and layering techniques can be used to reduce the problem to small subproblems that can be solved by brute force. In this paper we investigate whether it is possible to give an EPTAS for such problems.

For a set V of geometric objects, the *intersection graph* of V is a graph with vertex set V where two vertices are connected if and only if the two objects have non-empty intersection. Intersection graphs of disks, rectangles, line segments and other objects arise in applications such as facility location [14], frequency assignment [12], and map labeling [1]. Maximum independent set is NP-hard for the intersection graphs of unit disks, but admits an $n^{O(1/\epsilon)}$ time PTAS [11]. Alber and Fiala [2] considered the special case of λ -precision unit disk graphs, where the distance between the centers of disks have to be at least λ . For fixed λ , they gave a $O(2^{\sqrt{k}} + n)$ algorithm for finding an independent set of size k , which shows that the problem is FPT in this special case. The parameterized complexity of the problem without the precision restriction remained an open question. Here we answer this question by showing that maximum independent set is W[1]-complete for unit disk graphs. This result has two implications. First,

it shows that (unless $\text{FPT} = \text{W}[1]$) the uniformly polynomial algorithm of [2] for λ -precision unit disk graphs cannot be extended to general unit disk graphs. Furthermore, the PTAS of [11] cannot be improved to an EPTAS; the expression $1/\epsilon$ cannot be taken out of the exponent of n in the running time. Clearly, the same conclusion holds for the $n^{O(1/\epsilon^4)}$ PTAS of [9] that solves the more general problem of weighted independent set for disks with arbitrary diameter.

On the positive side, if we have a PTAS for a problem whose parameterized version is FPT, then this might indicate that the PTAS can be improved to an EPTAS. We present two examples for this situation. As shown in [2], the MAXIMUM INDEPENDENT SET problem is FPT for λ -precision unit disk graphs, and by [11], there is a linear-time EPTAS for such graphs. Furthermore, unlike MAXIMUM INDEPENDENT SET, the MINIMUM VERTEX COVER problem is FPT for every graph, hence it might be possible that there is an EPTAS for this problem on unit disk graphs. This is indeed so: in Section 3, we show that the PTAS of [11] can be improved to a linear-time EPTAS.

In Section 4, we study the problem of covering a given set of points by as few squares as possible. This problem is motivated by applications in image processing. Hochbaum and Maass [10] presented an $n^{O(1/\epsilon^2)}$ time PTAS for the problem. By proving that the corresponding parameterized problem is $\text{W}[1]$ -hard, we show that this PTAS cannot be improved to an EPTAS.

2 Maximum Independent Set

We prove that MAXIMUM INDEPENDENT SET remains $\text{W}[1]$ -complete when restricted to the intersection graphs of unit disks and unit squares. First we prove $\text{W}[1]$ -completeness for unit squares by a parameterized reduction from the MAXIMUM CLIQUE problem. A parameterized reduction transforms an instance (I, k) to an instance (I', k') where k' depends only on k , but not on I (ordinary reductions used to show NP-completeness usually do not have this property). As shown in Theorem 3, essentially the same reduction can be used for unit disks.

Theorem 1. *MAXIMUM INDEPENDENT SET is $\text{W}[1]$ -hard for the intersection graphs of axis-parallel unit squares in the plane.*

Proof. The reduction is from parameterized MAXIMUM CLIQUE. We have to determine whether the given graph G contains a clique of size k . For convenience, we assume that the number of vertices and the number of edges are both n .¹ The squares are open, two squares that share only a boundary do not intersect.

Set $\epsilon := 1/n^2$. The squares constructed in the reduction are partitioned into k' blocks, where k' depends only on k . Each block has a position (x, y) , which is a pair of integers. If a square belongs to the block at (x, y) , then the coordinates of its lower left corner are of the form $(x + i\epsilon, y + j\epsilon)$ for some integers $1 \leq i, j \leq n$. Therefore, a block can contain at most n^2 squares; a block containing all n^2 of them will be called a *complete block*. If the horizontal coordinate of a square is

¹ This can be achieved by adding/deleting isolated vertices and acyclic components.

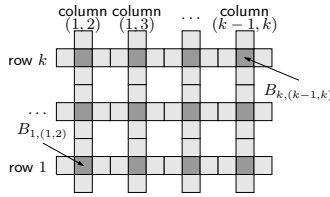


Fig. 1. The structure of the core

of the form $x + i\epsilon$, then we say that the *horizontal offset* of the square is i . The vertical offset is similarly defined. At most one square can be selected from each block, hence every independent set in G' has size at most k' . Furthermore, every size k' independent set contains exactly one square from each block.

The constructed instance consists of two parts: the core and the wrap-around machinery. The core is illustrated in Figure 1. There are k rows with $3\binom{k}{2}$ blocks in each, and there are $\binom{k}{2}$ columns with $3k$ blocks in each. The columns are indexed by the two element subsets of $\{1, 2, \dots, k\}$, i.e., each index is a pair (j_1, j_2) with $1 \leq j_1 < j_2 \leq k$. The block in the intersection of row i and column (j_1, j_2) is denoted by $B_{i,(j_1,j_2)}$. The set of squares contained in block $B_{i,(j_1,j_2)}$ is defined as follows. If $i \neq j_1$ and $i \neq j_2$, then the block $B_{i,(j_1,j_2)}$ is a complete block. Let e_1, e_2, \dots, e_n be the edges of G , and let $v^{(1)}(e_j) < v^{(2)}(e_j)$ be the two vertices of edge e_j . The block $B_{j_1,(j_1,j_2)}$ contains n squares: for each $1 \leq j \leq n$, it contains a square with horizontal offset $v^{(1)}(e_j)$ and vertical offset j . Similarly, $B_{j_2,(j_1,j_2)}$ contains a square with horizontal offset $v^{(2)}(e_j)$ and vertical offset j .

There are also blocks that are in one of the rows and in none of the columns, and there are blocks that are in one of the columns and in none of the rows. All these blocks (shown in lighter color in Fig. 1) are complete blocks.

Assume that one square is selected from each block of the core in such a way that they are pairwise non-intersecting. Such a solution will be called a *standard solution* if every square selected from row i has the same horizontal offset x_i and every square from column (j_1, j_2) has the same vertical offset $y_{(j_1,j_2)}$.

Lemma 2. *The core has a standard solution if and only if G has a size k clique.*

Proof. Assume that the core has a standard solution with horizontal offsets x_i and vertical offsets $y_{(j_1,j_2)}$. We claim that the numbers x_i correspond to a clique of size k in G . Suppose that, on the contrary, for some $1 \leq j_1 < j_2 \leq k$ the pair $x_{j_1}x_{j_2}$ is not an edge of G (including the possibility that $x_{j_1} = x_{j_2}$). Consider the square selected from block $B_{j_1,(j_1,j_2)}$, it has horizontal offset x_{j_1} and vertical offset $y' = y_{(j_1,j_2)}$. By construction, this means that $v^{(1)}(e_{y'}) = x_{j_1}$, i.e., the first vertex of edge $e_{y'}$ is x_{j_1} . Similarly, by considering the square selected from $B_{j_2,(j_1,j_2)}$, the second vertex of $e_{y'}$ is x_{j_2} , hence $x_{j_1}x_{j_2}$ is an edge.

Now assume that x_1, \dots, x_k is a clique of size k in G , with edge $e_{(j_1,j_2)}$ being the edge connecting x_{j_1} and x_{j_2} . Let the horizontal offset in row i be x_i , and let the vertical offset in column (j_1, j_2) be $e_{(j_1,j_2)}$. The construction ensures that we can select such a square from block $B_{i,(j_1,j_2)}$. Consider the four complete blocks next to $B_{i,(j_1,j_2)}$, and select a square from each of them that has the same offsets

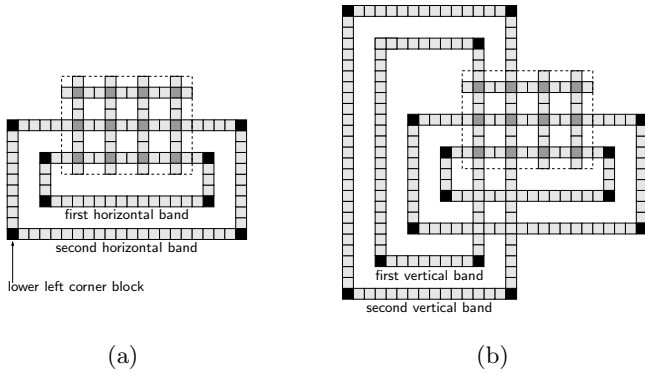


Fig. 2. (a) The first two horizontal wrap-around bands. (b) The first two horizontal and vertical wrap-around bands.

as the one selected from $B_{i,(j_1,j_2)}$. The square from $B_{i,(j_1,j_2)}$ and the four squares around it are pairwise non-intersecting, thus we get a (standard) solution. \square

The wrap-around machinery consists of k horizontal bands and $\binom{k}{2}$ vertical bands. The i -th horizontal band ensures that the selected squares in the i -th row have the same horizontal offset. Each horizontal band connects the first and the last block of the row, as shown in Figure 2a. Notice that the distance is at least two between two bands. Apart from the four corner blocks, each block is a complete block. The upper left and the lower right corner blocks have the same structure: for every $1 \leq i \leq n$, they contain the squares with horizontal offset i and vertical offset i . The upper right and the lower left corner blocks contain the squares with horizontal offset i and vertical offset $n - i + 1$ ($1 \leq i \leq n$).

We show that if one square is selected from each block such that they are independent, then the squares in row i have the same horizontal offset. Let ℓ_i (resp., r_i) be the horizontal offset of the first (resp., last) square of row i . The horizontal offset of the second square in row i is at least ℓ_i , otherwise it would conflict with the first square. Continuing this argument, we get that $\ell_i \leq r_i$, and the horizontal offset of every square in row i is between ℓ_i and r_i . Therefore, it is sufficient to show $r_i \leq \ell_i$. Consider the horizontal segment of the band extending from the last block of row i . By the same argument as above, the horizontal offset of each selected square is at least r_i . In particular, this is true for the upper right corner block. By construction, this means that the vertical offset of the square in that corner block is at most $n - r_i + 1$. This implies that in the right vertical segment of the band every block has a vertical offset of at most $n - r_i + 1$. Therefore, the horizontal offset of the square in the lower right corner block is at most $n - r_i + 1$. Continuing further in a similar way, it follows that the square in the lower left corner has vertical offset at least r_i , the square in the upper left corner has horizontal offset at least r_i , hence the horizontal offset of the first block of row i is also at least r_i . On the other hand, it is easy to see that if the square in the first and last block of row i have the same horizontal offset, then we can select one square from each block of the i -th horizontal band.

The $\binom{k}{2}$ vertical bands are defined analogously, band (j_1, j_2) ensures that the vertical offset is the same for every square selected from column (j_1, j_2) (see Figure 2b.) Notice that we reuse some blocks of the horizontal bands when a vertical band crosses a horizontal band, but that does not modify our conclusion that the bands enforce a standard solution for the core. The only thing that has to be carefully examined is whether a standard solution of the core can be extended to the horizontal and vertical bands simultaneously. Consider the block B at the intersection of horizontal band i and vertical band (j_1, j_2) . The horizontal (resp., vertical) band determines the horizontal (resp., vertical) offset of the square in this block. There are 4 other blocks next to block B , from each of them we select a square with the same horizontal and vertical offset as the one selected from B . The selected squares do not intersect each other, and they do not intersect the other squares on their band. This can be done independently for every intersection, since their distance is at least two blocks.

The reduction constructs k' blocks, where k' depends only on k . We have shown that there are k' independent squares if and only if there is a standard solution for the core, or equivalently, if the graph G has a clique of size k . \square

The same reduction shows hardness for unit disks. For convenience, we present the proof for unit-diameter disks instead of unit-radius disks.

Theorem 3. MAXIMUM INDEPENDENT SET is $W[1]$ -hard for the intersection graphs of unit disks in the plane.

Proof. The same reduction works as in Theorem 1: a square with lower left corner at $(x + i\epsilon, y + j\epsilon)$ can be replaced by a unit disk with center $(x + i\epsilon, y + j\epsilon)$. In the reduction, only the following two properties of the squares were used:

1. If we select a square with horizontal offset i_1 from the block at position (x, y) , and we select a square with horizontal offset i_2 from the block at position $(x + 1, y)$, then they intersect if and only if $i_1 > i_2$.
2. If we select a square with vertical offset j_1 from the block at position (x, y) , and we select a square with vertical offset j_2 from the block at position $(x, y + 1)$, then they intersect if and only if $j_1 > j_2$.

We show that the same properties hold for (open) unit disks. Consider two disks with centers $(x + i_1\epsilon, y + j_1\epsilon)$ and $(x + 1 + i_2\epsilon, y + j_2\epsilon)$. Clearly, if $i_1 \leq i_2$, then their distance is at least 1. On the other hand, if $i_1 > i_2$ then their distance is

$$\begin{aligned} & \sqrt{(1 + (i_2 - i_1)\epsilon)^2 + (j_2 - j_1)^2\epsilon^2} \\ & \leq \sqrt{(1 - \epsilon)^2 + n^2\epsilon^2} = \sqrt{1 - 2\epsilon + (n^2 + 1)\epsilon^2} < 1, \end{aligned}$$

if $\epsilon \leq 1/n^2$. Property 2 can be shown similarly. \square

The reduction constructs instances where the centers are arbitrarily close to each other. Therefore, the proof does not work for λ -precision unit disk graphs. This is not surprising, since it is shown in [2] that for every fixed $\lambda > 0$, MAXIMUM INDEPENDENT SET is fixed-parameter tractable for λ -precision unit disk

graphs. This opens the possibility that (unlike the general case) the problem restricted to λ -precision unit disk graphs admits an EPTAS. Indeed, [11] shows that there is a linear-time EPTAS for every fixed λ .

3 Minimum Vertex Cover

After giving a PTAS for MAXIMUM INDEPENDENT SET in unit disk graphs, Hunt et al. [11] briefly discuss a similar PTAS for the MINIMUM VERTEX COVER problem. From the parameterized complexity point of view, MAXIMUM INDEPENDENT SET and MINIMUM VERTEX COVER are very different: the first problem is W[1]-hard (even for unit disk graphs, Section 2), while the latter problem is fixed-parameter tractable for *every* graph (the current best algorithm is presented in [13]). Therefore, we cannot prove a MINIMUM VERTEX COVER analog of Theorem 3, which raises the possibility that the PTAS of [11] for MINIMUM VERTEX COVER can be improved to an EPTAS. We show here that some simple ideas are sufficient to turn this PTAS into a linear-time algorithm.

Let D be a set of unit-diameter disks. In the first phase of the algorithm, we ensure that every point of the plane is contained in at most $1/\epsilon$ disks. If point p is contained in more than $1/\epsilon$ disks, then add these disks into the set S , and remove them from D . We repeat this until no such p can be found, let D_0 be the set of remaining disks. We claim that the set S together with a $(1 + \epsilon)$ -approximation of the vertex cover for D_0 gives a $(1 + \epsilon)$ -approximate vertex cover for D . This follows from the fact that whenever we add to S the $\ell \geq 1/\epsilon + 1$ disks containing some point p , then at least $\ell - 1$ of these disks have to appear in every vertex cover. Hence S itself is a $(1 + \epsilon)$ -approximate vertex cover of S .

The linear-time EPTAS of [11] for MAXIMUM INDEPENDENT SET in λ -precision unit disk graphs is based on the observation that a constant-sized rectangle can contain at most a constant number of disks. If we perform the first phase of the algorithm, then this property will hold for our instance. Therefore, we can give a linear-time PTAS for MINIMUM VERTEX COVER that is similar to [11–Theorem 5.2] (we omit the details).

Theorem 4. *There is a $2^{O(1/\epsilon^2)} \cdot n$ time EPTAS for MINIMUM VERTEX COVER in unit disk graphs.* \square

4 Covering Points with Squares

Hochbaum and Maass [10] presented a PTAS for the problem of covering n given points in \mathbb{R}^d with the minimum number of d -dimensional unit-diameter balls or d -dimensional rectilinear blocks. As a special case, their result gives an $n^{O(1/\epsilon^2)}$ time approximation scheme for covering n points in the plane by unit squares. We show that it is unlikely that this PTAS can be improved to an EPTAS, since the parameterized version of the problem is W[1]-hard.

Theorem 5. COVERING POINTS WITH SQUARES *is* W[1]-hard.

Proof. The proof is similar in structure to the proof of Theorem 1. The reduction is from MAXIMUM CLIQUE; the k vertices are selected by k rows, and the $\binom{k}{2}$ edges are selected by $\binom{k}{2}$ columns. Horizontal and vertical bands are used to ensure the consistency of the rows and columns (see below for details). In Theorem 1, the structure of the graph was encoded by the squares available in certain blocks. In COVERING POINTS WITH SQUARES, we cannot prescribe which squares can be used in a solution, hence a more delicate construction is required to ensure that the selected vertices and edges form a correct solution.

We will use the directions east (increasing x coordinate), north (increasing y coordinate), northeast, etc. The directions will be abbreviated as E, N, NE, etc. For convenience, we assume that the squares are closed on west and south, and open on east and on north. The SW- and the SE-corner points belong to the square, but the NW- and the NE-corners do not. (It can be shown that the problem has the same complexity with open, closed, and half-open squares).

Let G be the graph where we have to find a size k clique. For convenience, we assume that the number of edges and vertices are both n in G . Set $\epsilon := 1/n^2$. Every point constructed in the reduction has coordinates that are integer multiples of ϵ . Therefore, it can be assumed that in a solution the coordinates of the corners of each square are integer multiples of ϵ .

We use the points to construct *blocks*, *connectors*, and *testers*. If there is a *block* at (x, y) , then this means that we add 5 points $(x+0.5, y+0.5)$, $(x+n\epsilon, y+0.5)$, $(x+1-n\epsilon-\epsilon, y+0.5)$, $(x+0.5, y+n\epsilon)$, $(x+0.5, y+1-n\epsilon-\epsilon)$. These 5 points are called the central, W, E, S, N *control points* of the block, respectively.

The problem parameter k' in the constructed instance of COVERING POINTS WITH SQUARES is equal to the number blocks. It can be shown (details omitted) that the only way k' squares can cover the control points of k' blocks is that if every square corresponds to some block:

Proposition 6. *For each block, there is a unique square in the solution that covers all five control points of the block.* □

Therefore, the SW-corner of the square of block (x, y) has coordinates $(x+i\epsilon, y+j\epsilon)$ for some integers $-n \leq i \leq n$, $-n \leq j \leq n$. These two integers are called the *horizontal and vertical offsets* of block (x, y) .

We will add *boundary points* to some of the blocks. There are four types of boundary points: N, S, E, W. Whenever we add a N (S etc.) boundary point to a block, then it will be true that there is no N (S etc.) neighbor of the block. Adding boundary points to the block at (x, y) will be done as follows:

- The N-boundary point is at $(x+0.5, y+1)$. It ensures that the vertical offset of the block is positive (recall that the squares are open on north).
- The S-boundary point is at $(x+0.5, y)$. It ensures that the vertical offset of the block is not positive.

The E-boundary (resp., W-boundary) points are defined analogously, they ensure that the horizontal offset is positive (resp., not positive).

A *connector* is a set of points whose job is to ensure that certain relations hold between the offsets of two neighboring blocks. A horizontal connector between

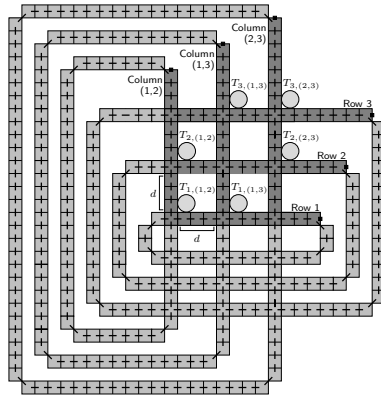


Fig. 3. Structure of the constructed instance in the proof of Theorem 5

blocks (x, y) and $(x + 1, y)$ consists of the $2n$ points $(x + 1 + i\epsilon, y + 0.5)$ ($-n \leq i \leq n - 1$). These points can be covered only by the squares of blocks (x, y) and $(x + 1, y)$. These two squares cover the connector if and only if the horizontal offset of block (x, y) is not smaller than the horizontal offset of block $(x + 1, y)$.

Similarly, the vertical connector between blocks (x, y) and $(x, y + 1)$ consists of the points $(x + 0.5, y + 1 + i\epsilon)$ ($-n \leq i \leq n - 1$). These points ensure that the vertical offset of (x, y) is not the smaller than the vertical offset of $(x, y + 1)$.

A diagonal connector between blocks (x, y) and $(x + 1, y + 1)$ consists of the points $(x + 1 + i\epsilon, y + 1 + i\epsilon)$ ($-n \leq i \leq n - 1$). These points ensure that if block (x, y) has offsets i_1, j_1 , and block $(x + 1, y + 1)$ has offsets i_2, j_2 , then $i_2, j_2 \leq \min(i_1, j_1)$. The blocks $(x, y + 1)$ and $(x + 1, y)$ can be connected in a similar way. In this case, if i_1, j_1 are the offsets of $(x, y + 1)$, and i_2, j_2 are the offsets of $(x + 1, y)$, then $i_2 \leq \min(i_1, -j_1)$ and $j_2 \geq \max(-i_1, j_1)$ follows.

Figure 3 shows the structure of the constructed instance of COVERING POINTS WITH SQUARES. As in Theorem 1, there are k rows (shown by darker blocks) that correspond to the k vertices of the clique, and there are $\binom{k}{2}$ columns (also shown in dark) that correspond to the $\binom{k}{2}$ edges of the clique. The rows are indexed from 1 to k , while the column indexes are pairs (j_1, j_2) ($1 \leq j_1 < j_2 \leq k$). The connector gadgets connecting the neighboring blocks are shown by short line segments in the figure. There are $2\binom{k}{2}$ tester gadgets (shown by circles in Figure 3): for every (j_1, j_2) ($1 \leq j_1 \leq j_2 \leq k$), there is a tester gadget $T_{j_1, (j_1, j_2)}$ connected to both row j_1 and column (j_1, j_2) , and there is a tester gadget $T_{j_2, (j_1, j_2)}$ connected to both row j_2 and column (j_1, j_2) . The distance between the rows/columns is d blocks, where d is sufficiently large to ensure that there is enough space for the tester gadgets between the rows and columns (e.g., $d = 20$ is enough).

For each row, the leftmost and the rightmost blocks are connected by a horizontal band. As in the proof of Theorem 1, this band ensures that in every solution, the horizontal offset is the same for every block of the row. If we follow what requirements the connectors prescribe on the adjacent blocks, then it turns

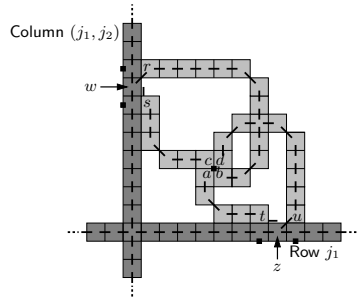


Fig. 4. The tester gadget

out that the horizontal offset of the rightmost block cannot be smaller or larger than the horizontal offset of the leftmost block. Therefore, the same horizontal offset has to appear in every block of the row. Similarly, the vertical bands ensure that the vertical offset is the same for every block of a given column.

We add an E-boundary point to the rightmost block of each row (see the small dot in Fig. 3). Similarly, we add a N-boundary point to the topmost block of each column. These points ensure that the horizontal (resp., vertical) offset is between 1 and n for every row (resp., column).

Given a solution to the constructed instance of COVERING POINTS WITH SQUARES, we interpret the horizontal offset of row i as the index of the i -th vertex of the clique, and the vertical offset of column (j_1, j_2) as the index of the edge e_{j_1, j_2} connecting the j_1 -th and j_2 -th vertices. The tester gadgets ensure that this interpretation gives a consistent solution: the two end points of the edge given by column (j_1, j_2) are the vertices given by rows j_1 and j_2 . More precisely, gadget $T_{j_1, (j_1, j_2)}$ ensures that the smaller end of e_{j_1, j_2} is the j_1 -th vertex of the clique, and $T_{j_2, (j_1, j_2)}$ ensures that the larger end of e_{j_1, j_2} is the j_2 -th vertex.

The tester gadget $T_{j_1, (j_1, j_2)}$ connected to row j_1 and column (j_1, j_2) is constructed as follows (the description of $T_{j_2, (j_1, j_2)}$ is similar). The gadget consists of 33 blocks and it is arranged as shown in Fig. 4. Besides these new blocks, we add some additional points to the rows and columns as well. A W-boundary point is added to the W-neighbor of block r and to the W-neighbor of block s (see Fig. 4). A S-boundary point is added to the S-neighbors of blocks t and u .

Let (x, y) be the coordinates of the SE-corner of block t . Let us add the points $(x + \ell\epsilon, y + \epsilon)$ ($1 \leq \ell \leq n$). The S-neighbor of block t has a S-boundary point, thus only block t and its SE-neighbor, z , can cover these new points. This means that the horizontal offset of block t cannot be smaller than the horizontal offset of row j_1 . On the other hand, if the horizontal offset of row j_1 is α , then z can cover $(x + \ell\epsilon, y + \epsilon)$ for every $\ell \geq \alpha$ if we set the vertical offset of z to at least 2. In this case, t has to cover $(x + \ell\epsilon, y + \epsilon)$ only for $\ell < \alpha$, hence if the horizontal offset of t is not smaller than α , then all the points are covered.

In a similar way, we force the vertical offset of block s to be at least as large as the vertical offset of column (j_1, j_2) . For this purpose, if (x, y) are the coordinates of the NW-corner of block s , then we add the points $(x + \epsilon, y + \ell\epsilon)$ ($1 \leq \ell \leq n$). The argument is the same as in the previous paragraph.

There is a diagonal connector between blocks z and u . However, this connector is slightly different from the one defined above. Let (x, y) be the coordinates of the NE-corner of block z . The connector consists of the points $(x + (\ell + 1)\epsilon, y + \ell\epsilon)$ for $-n \leq \ell \leq n$. Notice that these points cannot be covered by the S-neighbor of u , since that block has a S-boundary point and its horizontal offset is positive. A normal connector between z and u would ensure that the vertical offset of u is at most as large as the horizontal offset of z . This modified connector forces a stronger requirement: it ensures that the vertical offset of u is at most the horizontal offset of z minus 1. A similar connector forces the horizontal offset of r to be at most the vertical offset of w minus 1.

The construction described so far depends only on k and n , but not on the structure of the graph G . The *tester points* in each tester gadget encode the edges of G . Let (x, y) be the coordinates of the common corner of blocks a, b, c, d in gadget $T_{j_1, (j_1, j_2)}$ connected to row j_1 and column (j_1, j_2) . If the p -th vertex is *not* the smaller endpoint of the q -th edge ($1 \leq p, q \leq n$), then we add the point $(x - q\epsilon, y - p\epsilon)$. The gadget $T_{j_2, (j_1, j_2)}$ is similarly defined, but in this case a tester point $(x - q\epsilon, y - p\epsilon)$ signifies that the p -th vertex is not the larger end point of the q -th edge. This completes the description of the reduction.

To prove the correctness of the reduction, first we show that if there is a solution for COVERING POINTS WITH SQUARES, then there is a size k clique in G . Recall that the problem parameter (the maximum number of allowed squares) in the constructed instance of COVERING POINTS WITH SQUARES equals the number of blocks. This means that the squares in the solution correspond to the blocks. As we have seen above, each row has a horizontal offset between 1 and n , let v_i be the vertex indexed by the horizontal offset of row i . Similarly, let e_{j_1, j_2} be the edge indexed by the vertical offset of column (j_1, j_2) .

We claim that the v_i 's form a clique in G , with e_{j_1, j_2} being the edge connecting v_{j_1} and v_{j_2} . Suppose that this claim does not hold for some j_1 and j_2 . Assume without loss of generality that e_{j_1, j_2} is not incident to v_{j_1} . Let us look at what happens in tester gadget $T_{j_1, (j_1, j_2)}$. Let p be the index of v_{j_1} and q be the index of e_{j_1, j_2} . The horizontal offset of row j_1 is p , hence the horizontal offset of t is at least p , and the vertical offset of u is at most $p - 1$. If we follow the implications of this, then it turns out that the vertical offset of a is at most $-p$, and the vertical offset of d is at least $-p + 1$. Similarly, the horizontal offset of c is forced to be at most $-q$, and the horizontal offset of b is forced to be at least $-q + 1$. Let (x, y) be the coordinates of the common corner of a, b, c , and d . From the assumption that the q -th edge is not incident to the p -th vertex, it follows that there is a tester point at $(x - q\epsilon, y - p\epsilon)$. This point is not covered by any of the blocks a, b, c, d : for example, block a cannot cover a point with vertical coordinate at least $y - p\epsilon$; block b cannot cover a point with horizontal coordinate less than $x - q\epsilon + \epsilon$, etc. This gives a contradiction.

To prove the other direction, we have to show that if there is a size k clique in G , then there is a solution for the constructed instance of COVERING POINTS WITH SQUARES. Let v_i be the i -th vertex in the clique, and let e_{j_1, j_2} be the edge connecting v_{j_1} and v_{j_2} . It is clear that we can cover the points in the k rows, $\binom{k}{2}$

columns, horizontal bands, and vertical bands in such a way that the horizontal offset of row i is the index of v_i , and the vertical offset of column (j_1, j_2) is the index of e_{j_1, j_2} . The only thing that should be verified is whether the points in the tester points in the tester gadgets can be covered. Consider the tester gadget that is connected to row j_1 and column (j_1, j_2) . Let p be the index of v_{j_1} , and let q be the index of e_{j_1, j_2} . Each tester point has coordinates $(x - \alpha\epsilon, y - \beta\epsilon)$ for some α and β . We use block c to cover all the tester points with $\alpha < q$; we use block b to cover the tester points with $\alpha \geq q + 1$, etc. By construction, there is no tester point at $(x - q\epsilon, y - p\epsilon)$, hence all the points are covered. \square

References

1. P. K. Agarwal, M. van Kreveld, and S. Suri. Label placement by maximum independent set in rectangles. *Comput. Geom.*, 11(3-4):209–218, 1998.
2. J. Alber and J. Fiala. Geometric separation and exact solutions for the parameterized independent set problem on disk graphs. *J. Algorithms*, 52(2):134–151, 2004.
3. S. Arora. Polynomial time approximation schemes for Euclidean TSP and other geometric problems. In *FOCS 1996*, pages 2–11. IEEE Comput. Soc. Press, 1996.
4. S. Arora. Polynomial time approximation schemes for Euclidean traveling salesman and other geometric problems. *J. ACM*, 45(5):753–782, 1998.
5. C. Bazgan. Schémas d'approximation et complexité paramétrée. Technical report, Université Paris Sud, 1995.
6. M. Cesati and L. Trevisan. On the efficiency of polynomial time approximation schemes. *Inform. Process. Lett.*, 64(4):165–171, 1997.
7. R. G. Downey. Parameterized complexity for the skeptic. In *Proceedings of the 18th IEEE Annual Conference on Computational Complexity*, pages 147–169, 2003.
8. R. G. Downey and M. R. Fellows. *Parameterized complexity*. Monographs in Computer Science. Springer-Verlag, New York, 1999.
9. T. Erlebach, K. Jansen, and E. Seidel. Polynomial-time approximation schemes for geometric graphs. In *SODA 2001*, pages 671–679. SIAM, 2001.
10. D. S. Hochbaum and W. Maass. Approximation schemes for covering and packing problems in image processing and VLSI. *J. ACM*, 32(1):130–136, 1985.
11. H. B. Hunt, III, M. V. Marathe, V. Radhakrishnan, S. S. Ravi, D. J. Rosenkrantz, and R. E. Stearns. NC-approximation schemes for NP- and PSPACE-hard problems for geometric graphs. *J. Algorithms*, 26(2):238–274, 1998.
12. E. Malesińska. *Graph-Theoretical Models for Frequency Assignment Problems*. PhD thesis, Technical University of Berlin, 1997.
13. L. Sunil Chandran and F. Grandoni. Refined memorization for vertex cover. *Inform. Process. Lett.*, 93(3):125–131, 2005.
14. D. W. Wang and Y.-S. Kuo. A study on two geometric location problems. *Inform. Process. Lett.*, 28(6):281–286, 1988.

Geometric Clustering to Minimize the Sum of Cluster Sizes

Vittorio Bilò¹, Ioannis Caragiannis², Christos Kaklamanis²,
and Panagiotis Kanellopoulos²

¹ Dipartimento di Matematica “Ennio De Giorgi”
Università di Lecce, Provinciale Lecce-Arnesano, 73100 Lecce, Italy

² Research Academic Computer Technology Institute &
Department of Computer Engineering and Informatics,
University of Patras, 26500 Rio, Greece

Abstract. We study geometric versions of the *min-size k-clustering* problem, a clustering problem which generalizes clustering to minimize the sum of cluster radii and has important applications. We prove that the problem can be solved in polynomial time when the points to be clustered are located on a line. For Euclidean spaces of higher dimensions, we show that the problem is NP-hard and present polynomial time approximation schemes. The latter result yields an improved approximation algorithm for the related problem of *k-clustering* to minimize the sum of cluster diameters.

1 Introduction

Clustering is an area of combinatorial problems which is both algorithmically rich and practically relevant. Several clustering problems have been extensively studied since they have applications in many fields including database systems, image processing, data mining, information retrieval, molecular biology, and more.

Given a set of points X , we call a *cluster* any nonempty subset of X . A set of clusters is a *clustering* for X if each point of X belongs to some cluster. A clustering is called *k-clustering* if it consists of at most k clusters. In general, clustering problems are stated as follows: An instance of such a problem consists of a set X of n points, a distance function $\text{dist} : X \times X \rightarrow R$ and an integer k and the objective is to compute a *k-clustering* of the points in X minimizing $f(C_1, \dots, C_k)$, where f is a function defined on the clusters, typically using the distance function dist . Depending on the definition of the function f , many different clustering problems can be defined. The mostly studied ones are the *k-center*, *k-median*, and *k-clustering*. Their objectives are to assign the points to at most k clusters so that the maximum distance from any point to its cluster center (*k-center*) or the sum of distances from each point to its closest cluster center (*k-median*) or the sum of all distances between points in the same cluster (*k-clustering*) is minimized. These problems are NP-hard and several approximation algorithms have been proposed [3,5,13] including polynomial time approximation schemes for geometric instances of these problems [1,2,10,16].

In this paper, we study a variation of the problem of clustering a set of points into a specific number of clusters so as to minimize the sum of cluster sizes. The size of a cluster may be proportional to the radius/diameter of the cluster, to its area, etc. In particular, minimizing the sum of cluster radii/diameters has been suggested as an alternative to the k -center objective in certain applications so as to avoid the *dissection effect* [8]: using the maximum diameter/radius as the objective sometimes results in objects that should have been placed in the same cluster to be placed in different clusters.

Clustering to minimize the sum of diameters/radii has been studied for points in metric spaces in [6] and [8]. An approximation algorithm which computes a solution with at most $10k$ clusters of cost at most a factor of $O(\log n/k)$ within the optimal solution for k clusters was presented in [8]. This result was improved by Charikar and Panigrahy in [6] where an algorithm that computes a constant approximate solution using at most k clusters is presented. In metric spaces, ρ -approximation algorithms for clustering to minimize the sum of diameters give 2ρ -approximation algorithms for the corresponding radii problem (and vice versa). Negative results include a $2 - \epsilon$ inapproximability bound for minimizing the sum of diameters in metric spaces [8] while the complexity of the corresponding radii problem is open. For non-metrics, no approximation bound is possible for diameters in polynomial time unless $P = NP$ even for $k = 3$ [8]. When k is fixed, the optimal solution for radii/diameters can be found in polynomial time by enumerating the $O(n^k)$ possible solutions. The papers [12] and [15] present fast polynomial time algorithms for the case $k = 2$, addressing the Euclidean case as well. Capote et al. [7] study a generalized version of the problem for points on the Euclidean plane and show that, for fixed k and any function of the cluster diameters, it can be solved in polynomial time.

In this paper, we consider geometric versions of the *min-size k -clustering* problem. Formally, an instance (X, F, d, α) of the problem has a set X of n points with rational coordinates on the d -dimensional Euclidean space, a cost function F that associates a fixed non-negative cost with each point, and a constant value α . The objective is to compute a k -clustering \mathcal{C} together with center points $c \in X$ in each cluster C such that $\sum_{C \in \mathcal{C}} COST(C)$ is minimized, where $COST(C)$ is defined as $(\max_{p \in C} \text{dist}(p, c))^\alpha + F_c$ and $\text{dist}(p, c)$ denotes the Euclidean distance between the points p and c . The quantity $\max_{p \in C} \text{dist}(p, c)$ is the *radius* of cluster C with center c .

Besides its importance for clustering optimization, another motivation for studying the min-size k -clustering problem is the following scenario. Assume that a telecommunication agency wishes to give wireless access to users scattered in several locations. This can be achieved by establishing a network of base stations (antennas) to specific locations and setting appropriately the range of each base station such that all the locations are within the range of some station. From the point of view of the agency, establishing a base station incurs a setup cost and an operational cost which is proportional to the range of the station (i.e., the square of the distance of the farthest location within range from the base station). Min-size k -clustering models the problem of minimizing the costs for

building and operating the network. Very recently, we became aware of [14] which studied special cases of min-size k -clustering under this motivation. The authors of [14] study instances $(X, F, d, 1)$ of min-size k -clustering with $k = n$ and fixed costs in $\{0, \infty\}$. They present a dynamic programming algorithm that solves the problem optimally when the points are located on the line and a polynomial-time approximation scheme for points in Euclidean spaces of constant dimensions. This latter result is based on ideas of a dynamic programming algorithm of [9] for approximating the minimum vertex cover of disk graphs.

Min-size k -clustering generalizes the problem of minimizing the sum of radii. We consider the case where k is arbitrary. The result of [6] for metric spaces implies an algorithm with approximation ratio slightly worse than 3^α in our case. We show that the problem is NP-complete in 2-dimensional Euclidean spaces and $\alpha \geq 2$, while a generalized version is solvable in polynomial-time when the points are located on a line. For higher dimensions, we present a polynomial time approximation scheme that computes an $(1 + \epsilon)$ -approximate solution using at most k clusters; the running time of our algorithm is $n^{(\alpha/\epsilon)^{O(d)}}$. Our techniques yield a $(2 + \epsilon)$ -approximation algorithm for the k -clustering to minimize the sum of cluster diameters. Like [14], our algorithm uses and extends ideas from [9]. Our results are stronger than those in [14] since we assume that k can be arbitrary, that the fixed costs of the points may have arbitrary positive values, and we consider the more general case $\alpha \geq 1$. Our algorithm is guaranteed to find approximate solutions in polynomial time due to structural properties of the optimal or approximate solutions. This is captured by corresponding *Structure Lemmas*.

The rest of the paper is structured as follows. In Section 2 we give complexity results for the problem. We present the algorithm and its analysis in Section 3. Section 4 contains the statements and proofs of the Structure Lemmas. We conclude with some extensions and open problems in Section 5. Due to lack of space, most of the proofs have been omitted.

2 Complexity Results

We first show that the problem is solvable in polynomial time when the points are located on the line.

Theorem 1. *Min-size k -clustering for instances $(X, F, 1, \alpha)$ is in P.*

The proof of this statement follows by expressing the problem as an integer linear program with totally unimodular matrix and concluding that an optimal clustering is obtained by computing a basic solution for the linear program. The statement also holds if the clusters have arbitrary positive costs. Previous results include weaker statements with more complicated proofs [4,14].

In the sequel we consider points in higher dimensions. We can show that two important cases of the problem on the Euclidean plane are NP-hard. The first case is an interesting geometric version of set cover which is also studied in [11]. We have two disjoint sets of points S and T on the Euclidean plane.

We wish to cover all points in T by disks centered in points of S so that the total area of the disks is minimized. It is not difficult to see that this problem is equivalent to the min-size k -clustering with $k = |S \cup T| = n$ and input instance $(S \cup T, F, 2, 2)$ where $F_p = \infty$ if $c \in T$ (this guarantees that points of T should not be cluster centers) and $F_p = 0$ if $c \in S$ (this guarantees that all points of S can be centers of clusters including no points of T). In the instances of the second case that we prove to be NP-hard, all points have zero fixed costs. Our NP-hardness statements follow.

Theorem 2. *Let $(X, F, 2, \alpha)$ be an instance of the problem with $\alpha \geq 2$ and F such that $F_p \in \{0, \infty\}$ for any point $p \in X$. Deciding whether $(X, F, 2, \alpha)$ has any min-size clustering of cost at most K is NP-complete.*

Theorem 3. *Let $(X, F, 2, \alpha)$ be an instance of the problem with $\alpha \geq 2$ and $F_p = 0$ for any point $p \in X$. Deciding whether $(X, F, 2, \alpha)$ has any min-size k -clustering of cost at most K is NP-complete.*

3 The Algorithm

Our algorithm uses the idea of plane subdivision from an algorithm of Erlebach et al. [9] that approximates the minimum vertex cover of disk graphs. Given disks on the plane, the corresponding disk graph is the graph having a node for each disk and an edge between any pair of nodes corresponding to overlapping disks. Although it is not at all related to minimum vertex cover in disk graphs, the min-size k -clustering can be seen as a covering problem with disks as well. We may think of a cluster C with center c as a disk centered at the point c and with radius equal to the maximum distance of c from any point of C (and possibly zero if c is the only point of C). Such a disk has a cost equal to the quantity $(\max_{p \in C} \text{dist}(p, c))^\alpha + F_c$. Now, the min-size k -clustering problem asks for a set of at most k disks with minimum total cost which include (i.e., cover) all points of X .

Before we describe the min-size k -clustering algorithm, we adapt the terminology of [9] to our setting. We use the term cluster instead of the term disk. Fix a positive integer $\lambda > 1$. Consider an instance $(X, F, 2, \alpha)$ of min-size k -clustering and let \mathcal{D} denote the set of all possible n^2 clusters obtained by considering all possible radii for each point in X . Among all clusters of \mathcal{D} with non-zero radius, let r_{min} and r_{max} be the radius of the smallest and the largest cluster, respectively. Partition \mathcal{D} into $L + 1$ levels, where $L = \lfloor \log_{\lambda+1}(r_{max}/r_{min}) \rfloor$. For $0 \leq j \leq L$, level j consists of all clusters d_i having radius r_i such that $(\lambda + 1)^{-j}r_{max} \geq r_i > (\lambda + 1)^{-(j+1)}r_{max}$. Note that the smaller the level, the larger the radii of the clusters are. Thus, the cluster with radius r_{min} will be on level L . We assume that clusters with zero radius belong to level L as well.

For each level j , $0 \leq j \leq L$, impose a grid on the plane consisting of lines that are $2(\lambda + 1)^{-j}r_{max}$ apart from each other. The v -th vertical line, for integer v in $(-\infty, \infty)$, is at $x = 2v(\lambda + 1)^{-j}r_{max}$. The h -th horizontal line, for integer h in $(-\infty, \infty)$, is at $y = 2h(\lambda + 1)^{-j}r_{max}$. We say that the v -th vertical line

has index v and that the h -th horizontal line has index h . Furthermore, we say that a cluster d_i with center (x_i, y_i) and radius r_i hits a vertical line at $x = a$ if $a - r_i < x_i \leq a + r_i$. Similarly, we say that d_i hits a horizontal line at $y = b$ if $b - r_i < y_i \leq b + r_i$. Intuitively, by considering clusters as disks, a cluster hits a line if it intersects that line, except if it only touches the line from the left or from below. Note that every cluster can hit at most one horizontal line and at most one vertical line on its level.

Let $0 \leq r, s < \lambda$ and consider the vertical lines whose index modulo λ equals r and the horizontal lines whose index modulo λ equals s . We say that these lines are *active* for (r, s) . Consider one particular level j . The lines on level j that are active for (r, s) partition the plane into squares. More precisely, for consecutive active vertical lines at $x = a_1$ and $x = a_2$ and consecutive active horizontal lines at $y = b_1$ and $y = b_2$, one square $\{(x, y) | a_1 < x \leq a_2, b_1 < y \leq b_2\}$ is obtained. We refer to these squares on level j as *j-squares*. As observed in [9], for any j , $0 \leq j < L$, every $(j + 1)$ -square is completely contained in some j -square. An example is depicted in Figure 1.

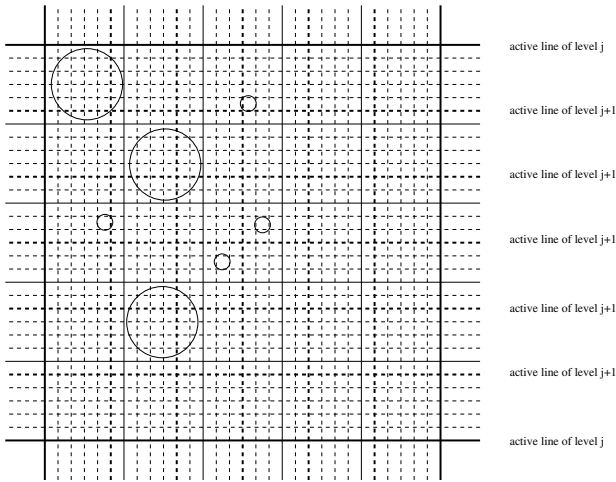


Fig. 1. An example of the plane subdivision for $\lambda = 5$. The disks shown represent clusters of level j of the minimum and maximum possible radius.

A j -square S is relevant if there exists at least one cluster of level j in \mathcal{D} containing a point $p \in S \cap X$. Observe that the number of relevant squares is polynomial in n , since the number of clusters is n^2 and a cluster may cover points in at most 4 squares of its level. For a relevant j -square S and a relevant j' -square S' with $j' > j$, we say that S' is a *child square* of S (and S is a *parent* of S') if S' is contained in S and there is no relevant j'' -square S'' with $j' > j'' > j$, such that S' is contained in S'' and S'' is contained in S . It can be easily seen that the number of relevant 0-squares is at most 4; these are the only squares

without a parent. We show the following property which holds specifically for instances of min-size k -clustering.

Lemma 1. *Each relevant square has at most $O(\lambda^4)$ child squares.*

Proof. Clearly, a square S of level j and of side length ℓ may have at most $(\lambda+1)^4$ child squares of levels $j+1$ and $j+2$. If S has more than $(\lambda+1)^4$ child squares, then it should have child squares of level at least $j+3$. We will show that the number of child squares of S of level at least $j+3$ is at most $\frac{16}{\pi}(2(\lambda+1)^2+1)^2$.

Pick a square S' of smallest level $j' \geq j+3$ among the child squares of S and let p be a point contained in it. Then, all other points will be at distance either smaller than $\frac{\ell}{2(\lambda+1)^{j'-j+1}}$ or at least $\frac{\ell}{2(\lambda+1)^2}$, otherwise the j'' -square S'' containing S' with $j < j'' < j'$ would be relevant and, hence, S'' , instead of S' , would be a child of S . Now, observe that, within a disk of radius $\frac{\ell}{4(\lambda+1)^2}$ centered at p , there can be at most four child squares of S of level at least $j+3$, including S' ; this is the maximum number of squares that may have one point at distance smaller than $\frac{\ell}{2(\lambda+1)^{j'-j+1}}$ from p . Repeat recursively this procedure for the child squares of S of level at least $j+3$ which are not contained in the disk until all squares of level at least $j+3$ have been included in disks. The disks do not overlap, otherwise this would mean that the center of some disk which is a point in a square of level $j+3$ has distance smaller than $\frac{\ell}{2(\lambda+1)^2}$ and at least $\frac{\ell}{2(\lambda+1)^{j''-j+1}}$ from some other point. Also, they all have their centers in S , thus they are all contained in the square of side length $\left(1 + \frac{1}{2(\lambda+1)^2}\right)\ell$. Hence, their number is at most

$$\frac{\left(1 + \frac{1}{2(\lambda+1)^2}\right)^2 \ell^2}{\pi \left(\frac{\ell}{4(\lambda+1)^2}\right)^2} \leq \frac{4}{\pi}(2(\lambda+1)^2+1)^2,$$

and the number of child squares of S of level at least $j+3$ cannot exceed $\frac{16}{\pi}(2(\lambda+1)^2+1)^2$. □

Consider some j -square S and denote by I^S the set of clusters in \mathcal{D} intersecting S . We denote by $I^S_{<j}$ the set of clusters in I^S having level smaller than j and define $I^S_{\leq j}$, $I^S_{=j}$, $I^S_{\geq j}$ and $I^S_{>j}$ analogously. We say that a set $C \subseteq I^S$ is a *pseudoclustering* of S if for any point $p \in X \cap S$ there exists a cluster in C containing p . For any pseudoclustering C of S , call $I^S_{<j} \cap C$ the *projection* of C onto $I^S_{<j}$ (and similarly for $I^S_{\leq j}$).

Now, we are ready to describe the algorithm. Given an instance $(X, F, 2, \alpha)$ of min-size k -clustering, the algorithm assigns levels to all possible clusters defined by X and implicitly defines horizontal and vertical lines on the plane as discussed above. Then, for each possible value of $r, s \in \{0, \dots, \lambda-1\}$, it executes an iteration. In each iteration, a k -clustering is computed; the best k -clustering among all iterations is output as the final solution. In each iteration associated with r, s , the algorithm processes all relevant squares defined by the plane subdivision according to r and s in a bottom-up fashion (i.e., in decreasing order of levels). At

a relevant j -square S , the projections of polynomially many pseudoclusterings of S are enumerated. During this enumeration process, a table $Table_S$ is constructed by looking up the respective entries stored in tables at children of S . The entry $Table_S(P, i)$ for a projection $P \subseteq I_{< j}^S$ of a pseudoclustering of S onto $I_{< j}^S$ and an integer i such that $1 \leq i \leq k$, will be a set $J \subseteq I_{\geq j}^S$ such that $P \cup J$ is a pseudoclustering of S with exactly i clusters. At the end of each iteration, the algorithm computes a k -clustering by enumerating all clusterings obtained by choosing entries from each table $Table_S$ taken over all relevant squares S having no parent.

1. $Table_S \leftarrow \emptyset$
2. $I_{\leq j}^S \leftarrow$ all clusters in \mathcal{D} of level at most j intersecting S
3. for all $Q \subseteq I_{\leq j}^S$ such that $|Q| \leq \min\{\xi, k\}$ do
4. $J \leftarrow \{D \in Q \mid D \text{ has level } j\}$
5. $P \leftarrow \{D \in Q \mid D \text{ has level smaller than } j\}$
6. if S has no children then
7. $Table_S(P, |Q|) \leftarrow J$
8. else
9. let S_1, S_2, \dots, S_t be the child squares of S
10. for each child square S_y do
11. $P'(S_y) \leftarrow \{D \in Q \mid D \text{ intersects } S_y\}$
12. for each possible combination of (i_1, i_2, \dots, i_t)
13. with $1 \leq i_y \leq k$ for $y = 1, \dots, t$ do
14. $J' \leftarrow J \cup \bigcup_{y=1}^t Table_{S_y}(P'(S_y), i_y)$
15. $i' = |P \cup J'|$
16. if $i' \leq k$ and $P \cup J'$ is a pseudoclustering of S then
17. if $Table_S(P, i')$ is undefined
18. or $\omega(J') < \omega(Table_S(P, i'))$ then
19. $Table_S(P, i') \leftarrow J'$

Fig. 2. The pseudocode for computing $Table_S$ once the tables $Table_{S'}$ have been computed for all children S' of S and all values of i

In Figure 2, we present the pseudocode for computing $Table_S$ once the tables $Table_{S'}$ have been computed for all children S' of S and all values of i . The parameter ξ is used to constrain the size of pseudoclusterings of S considered. We use $\omega(\cdot)$ to denote the cost of a cluster or the total cost of a set of clusters.

The algorithm executes λ^2 iterations. In each iteration, at most $O(n^2)$ relevant squares are processed. Using Lemma 1, we can easily see that the time required for computing the table entries for each relevant square is at most $n^{O(\lambda^4 + \xi)}$. Since the number of relevant squares having no parent in each iteration is at most 4, the last step of each iteration completes in polynomial time. Overall, the running time of the algorithm is $n^{O(\lambda^4 + \xi)}$.

In the following, we present the main arguments for analyzing the performance of the algorithm. Let $(X, F, 2, \alpha)$ be an instance of min-size k -clustering and consider all solutions which, for any square of side ℓ contain at most ξ clusters of radius at least $\frac{\ell}{2(\lambda+1)^2}$ that can include all the points of X in the square.

We call such solutions ξ -solutions for instance $(X, F, 2, \alpha)$. Clearly, for any relevant j -square defined by the plane subdivision according to r, s , a ξ -solution for $(X, F, 2, \alpha)$ contains at most ξ clusters of level at most j covering all the points of X in the square. The proof of the efficiency of the algorithm will be based on the comparison of the cost of the solutions obtained with the cost of the best ξ -solution. This will follow by Lemmas 2 and 3. First, in Lemma 2, we show that the cost of the solution computed by the algorithm in an iteration associated with r, s is upper-bounded by a quantity defined as a function of the cost of the clusters in the best ξ -solution and the plane subdivision defined by r, s . Then, in Lemma 3, we show that there are values of r, s such that this latter quantity (and, consequently, the cost of the best solution computed by the algorithm) is not much larger than the cost of the best ξ -solution.

Denote by C^* the best ξ -solution of instance $(X, F, 2, \alpha)$. For any relevant j -square S , denote by $C^*(S)$ the clusters of level j in C^* intersecting S .

Lemma 2. *Let $r, s \in \{0, \dots, \lambda - 1\}$ and $\mathcal{S}(r, s)$ be the set of relevant squares defined by r, s and X . In the iteration associated with r, s , the algorithm computes a k -clustering $A(r, s)$ of X of cost $\omega(A(r, s)) \leq \sum_{S \in \mathcal{S}(r, s)} \omega(C^*(S))$.*

Proof. Since C^* is a ξ -solution, we may assign each point to exactly one cluster so that all points are assigned to some cluster and the number of clusters intersecting with some square S which have been assigned points contained in S is at most ξ . We call a cluster intersecting with a relevant square S and having been assigned a point of S , a cluster associated with S .

For any relevant j -square S , let C^S be the set of clusters in C^* associated with S . Define $C_{<j}^S, C_{\leq j}^S$ and $C_{=j}^S$ as usual. We claim that after $Table_S$ has been computed, it holds

$$\omega(Table_S(C_{<j}^S, |C^S|)) \leq \sum_{S' \prec S} \omega(C^*(S')), \tag{1}$$

where $S' \prec S$ denotes that S' is a relevant square that is contained in S . Note that $S \prec S$.

The proof is by induction on the order in which the relevant squares are processed during an iteration. It is trivially true when S has no children. Assume that the algorithm is about to process the relevant j -square S and that (1) holds for all squares processed before S . In one of the iterations of the outer loop in the pseudocode of Figure 2, we have $Q = C_{\leq j}^S$ (and $J = C_{=j}^S$). In this iteration, consider the combination (i_1, i_2, \dots, i_t) such that $P'(S_y) = C_{\leq j}^{S_y}$ and $i_y = |C^{S_y}|$ for any $1 \leq y \leq t$. Observe that for each j' -square which is a child of S , it is $C_{\leq j}^{S'} = C_{<j'}^{S'}$. Also, clearly, it is $C_{=j}^S \subseteq C^*(S)$. Thus, the minimum cost set J' such that $P \cup J'$ is a pseudoclustering of S and $|P \cup J'| = |C^S|$ assigned to the entry $Table_S(C_{<j}^S, |C^S|)$ has cost at most

$$\sum_{S' \text{ child of } S} \omega(Table_{S'}(C_{\leq j}^{S'}, |C^{S'}|)) + \omega(C^*(S)) \leq \sum_{S' \prec S} \omega(C^*(S')),$$

and, hence, (1) holds also for S .

Finally, let $\mathcal{S}_0(r, s)$ be the set of all relevant squares without a parent. Once again the algorithm performs a complete enumeration of all possible solutions obtained by choosing exactly one entry from each table $Table_S$ for all $S \in \mathcal{S}_0(r, s)$. By applying the same argument used above and using the fact that for any relevant j -square $S \in \mathcal{S}_0(r, s)$ it is $C_{<j}^S = \emptyset$, we obtain that $\omega(A(r, s)) \leq \sum_{S \in \mathcal{S}_0(r, s)} \omega(Table_S(\emptyset, |C^S|)) \leq \sum_{S \in \mathcal{S}(r, s)} \omega(C^*(S))$. \square

Lemma 3. *There exist $r, s \in \{0, 1, \dots, \lambda - 1\}$ such that $\sum_{S \in \mathcal{S}(r, s)} \omega(C^*(S)) \leq (1 + \frac{6}{\lambda}) \omega(C^*)$.*

Similar statements with Lemma 3 are proved in [9,14]. So far (by combining Lemmas 2 and 3), we have bounded the cost of the best solution computed by the algorithm after all iterations in terms of the cost of the best ξ -solution. In the next section, we prove that for any instance $(X, F, 2, \alpha)$ the optimal solution (for $\alpha = 1$) or approximate solutions (for $\alpha > 1$) are essentially ξ -solutions. Combining the analysis above with Lemmas 4 and 5, we can bound the cost of the solution computed by our algorithm in terms of the cost of the optimal solution. By appropriately setting the parameters λ and ξ in terms of ϵ (for any $\epsilon > 0$), we obtain the following theorems.

Theorem 4. *There exists an algorithm for min-size k -clustering which, for each instance $(X, F, 2, 1)$ of the problem, computes an $(1 + \epsilon)$ -approximate solution in time $n^{O(1/\epsilon^4)}$ for any $\epsilon > 0$.*

Theorem 5. *There exists an algorithm for min-size k -clustering which, for each instance $(X, F, 2, \alpha)$ of the problem, computes an $(1 + \epsilon)$ -approximate solution in time $n^{O(\alpha^4/\epsilon^6)}$ for any $\epsilon > 0$.*

4 The Structure Lemmas

The following lemmas imply that for any instance $(X, F, 2, \alpha)$ of the min-size k -clustering problem, there exist constant values for ξ such that any optimal solution (for $\alpha = 1$) or at least a particular approximate solution (for $\alpha > 1$) are essentially ξ -solutions (and, hence, the best ξ -solution is optimal or almost optimal, respectively).

Lemma 4 (Structure Lemma). *For any integer constant $\lambda > 1$, there exists a constant $\xi = \xi(\lambda) = O(\lambda^4)$ such that the following is true: For any square S of side length ℓ , any optimal solution for any instance $(X, F, 2, 1)$ of the min-size k -clustering problem, contains at most ξ clusters of radius at least $\frac{\ell}{2(\lambda+1)^2}$ which intersect with S .*

A slightly different version of this Structure Lemma can also be found in [14]. For the case $\alpha > 1$, we cannot show a statement as strong as Lemma 4. Actually, it can be shown that there exist instances $(X, F, 2, \alpha)$ with $\alpha > 1$ and squares S of side length ℓ such that optimal solutions for $(X, F, 2, \alpha)$ contain

an unbounded number of clusters of radius at least $\frac{\ell}{2(\lambda+1)^2}$ intersecting with S . However, we can prove the next *Approximate Structure Lemma* which states that an approximate solution is a ξ -solution and, hence, it suffices for our purposes.

Lemma 5 (Approximate Structure Lemma). *For any constants $\gamma > 0$, $\alpha > 1$, and integer $\lambda > 1$, there exists a constant $\xi = \xi(\lambda, \alpha, \gamma) = O\left(\frac{\alpha^2 \lambda^4}{\gamma^2}\right)$ such that the following is true: Any instance $(X, F, 2, \alpha)$ of the min-size k -clustering problem has an $(1 + \gamma)^\alpha$ -approximate solution which, for any square S of side ℓ , contains a subset of at most ξ clusters of radius at least $\frac{\ell}{2(\lambda+1)^2}$ which contain all points in S .*

Proof. Consider an instance $(X, F, 2, \alpha)$ of the min-size k -clustering problem and an optimal solution D_{OPT}^* for $(X, F, 2, \alpha)$. Let ψ_{OPT} be a function that assigns to each point of X a cluster of D_{OPT}^* containing this point. We obtain a $(1 + \gamma)^\alpha$ -approximate solution D^* by increasing the radius of each disk in D_{OPT}^* by a factor of $1 + \gamma$. Define the assignment ψ which assigns each point of X to the smallest cluster (i.e., the one with the smallest radius) of D^* that contains it. We will show that, for any square of side length ℓ , the number of clusters of D^* of radius at least $\frac{\ell}{2(\lambda+1)^2}$ which are assigned by ψ to points of S is at most

$$\xi(\lambda, \alpha, \gamma) = \left(\left(6\sqrt{2} + \frac{4\sqrt{2}}{\gamma} \right) \frac{\alpha(1 + \gamma)(\lambda + 1)^2}{\ln 2} + 1 \right)^2 + 1.$$

Let S be a square of side length ℓ . Denote by X_1 and X_2 the sets of points of S assigned by ψ_{OPT} to clusters of D_{OPT}^* of radii smaller than $\ell\sqrt{2}/\gamma$ and at least $\ell\sqrt{2}/\gamma$, respectively. All points in X_1 are assigned to clusters of D^* of radii smaller than $\ell\sqrt{2}(1 + 1/\gamma)$ by ψ . Furthermore, the radius of the clusters of D_{OPT}^* to which points of X_2 are assigned by ψ_{OPT} is increased by at least $\ell\sqrt{2}$ and, hence, the resulting clusters of D^* cover the whole square. Among these clusters, denote by d the one with the smallest radius. The points of X_2 (if any) will be assigned either to cluster d or to clusters of radius smaller than $\ell\sqrt{2}(1 + 1/\gamma)$.

Now assume that more than $\xi(\lambda, \alpha, \gamma)$ clusters of D^* are assigned to points of the square S by ψ . This means that more than $\xi(\lambda, \alpha, \gamma) - 1$ clusters of radius larger than $\frac{\ell}{2(\lambda+1)^2}$ and at most $\ell\sqrt{2}/\gamma$ have their centers at distance at most $\left(\frac{3}{\sqrt{2}} + \frac{\sqrt{2}}{\gamma}\right)\ell$ from the center O of the square, otherwise, these clusters would not cover any point of S . Now shrink all these clusters around their centers to obtain disks of radius $\frac{2^{1/\alpha}-1}{4(1+\gamma)(\lambda+1)^2}\ell$. Let D' be the set of shrunk disks. We claim that any two disks of D' are disjoint. Assume otherwise and consider two disks of D' centered at points c_1 and c_2 of distance δ smaller than $\frac{(2^{1/\alpha}-1)\ell}{2(1+\gamma)(\lambda+1)^2}$. Let d_1 and d_2 be the clusters centered at c_1 and c_2 in the optimal solution D_{OPT}^* and let r and R be their radii. Without loss of generality, assume that $r \leq R$. Clearly, $r, R \geq \frac{\ell}{2(\lambda+1)^2(1+\gamma)}$. If $R \geq r + \delta$, then this means that the cluster d_2 could include all points included in the cluster d_1 , hence the solution D^* would not be optimal. If $R < r + \delta$, then we can include all points included in clusters

d_1 and d_2 in the solution D_{OPT}^* by increasing the radius of the cluster d_2 to $r + \delta$ and removing cluster d_1 from D_{OPT}^* . The new cost of cluster d_2 is now

$$\begin{aligned} F_{c_2} + (r + \delta)^\alpha &< F_{c_2} + \left(r + \frac{2^{1/\alpha} - 1}{2(1 + \gamma)(\lambda + 1)^2} \ell \right)^\alpha \leq F_{c_2} + (r + (2^{1/\alpha} - 1)r)^\alpha \\ &\leq F_{c_2} + 2r^\alpha \leq F_{c_1} + r^\alpha + F_{c_2} + R^\alpha \end{aligned}$$

which means that D^* is not optimal. Hence, all disks of D' are disjoint. By their definition, they are contained in a disk d' with radius $\left(\frac{3}{\sqrt{2}} + \frac{\sqrt{2}}{\gamma} + \frac{2^{1/\alpha} - 1}{4(1 + \gamma)(\lambda + 1)^2} \right) \ell$ centered at O . Since they are disjoint, their total area is more than

$$\begin{aligned} &(\xi(\lambda, \alpha, \gamma) - 1)\pi \left(\frac{(2^{1/\alpha} - 1)\ell}{4(1 + \gamma)(\lambda + 1)^2} \right)^2 \\ &\geq \left(\left(6\sqrt{2} + \frac{4\sqrt{2}}{\gamma} \right) \frac{\alpha(1 + \gamma)(\lambda + 1)^2}{\ln 2} + 1 \right)^2 \pi \left(\frac{(2^{1/\alpha} - 1)\ell}{4(1 + \gamma)(\lambda + 1)^2} \right)^2 \\ &\geq \left(\left(6\sqrt{2} + \frac{4\sqrt{2}}{\gamma} \right) \frac{(1 + \gamma)(\lambda + 1)^2}{2^{1/\alpha} - 1} + 1 \right)^2 \pi \left(\frac{(2^{1/\alpha} - 1)\ell}{4(1 + \gamma)(\lambda + 1)^2} \right)^2 \\ &\geq \pi \left(\frac{3}{\sqrt{2}} + \frac{\sqrt{2}}{\gamma} + \frac{2^{1/\alpha} - 1}{4(1 + \gamma)(\lambda + 1)^4} \right)^2 \ell^2 \end{aligned}$$

which contradicts the fact that they are completely contained in the disk d' . Hence, the number of clusters of D^* of radius at least $\frac{\ell}{2(\lambda + 1)^2}$ which are assigned to points of S cannot exceed $\xi(\lambda, \alpha, \gamma)$. \square

5 Extensions and Open Problems

Our techniques naturally extend to higher dimensions by using similar subdivisions of Euclidean spaces. Again, appropriate Structure Lemmas can be shown with slightly more complicated arguments. We can show the following statement.

Theorem 6. *There exists an algorithm for min-size k -clustering which, for each instance (X, F, d, α) of the problem, computes an $(1 + \epsilon)$ -approximate solution in time $n^{(\alpha/\epsilon)^{O(d)}}$ for any $\epsilon > 0$, $\alpha \geq 1$, and constant integer $d \geq 2$.*

The most important open problem is to explore the complexity of the problems in the case $\alpha = 1$. problems are still open for metric spaces as well; the best known approximability result is the constant approximation algorithm of [6].

In k -clustering to minimize the sum of cluster diameters, the cluster centers need not necessarily be points of X . Our polynomial-time approximation scheme for min-size k -clustering with $\alpha = 1$ yield a $(2 + \epsilon)$ -approximation algorithm for any constant dimension. To our knowledge, this is the best approximation guarantee for arbitrary k . Further improvements are also possible. Again, the complexity of the problem in multidimensional Euclidean spaces is still open.

References

1. S. Arora, P. Raghavan, and S. Rao. Approximation schemes for the Euclidean k -medians and related problems. In *Proc. of the 30th ACM Symposium on Theory of Computing (STOC '98)*, pp. 106-113, 1998.
2. M. Bădoiu, S. Har-Peled, and P. Indyk. Approximate clustering via core-sets. In *Proc. of the 34th Annual ACM Symposium on Theory of Computing (STOC '02)*, pp. 250-257, 2002.
3. Y. Bartal, M. Charikar, and D. Raz. Approximating min-sum k -clustering in metric spaces. In *Proc. of the 33rd Annual ACM Symposium on Theory of computing (STOC '01)*, p.11-20, 2001.
4. P. Brucker. On the complexity of clustering problems. *Optimization and Operations Research*, Lecture Notes in Economics and Mathematical Sciences, Vol. 157, pp. 45-54, 1978.
5. M. Charikar, S. Guha, E. Tardos, and D. S. Shmoys. A constant factor approximation algorithm for the k -median problem. *Journal of Computer and Systems Sciences*, Vol. 65 (1), pp. 129-149, 2002.
6. M. Charikar and R. Panigrahy. Clustering to minimize the sum of cluster diameters. *Journal of Computer and Systems Sciences*, Vol. 68 (2), pp. 417-441, 2004.
7. V. Capoyleas, G. Rote, and G. J. Woeginger. Geometric Clusterings. *Journal of Algorithms*, Vol. 12(2), pp. 341-356, 1991.
8. S. R. Doddi, M. V. Marathe, S. S. Ravi, D. S. Taylor, and P. Widmayer. Approximation algorithms for clustering to minimize the sum of diameters. *Nordic Journal of Computing*, Vol. 7(3), pp. 185-203, 2000.
9. T. Erlebach, K. Jansen, and E. Seidel. Polynomial-time approximation schemes for geometric graphs. In *Proc of the 12th Annual Symposium on Discrete Algorithms (SODA '01)*, pp. 671-679, 2001.
10. W. Fernandez de la Vega, M. Karpinski, C. Kenyon, and Y. Rabani. Approximation schemes for clustering problems. In *Proc. of the 35th Annual ACM Symposium on Theory of Computing (STOC '03)*, pp. 50-58, 2003.
11. A. Freund and D. Rawitz. Combinatorial interpretations of dual fitting and primal fitting. In *Proc. of the First International Workshop on Approximation and Online Algorithms (WAOA '03)*, LNCS 2909, Springer, pp. 137-150, 2003.
12. P. Hansen and B. Jaumard. Minimum sum of diameters clustering. *Journal of Classification*, Vol. 4, pp. 215-226, 1987.
13. K. Jain and V. V. Vazirani. Approximation algorithms for metric facility location and k -median problems using the primal-dual scheme and Lagrangian relaxation. *Journal of the ACM*, Vol. 48, pp. 274-296, 2001.
14. N. Lev-Tov and D. Peleg. Polynomial time approximation schemes for base station coverage with minimum total radii. *Computer Networks*, Vol. 47, pp. 489-501, 2005.
15. C. L. Monma and S. Suri. Partitioning points and graphs to minimize the maximum or the sum of diameters. *Graph Theory, Combinatorics and Applications*, John Wiley and Sons, pp. 880-912, 1991.
16. R. Ostrovsky and Y. Rabani. Polynomial-time approximation schemes for geometric clustering problems. *Journal of the ACM*, Vol. 49(2), pp. 139-156, 2002.

Approximation Schemes for Minimum 2-Connected Spanning Subgraphs in Weighted Planar Graphs^{*}

André Berger¹, Artur Czumaj², Michelangelo Grigni¹, and Hairong Zhao²

¹ Department of Mathematics and Computer Science,
Emory University, Atlanta GA 30322, USA
aberge2@emory.edu, mic@mathcs.emory.edu

² Department of Computer Science,
New Jersey Institute of Technology, Newark NJ 07102, USA
{czumaj, hairong}@cis.njit.edu

Abstract. We present new approximation schemes for various classical problems of finding the minimum-weight spanning subgraph in edge-weighted undirected *planar graphs* that are resistant to edge or vertex removal. We first give a PTAS for the problem of finding minimum-weight 2-edge-connected spanning subgraphs where duplicate edges are allowed. Then we present a new greedy spanner construction for edge-weighted planar graphs, which augments any connected subgraph A of a weighted planar graph G to a $(1 + \varepsilon)$ -spanner of G with total weight bounded by $\text{weight}(A)/\varepsilon$. From this we derive quasi-polynomial time approximation schemes for the problems of finding the minimum-weight 2-edge-connected or biconnected spanning subgraph in planar graphs. We also design approximation schemes for the minimum-weight 1-2-connectivity problem, which is the variant of the survivable network design problem where vertices have 1 or 2 connectivity constraints. Prior to our work, for all these problems no polynomial or quasi-polynomial time algorithms were known to achieve an approximation ratio better than 2.

1 Introduction

The survivable network design problem is a fundamental problem in algorithmic graph theory with numerous applications in computer science and operations research (see, e.g., [12,15,22]). The classical k -connectivity problems are perhaps the most extensively studied problems in network design. We are given a graph G with n vertices and a nonnegative weight $w(e)$ on each edge e , and we want to find a k -edge or k -vertex connected *spanning* subgraph S , such that its total edge weight $w(S)$ equals the minimum possible OPT. It is well-known that all non-trivial variants of the survivable network design problem are \mathcal{NP} -hard and therefore the main research interest lies in the design of efficient approximation algorithms (see, e.g., [15] for a survey).

^{*} Research supported in part by NSF grants CCR-0208929 and ITR-CCR-0313219.

In this paper we consider approximation algorithms for the most basic case of the survivable network design problem in which the resulting subgraphs should be resistant to the removal of a *single* edge or vertex. The two classical problems here are to find a minimum-weight 2-edge-connected (2-EC) spanning subgraph (a 2-ECSS) of G , or a 2-vertex-connected (2-VC or biconnected) spanning subgraph (a 2-VCSS) of G . We also consider a standard relaxation of the 2-ECSS problem of finding a minimum weight 2-EC spanning sub-*multigraph* (2-ECSSM) H of G , meaning that an edge of G can be used multiple times in H (consequently its weight is also counted multiple times in H). Another classical extension is the *1-2-connectivity problem*: each vertex v is assigned a connectivity type $r_v \in \{1, 2\}$. The problem is to find a minimum weight spanning subgraph such that for any pair of vertices $v, u \in V$, there are at least $r_{uv} = \min\{r_u, r_v\}$ edge-disjoint or vertex-disjoint paths between v and u . We denote the 1-2-edge-connectivity by $\{1,2\}$ -EC, and 1-2-vertex-connectivity by $\{1,2\}$ -VC. We also consider the relaxed 1-2-edge-connectivity problem where each edge may be used more than once.

All the problems mentioned above have been extensively studied in the literature. Since all these problems are \mathcal{NP} -hard, the main research has been devoted to design efficient approximation algorithms, see the survey [15] and more recent advances [4,10,11,14,17]. In general, we would prefer to design a *polynomial-time approximation scheme (PTAS)*, which is a c -approximation algorithm taking both G and c as inputs, and running in polynomial time for each fixed $c > 1$. However, all the problems we consider in this paper are max-SNP-hard [6], even for unweighted graphs or when duplicate edges are allowed; therefore they do not have a PTAS unless $\mathcal{P} = \mathcal{NP}$. But this does not preclude a PTAS for restricted classes of graphs: indeed, there exist PTAS's for all these problems in *geometric graphs* in low dimensions [6,8], and also for the 2-ECSS and 2-VCSS problems in *unweighted planar graphs* [5]. In fact, the approximation schemes of [5] allow *weighted* planar graphs, but then the algorithms will either run in time $n^{O(\frac{w(G)}{\varepsilon \cdot \text{OPT}})}$ to ensure an $(1 + \varepsilon)$ -approximate solution or they run in polynomial time with an approximation guarantee of $1 + O(\frac{w(G)}{\varepsilon \cdot \text{OPT}})$. Since the ratio $w(G)/\text{OPT}$ could be arbitrarily large, these algorithms are in general not PTAS's for weighted planar graphs.

For both the 2-ECSS and 2-VCSS problems in weighted planar graphs, the best known polynomial-time or even quasi-polynomial-time approximation guarantee is still 2 [16,20], which is achieved by polynomial-time algorithms working for general weighted graphs. On the other hand, besides the PTAS for unweighted planar graphs [5], there are better constant approximation guarantees known for general unweighted graphs. For example, there exists a $\frac{5}{4}$ -approximation algorithm for the unweighted 2-ECSS problem [14], and a $\frac{4}{3}$ -approximation algorithm for the unweighted 2-VCSS problem [23].

A similar phenomenon can be seen for the 1-2-connectivity problems. For both the unweighted $\{1,2\}$ -ECSS and the unweighted $\{1,2\}$ -VCSS problem, Krysta [18] gives $\frac{3}{2}$ -approximation algorithms. If the graph is weighted, the best known result for $\{1,2\}$ -ECSS is a 2-approximation algorithm, due to Jain [13],

which in fact solves the more general problem where $r_v \leq k$ for any k . For the weighted $\{1,2\}$ -VCSS problem, Fleischer [9] gives a 2-approximation algorithm, which actually solves the $\{0,1,2\}$ -VCSS problem. A PTAS for the geometric version of these problems is presented in [8].

1.1 New Contributions and Techniques

We present efficient approximation schemes for all the above mentioned problems in weighted planar graphs. Our approximation algorithms depend in a crucial way on our new construction of *light spanners* for planar graphs.

Let G be a weighted graph. We use $d_G(u, v)$ to denote the weighted shortest path distance between the vertices u and v in G . An s -spanner of G is a spanning subgraph H of G such that $d_H(u, v) \leq s \cdot d_G(u, v)$ for all u, v . A spanner provides an approximate representation of the shortest path metric (1-connectivity) in G , but it may be much lighter than G .

Althöfer et al. [1] designed a simple greedy algorithm that for an arbitrary graph G computes an s -spanner H of G for any $s > 1$. In the case of *planar* graphs, it is shown in [1] that this spanner has weight $w(H) \leq (1 + 2/(s - 1))\text{MST}(G)$, where $\text{MST}(G)$ is the weight of a minimum spanning tree in G . Since $\text{MST}(G) \leq \text{OPT}$ for all the problems we consider, this bounds the ratio $w(H)/\text{OPT}$ in terms of just s . If all weighted graphs in a graph family have spanners with such a bound on $w(H)/\text{OPT}$ (depending only on s), then we say the family has *light spanners* for this problem. Light spanners are known to be very useful for solving various optimization problems on graphs. For example, planar graphs have light spanners for metric-TSP: the first step in the metric-TSP PTAS for weighted planar graphs [2] is to replace the input graph with an accurate enough s -spanner (using [1]), thus effectively bounding $w(G)/\text{OPT}$ for the remainder of the algorithm. Spanners are also used in complete geometric graphs to design efficient PTAS's for geometric TSP and related problems [21], and to design PTAS's for the 2-edge and 2-vertex-connectivity problems [7,8].

By combining the spanner constructed in [1] with the planar separator decomposition approach tuned to analyze 2-connected graphs [5], we show that one can design a PTAS for the 2-ECSSM problem and a PTAS for the $\{1,2\}$ -ECSSM problem. However, this approach of replacing the input graph with an s -spanner fails for the 2-ECSS and 2-VCSS problems. The reason is that a spanner does not have to be 2-connected, thus may not contain the optimal or a near optimal solution in most cases. Naturally, one may think to use light *fault-tolerant* spanners (see, e.g., in [19]), which are subgraphs that persist as s -spanners even after deleting a constant number of vertices or edges. Unfortunately, this concept is not useful here, since simple examples show that light fault-tolerant spanners do not exist in weighted planar graphs, not even for a single edge deletion.

To solve the problem mentioned above, we present our main contribution: a new greedy spanner construction which produces a light planar spanner with certain desirable properties. Specifically, given a weighted planar graph G , a connected spanning subgraph A of G and $s > 1$, it computes an s -spanner H of G . H contains A as a subgraph and has total weight $w(H) = O(1/(s - 1)) \cdot$

$w(A)$). Thus if we feed the algorithm with α -approximate solutions H to the various connectivity problems in a weighted planar graph G , then we obtain an $O(\alpha/(s - 1))$ -approximation H^* for that problem, which at the same time is an s -spanner for G . Furthermore, we can show that while H^* need not contain a $(1 + \varepsilon)$ -approximate solution S , we can put a bound on the number of edges of S “crossing” each face of H^* (Lemma 3).

Using our new spanner construction technique and the planar separator decomposition, we design approximation schemes for the 2-ECSS and 2-VCSS, $\{1,2\}$ -ECSS and $\{1,2\}$ -VCSS problems, which find solutions with weight at most $(1 + \varepsilon) \cdot \text{OPT}$ in $n^{O(\log n \log(1/\varepsilon)/\varepsilon)}$ time; these are *quasi-polynomial time approximation schemes* (QPTAS’s).

Organization. We first present a PTAS for the 2-ECSSM problem in Section 2. This section contains also a description of the main algorithmic approach used in our approximation schemes, which is a combination of the use of spanners, a recursive approach driven by a variant of the planar separator theorem, and dynamic programming. Next, in Sections 3 and 4, we describe our new construction of spanners and discuss the special properties of the spanners. In Section 5, we present quasi-polynomial approximation schemes for the 2-ECSS and the 2-VCSS problems. Finally in Section 6, we consider $\{1,2\}$ -ECSS and $\{1,2\}$ -VCSS problems: we show a PTAS for the $\{1,2\}$ -ECSSM problem, and a QPTAS for each of the $\{1,2\}$ -ECSS and $\{1,2\}$ -VCSS problems.

2 PTAS for the 2-ECSSM Problem

Let G be a connected weighted graph. A 2-ECSSM H of G is a spanning submultigraph of G in which edges can have some multiplicity and in which every pair of vertices is connected by at least two edge-disjoint paths. Note that G may not have any multiple edges at all. If an edge is used multiple times in H , its weight also contributes multiple times to the weight of H . Since it never helps to use an edge more than twice, we may cap all edge multiplicities at two. We now present a PTAS for this problem, running in $n^{O(1/\varepsilon^2)}$ time.

Given G and $\varepsilon > 0$, we choose s so that $s^2 \leq 1 + \varepsilon$. We first compute an s -spanner H in G by the greedy spanner algorithm [1], with weight $w(H) = O((1/\varepsilon) \cdot \text{OPT})$. Now we show that there is a $(1 + \varepsilon)$ -approximate 2-ECSSM that uses only edges from H . Suppose S^* is an optimal 2-ECSSM in G with $w(S^*) = \text{OPT}$. Now we modify S^* such that it uses only edges from H . For each edge e of S^* not in H , we remove e and add a shortest path from H of total weight at most $s \cdot w(e)$. When we add the path, we add the edges with multiplicity, but capped at two. The result of all these modifications is another 2-ECSSM S , using only edges from H , each edge used at most twice, with $w(S) \leq s \cdot \text{OPT}$.

Next we apply the 2-ECSS s -approximation algorithm from [5] to the graph H' , which is H with each edge duplicated. Since this algorithm forms also a core of our algorithm for other problems discussed later in Sections 5 and 6, we briefly describe it here. The algorithms in [5] use a recursive approach driven by the following planar separator theorem from [3] (see also [5]):

Lemma 1. *Let G be a connected planar graph on $n \geq 3$ vertices embedded in the plane. Suppose G has non-negative weights on its vertices, edges and faces, and non-negative costs on its edges. Let W be the total weight of the graph and let M be its total cost and assume that no edge has weight more than $(3/4) \cdot W$. Then for any positive integer k , we can find a subgraph F of G and a closed Jordan curve J in $O(n)$ time such that:*

1. *F is the union of at most two vertex-disjoint simple cycles (maybe none). The total cost of the edges on each cycle is at most M/k . If F contains two cycles A and C , then $\text{interior}(C) \subset \text{interior}(A)$. The interior of C and the exterior of A (if they exist) both have weight at most $W/2$.*
2. *Denote by G' the embedded graph that results after deleting the interior of C and the exterior of A (if they exist) and contracting each cycle in F to a vertex of weight 0. Then J is a Jordan curve which intersects edges of G' only at their endpoints and passes through $O(k)$ vertices (“portals”) including the new contracted vertices. The interior and exterior of J both have weight at most $(3/4) \cdot W$.*

First, we decompose H' according to Lemma 1 (with $k = \Theta((\log n/\varepsilon) \cdot \frac{w(H')}{\text{OPT}}) = \Theta(\log n/\varepsilon^2)$) into at most four pieces: the interior of the cycle C , the exterior of the cycle A , and the interior and the exterior of the Jordan curve J . By assigning weight to the new portals and faces properly, we can make sure that each piece has weight at most a constant fraction of the H' . We continue to decompose the small pieces recursively. It is easy to see that the depth of the recursion is logarithmic, and the number of pieces is $O(n \log n)$. By the weighting scheme of the new portals ([2,5]), one can show that each piece has a portal set P with size $O(k)$.

We need to find a low cost spanning subgraph in each piece and then combine them together to form an almost minimum weight 2-ECSS of H' . Of course we do not know the remaining subgraph outside this piece. Thus, for each piece, we enumerate all the different ways that some subgraph of H' (outside this piece) may influence the connectivity constraints within this piece. We call these the *external types* of the piece, and one can show that the number of such types is $2^{O(|P|)} = n^{O(1/\varepsilon^2)}$ [5, Lemma 2.4], where P is the set of portals of this piece.

For each piece and each external type, we must find a near minimum cost subgraph of the piece, so that this subgraph together with the external type can meet the global connectivity constraints. We use dynamic programming to solve the subproblems in each of the pieces.

During the course of the algorithm, we always commit the cycle edges found in the separator to the solution, which is the only source of the error in our algorithm. Since the cycle edges has total cost at most $w(H')/k$ at each level of the recursion and there are at most $O(\log n)$ levels, by setting appropriately the leading constant in k , we can show that the total error introduced in the algorithm is $\varepsilon \cdot \text{OPT}$. For each piece, the number of types is bounded by $n^{O(w(H')/(\varepsilon \cdot \text{OPT}))} = n^{O(1/\varepsilon^2)}$. There are $O(n \log n)$ pieces. Therefore the algorithm solves $n^{O(1/\varepsilon^2)}$ subproblems, each in time $n^{O(1/\varepsilon^2)}$.

In summary, we have the following theorem.

Theorem 1. *Let $\varepsilon > 0$ and let G be a connected weighted planar graph with n vertices. There is an algorithm running in time $n^{O(1/\varepsilon^2)}$ that outputs a 2-ECSSM of G whose weight is at most $(1 + \varepsilon)$ times the minimum.*

The above technique does not work for other problems considered in this paper, because there we are not allowed to duplicate edges from G in the output graph. Instead, our approximation schemes must consider the possibility that the near-optimal S needs some “extra” edges from outside the spanner. In Sections 3 and 4 we develop a new type of light planar spanners and we limit the number and arrangements of those extra edges outside the spanner.

3 Augmented Planar Spanners

In this section we present a new greedy algorithm constructing s -spanners in weighted planar graphs, resembling the standard greedy algorithm [1] for general graphs. Just as in the standard algorithm, we take a connected weighted graph G and a parameter $s \geq 1$, and produce an s -spanner H . Unlike the general algorithm, our G must be planar, and for each edge e of G not in H we guarantee that $s \cdot w(e)$ is at least the length of some path in the face of H containing e . We also provide our algorithm with a third argument: a “seed” spanning subgraph A , containing edges that must appear in H . In Section 4 we will use A to enforce some 2-connectivity properties in the spanner.

Suppose G is a weighted *plane graph* (that is, an embedded planar graph) and H is a subgraph. A *chord* e of H is an edge of G not in H . Note that H and e inherit embeddings from G . For each chord e we define $w_H(e)$ as the length of the shortest walk connecting the endpoints of e , along the boundary of the face of H containing e .

More precisely, if the endpoints of e are disconnected in H , then we define $w_H(e) = +\infty$. Otherwise e connects two vertices in a component of H , and e is embedded in some face f of this component. The boundary of f is a cyclic walk of (oriented) edges, with total weight $w(f)$; note that a cut-edge may appear twice in the boundary (once per orientation), and its weight would then count twice in $w(f)$. Similarly a cut-vertex may appear multiple times. The edge e splits the boundary sequence into two walks P_1 and P_2 , both connecting the endpoints of e , with $w(P_1) + w(P_2) = w(f)$. Now we define $w_H(e) = \min(w(P_1), w(P_2))$ (see Figure 1).

Given G , s , and A as above, we compute $H = \text{Augment}(G, s, A)$ as follows:

```

Augment( $G, s, A$ ):
   $H \leftarrow A$ 
  for all edges  $e$  of  $G$  in non-decreasing  $w(e)$  order do
    if  $e$  is not in  $H$  and  $s \cdot w(e) < w_H(e)$  then
      add  $e$  to  $H$ 
  return  $H$ 
    
```

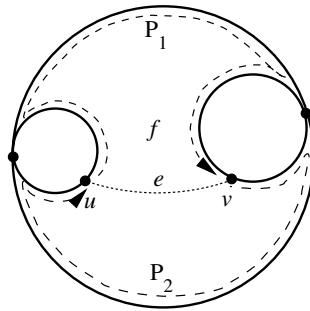


Fig. 1. A non-simple face f in H , a chord e , and walks P_1 and P_2

Note $A \subseteq H \subseteq G$. If A is empty (has all vertices of G but no edges), then this is like the general greedy spanner algorithm [1], except that we have w_H in place of d_H .

Theorem 2. *Let G be a weighted plane graph, $s > 1$, and A a spanning subgraph of G . Then $H = \text{Augment}(G, s, A)$ is an s -spanner of G . If A is connected, then $w(H) \leq (1 + 2/(s - 1)) \cdot w(A)$.*

Proof. To show that H is an s -spanner it suffices to show that each edge of G is s -approximated in H . For e not in H , at the moment it was rejected we had $w_H(e) \leq s \cdot w(e)$. Note that $w_H(e)$ may only decrease after that, so $d_H(e) \leq w_H(e) \leq s \cdot w(e)$ at the end of the algorithm.

For the second part we need to show that the weight of all edges in H but not A is at most $(2/(s - 1)) \cdot w(A)$. Suppose e is such an edge; then e is not a cut edge in H since A is a connected spanning subgraph. Therefore e is bounded by two distinct faces. Let f be either face bounding e . We first claim that $w(f) > (1 + s) \cdot w(e)$. To see this, consider the *last* edge e' added to f whose boundary consists of a path P plus e' . Since e' is added to H , we must have that $s \cdot w(e') < w_H(e')$ and $w_H(e') \leq w(P)$. Adding $w(e')$ to both sides of $s \cdot w(e') < w(P)$, and noting $w(e) \leq w(e')$, we get the claim.

For each face f of A , let E_f be the set of edges in H crossing the interior of f . Since the sum of $w(f)$ over all faces of A is $2 \cdot w(A)$, it suffices to show that $w(E_f) \leq (1/(s - 1)) \cdot w(f)$. Note that the edges dual to E_f define a tree on the faces of H inside f . Orient this dual tree away from some arbitrarily chosen root: now for each $e \in E_f$, we have chosen an adjacent face f_e of H (only the root was not picked). For each $e \in E_f$ we know $w(f_e) - 2 \cdot w(e) > (s - 1) \cdot w(e)$, from the previous paragraph. Summing these inequalities over all $e \in E_f$, we get at most $w(f)$ on the left hand side, and exactly $(s - 1) \cdot w(E_f)$ on the right.

4 Spanners and 2-EC Subgraphs

Suppose we are given a weighted plane 2-EC graph G and we want to find a $(1 + \epsilon)$ -approximate 2-ECSS. We first construct an auxiliary subgraph H^* as follows:

1. Compute a 2-approximate 2-ECSS A , in polynomial time.
2. Compute $H^* = \text{Augment}(G, \sqrt{2}, A)$.

The constant $\sqrt{2}$ here is not critical, just convenient. By Theorem 2, H^* is a 12-approximate 2-ECSS. Below we show that for every $\varepsilon > 0$, this H^* has nice intersection properties with some $(1 + \varepsilon)$ -approximate 2-ECSS in G .

Given a face f in H^* , the chords of f are the edges of G embedded inside this face, according to G 's embedding. A *face-edge* e of f is an abstract edge connecting two vertices of f ; unlike a chord, a face-edge is not necessarily an edge of G . (If vertices appear more than once on f , we must specify which appearances we want as the endpoints of e .) We say the face edge e *crosses* a chord c if: c is a chord of the same face f , their endpoints are distinct vertex appearances on f , and they appear in cyclic “ $ecec$ ” order around the boundary of f . Note that we may embed e inside f so e intersects only the crossed chords.

Suppose S is a 2-ECSS in G , and an edge c of S is not in H^* . Then c is a chord of some face f of H^* . Let P_c be the path in f connecting the endpoints of c , such that $w(P_c) \leq \sqrt{2} \cdot w(c)$. Then the *chord move at c* is the following modification of S : add to S all the edges of P_c that were not already in S , and remove from S any chords inside the cycle $c \cup P_c$ (see Figure 2(a)). Since H^* is 2-EC, the cycle has no repeated edges, and therefore S is still a 2-ECSS after the chord move. The chord move is *improving* if $w(S)$ decreases; this happens whenever $w(P_c)$ (or $\sqrt{2} \cdot w(c)$) is less than the weight of the discarded chords. Any non-trivial chord move brings S closer to H^* (in Hamming distance), thus at most $O(n)$ improving chord moves apply to any given 2-ECSS S .

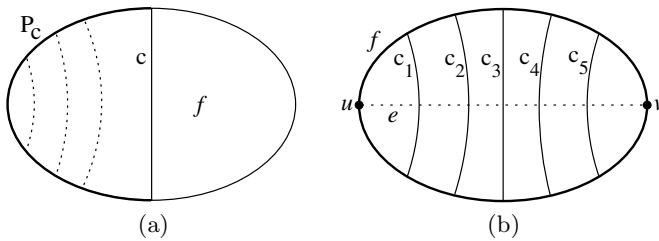


Fig. 2. (a) Face f of H^* (oval) with chord c , path P_c (bold), and chords removed from S by the chord move at c (dotted). (b) Face f with a face-edge e (dashed) crossed by five chords from S .

Lemma 2. *Let e be a face edge of a face f in H^* and S be a 2-ECSS of G . Suppose C is a set of edges of S all of which are chords crossing e , $\sqrt{2} \cdot \min_{c \in C} w(c) > \max_{c \in C} w(c)$, and no chord in C gives an improving chord move. Then $|C| \leq 4$.*

Proof. If not, S has five chords crossing e as in Figure 2(b). But then we have an improving chord move at c_3 , since the discarded chords ($\{c_1, c_2\}$ or $\{c_4, c_5\}$) weigh more than $\sqrt{2} \cdot w(c_3)$.

Now we argue that by accepting a small additive error in our 2-ECSS, we may assume it has only a small number of chords crossing a given face-edge:

Lemma 3. *Suppose G and H^* are as above, $\varepsilon > 0$, S is a 2-ECSS, f is a face in H^* , and e is a face-edge in f . Then there exists a 2-ECSS S' such that $w(S') \leq w(S) + \varepsilon \cdot w(C'_f)$, where C'_f is the set of edges of S' that are chords crossing e , $C'_f \subseteq S$, and $|C'_f| = O(\log(1/\varepsilon))$.*

Proof. First we may suppose that S has no improving chord move at a chord crossing e , since such a move could only remove some chords crossing e . Let C_f be the set of chords in S crossing e . Arrange C_f in “left to right” order, according to how they intersect e . Let $c_0 \in C_f$ be the chord with maximum weight. Say that a chord $c \in C_f$ is *short* if $w(c) \leq \varepsilon \cdot w(c_0)/(2\sqrt{2})$. Now if there are short chords to the left of c_0 , perform a chord move at the rightmost one, c_l . Similarly if there are short chords to the right of c_0 , perform a chord move at the leftmost one, c_r . S' is the result of these (at most) two chord moves; note that C'_f contains no short chords except possibly c_l and c_r .

Map each non-short chord $c \in C'_f$ to the real number $\log(w(c_0)/w(c))$, a point in the real interval $I = [0, \log(1/\varepsilon) + 3/2]$. Note that two edges can be mapped to the same semi-open subinterval of I of length $1/2$ only if the heavier edge has weight less than $\sqrt{2}$ times that of the lighter edge. By Lemma 2, at most four edges can be mapped into the same subinterval of I of length $1/2$. This implies $|C'_f| = O(\log(1/\varepsilon))$.

The chord moves in f increased $w(S')$ by at most $\sqrt{2}(w(c_l) + w(c_r)) \leq \varepsilon \cdot w(c_0)$, which is at most $\varepsilon \cdot w(C'_f)$.

Remarks: In the 2-VCSS case, the initial A should be a 2-approximate 2-VCSS, so that H^* is a 12-approximate 2-VCSS. Then in the chord move the cycle has no repeated vertices, therefore S remains a 2-VCSS after the move. In Lemmas 2 and 3, the only properties of H^* that we needed were that it was 2-EC (or 2-VC), and that $w_{H^*}(e) \leq \sqrt{2} \cdot w(e)$ for each chord e .

5 Approximation Schemes for the 2-ECSS and 2-VCSS Problems

In this section we will show how to use our new spanner construction to find quasi-polynomial time approximation schemes for the 2-ECSS and the 2-VCSS problems in weighted planar graphs. We start with the QPTAS for 2-ECSS problem.

We use a similar framework as that in the PTAS for 2-ECSSM problem in Section 2. But instead of using the spanner constructed as in [1], now we use the augmented spanner H^* as constructed in Section 4.

We first apply Lemma 1 to H^* with $k = \Theta(\log n/\varepsilon)$ to decompose H^* . However, different from the PTAS for the 2-ECSSM problem, H^* may not contain a near-optimal solution of the 2-ECSS problem. Thus we cannot work on the pieces of H^* directly. Fortunately, Lemma 3 guarantees that there exists a near-optimal solution with at most $O(k \log(1/\varepsilon))$ edges crossing the Jordan curve J . We guess these crossing edges by trying all $n^{O(k \log(1/\varepsilon))}$ possibilities. We add the

guessed edges to the corresponding pieces. The vertices of H^* along J together with the endpoints of the guessed edges determine the set of portals for the new pieces. For each new piece with its guessed edges, we assign weights to the new portals such that each new piece has cost at most a constant fraction of H^* and $O(k)$ portals. Then we recursively decompose the new pieces.

As in the PTAS for the 2-ECSSM problem, for each piece we define edge-connectivity types which describe how these portals may be connected outside this piece in a $(1 + \varepsilon)$ -approximate solution. The number of types for each piece is $2^{O(k \log(1/\varepsilon))}$. Then we use dynamic programming to solve the subproblems as before and we commit the cycle edges to the solution.

The approximation scheme for the 2-VCSS problem is similar and we only mention the differences: first, we redefine H^* as remarked at the end of Section 4. Then we need to define vertex-connectivity types using the same techniques as in [5].

The error of our final solution comes from two sources. First, we committed the edges of the cycles that arose from the application of the separator theorem to the solution. Since each piece in the decomposition has weight at most constant fraction of its parent weight, the depth of the recursive calls is $O(\log n)$. As before, the total error per recursive level is $O((w(H^*)/k) \log n)$, where $k = \Theta(\log n/\varepsilon)$ and $w(H^*) = O(\text{OPT}/\varepsilon)$. By an appropriate choice of the leading constant defining k , this is at most $(\varepsilon/2) \cdot \text{OPT}$. Moreover, each time a face of H^* (or its pieces) is cut by a Jordan curve, we guess $O(\log(1/\varepsilon))$ crossing edges. If we guess these edges optimally (they were edges in some original optimal S^*), then by Lemma 3 we may pay an additive error of at most $\varepsilon/2$ times the weight of these guessed edges. Summing over the entire assembly of a possible solution, the total of these errors is at most $(\varepsilon/2) \cdot \text{OPT}$.

The dominating factor in the running time comes from trying all $n^{O(k \log(1/\varepsilon))}$ possibilities for the guessed edges. The weights of the subproblems are only a constant times the weight of their respective parents and therefore a pure recursive approach (without dynamic programming) leads to a time bound of $T(n) \leq n^{O(k \log(1/\varepsilon))} T(c \cdot n)$ ($0 < c < 1$), with solution $n^{O((1/\varepsilon) \cdot \log(1/\varepsilon) \cdot \log^2 n)}$. We may improve this bound by a logarithmic factor in the exponent by using dynamic programming and by a more careful count of subproblems. It has been proved in [3] that for each piece, one can find a list of $O(n^2)$ separations such that for any valid weight scheme of the vertices, edges and faces of the piece, some separation in this list satisfies the properties of Lemma 1. This implies:

Lemma 4. *The total number of distinct pieces (contracted subgraphs) of the original H^* that occur during our recursive decomposition is $n^{O(\log n)}$. Therefore the number of distinct subproblems (a piece, $|P| = O(k \log(1/\varepsilon))$ portals selected in the piece, and an external connectivity type on those portals) is $n^{O(\log n)} n^{O(|P|)} 2^{O(|P|)} = n^{O(k \log(1/\varepsilon))}$.*

Theorem 3. *Let $\varepsilon > 0$ and let G be a 2-EC (2-VC) weighted planar graph with n vertices. There is an algorithm running in time $n^{O(\log n \cdot \log(1/\varepsilon)/\varepsilon)}$ that outputs a 2-ECSS (2-VCSS) H of G such that $w(H) \leq (1 + \varepsilon) \cdot \text{OPT}$.*

6 Extensions to the $\{1, 2\}$ -Connectivity Problem

In this section, we extend our results to the $\{1, 2\}$ -connectivity problems in weighted planar graphs. We focus on the algorithm for the $\{1, 2\}$ -ECSS problem only. The algorithm for the $\{1, 2\}$ -VCSS problem can be obtained similarly. The algorithms are modifications of the respective algorithms in Sections 2 and 5.

First consider the $\{1, 2\}$ -ECSSM problem, a relaxed version of the $\{1, 2\}$ -ECSS problem, where duplicate edges are allowed. As in Section 2, we can show there is a $(1 + \varepsilon)$ -approximate $\{1, 2\}$ -ECSSM that uses only edges from a light $(1 + \varepsilon)$ -spanner H . So we can work on H with duplicated edges instead of G .

The main difference from Section 2 is the dynamic programming part. We need to redefine the connectivity types to reflect the non-uniform connectivity requirements. For this, we can use the connectivity type construction in [8]. Each time we contract a 2-connected component or path, we assign the highest connectivity requirement among all contracted vertices to the new vertex. This increases the number of types from $2^{O(|P|)}$ to $2^{O(2|P|)}$, where P is the set of portals in the given graph. We again obtain a PTAS with running time $n^{O(1/\varepsilon^2)}$.

Now consider the $\{1, 2\}$ -ECSS problem. We first find a 2-approximate solution A using algorithms from [13]. Then we augment A into a light spanner H^* as in Section 4. Using similar arguments as in the proof of Lemma 3, we can show that there is a $(1 + \varepsilon)$ -approximate $\{1, 2\}$ -ECSS S so that for each picked face-edge e , only $O(\log(1/\varepsilon))$ edges of S cross e . Now redefine the connectivity types as above and use dynamic programming to solve the problem. The running time is still dominated by the number of subproblems $n^{O(\log n \cdot \log(1/\varepsilon)/\varepsilon)}$. Hence, we get a QPTAS in this case.

Our results in this section are summarized as follows.

Theorem 4. *Let $\varepsilon > 0$ and let G be a weighted planar graph with n vertices. There is an algorithm running in time $n^{O(1/\varepsilon^2)}$ that outputs a $\{1, 2\}$ -ECSSM of G whose weight is at most $(1 + \varepsilon) \cdot \text{OPT}$.*

Theorem 5. *Let $\varepsilon > 0$ and let G be a weighted planar graph with n vertices. There is an algorithm running in time $n^{O(\log n \cdot \log(1/\varepsilon)/\varepsilon)}$ that outputs a $\{1, 2\}$ -ECSS H of G such that $w(H) \leq (1 + \varepsilon) \cdot \text{OPT}$.*

Theorem 6. *Let $\varepsilon > 0$, and let G be a weighted planar graph with n vertices. There is an algorithm running in time $n^{O(\log n \cdot \log(1/\varepsilon)/\varepsilon)}$ that outputs a $\{1, 2\}$ -VCSS H of G such that $w(H) \leq (1 + \varepsilon) \cdot \text{OPT}$.*

References

1. I. Althöfer, G. Das, D. Dobkin, D. Joseph, and J. Soares. On sparse spanners of weighted graphs. *Discrete Comput. Geom.*, 9: 81–100, 1993.
2. S. Arora, M. Grigni, D. Karger, P. Klein, and A. Woloszyn. A polynomial time approximation scheme for weighted planar graph TSP. *SODA* 1998, pp. 33–41.
3. A. Berger, M. Grigni and H. Zhao. A well-connected separator for planar graphs. *Manuscript*, 2004.

4. J. Cheriyan, S. Vempala and A. Vetta. An approximation algorithm for the minimum-size k -vertex connected subgraph. *SIAM J. Comput.*, 32(4):1050–1055, 2003.
5. A. Czumaj, M. Grigni, P. Sissokho, and H. Zhao. Approximation schemes for minimum 2-edge-connected and biconnected subgraphs in planar graphs. *SODA 2004*, pp. 489–498.
6. A. Czumaj and A. Lingas. On approximability of the minimum-cost k -connected spanning subgraph problem. *SODA 1999*, pp. 281–290.
7. A. Czumaj and A. Lingas. Fast approximation schemes for Euclidean multi-connectivity problems. *ICALP 2000*, pp. 856–868.
8. A. Czumaj, A. Lingas, and H. Zhao. Polynomial-time approximation schemes for the Euclidean survivable network design problem. *ICALP 2002*, pp. 973–984.
9. L. Fleischer. A 2-approximation for minimum cost $\{0, 1, 2\}$ vertex connectivity. *IPCO 2001*, pp. 115–129.
10. H. N. Gabow. An ear decomposition approach to approximating the smallest 3-edge connected spanning subgraph of a multigraph. *SODA 2002*, pp. 84–93.
11. H. N. Gabow. Better performance bounds for finding the smallest k -edge connected spanning subgraph of a multigraph. *SODA 2003*, pp. 460–469.
12. M. Grötschel, C. L. Monma, and M. Stoer. Design of survivable networks. In M. O. Ball et al., eds., *Handbooks in Operations Research and Management Science*, vol 7: *Network Models*, chapter 10, pp. 617–672. North-Holland, Amsterdam, 1995.
13. K. Jain. A factor 2 approximation algorithm for the generalized Steiner network problem. *Combinatorica*, 21(1):39–60, 2001.
14. R. Jothi, B. Raghavachari, and S. Varadarajan. A $5/4$ -approximation algorithm for minimum 2-edge-connectivity. *SODA 2003*, pp. 725–734.
15. S. Khuller. Approximation algorithms for finding highly connected subgraphs. In D. S. Hochbaum, ed., *Approximation Algorithms for \mathcal{NP} -Hard Problems*, pp. 236–265, 1996.
16. S. Khuller and U. Vishkin. Biconnectivity approximations and graph carvings. *Journal of the ACM*, 41(2):214–235, March 1994.
17. G. Kortsarz and Z. Nutov. Approximation algorithm for k -node connected subgraphs via critical graphs. *STOC 2004*, pp. 138–145.
18. P. Krysta. Approximating minimum size 1,2-connected networks, *Discrete Appl. Math.*, 125:267–288, 2003.
19. C. Levcopoulos, G. Narasimhan, and M. H. M. Smid. Efficient algorithms for constructing fault-tolerant geometric spanners. *STOC 1998*, pp. 186–195.
20. M. Penn and H. Shasha-Krupnik. Improved approximation algorithms for weighted 2- and 3-vertex connectivity augmentation problems. *J. Algorithms*, 22:187–196, 1997.
21. S. B. Rao and W. D. Smith. Approximating geometrical graphs via “spanners” and “banyans”. *STOC 1998*, pp. 540–550.
22. M. Stoer. *Design of Survivable Networks. Lect. Notes in Math.* 1531, Springer-Verlag, 1992.
23. S. Vempala and A. Vetta. Factor $4/3$ approximations for minimum 2-connected subgraphs. *APPROX 2000*, pp. 262–273.

Packet Routing and Information Gathering in Lines, Rings and Trees

Yossi Azar^{1,*} and Rafi Zachut¹

School of Computer Science, Tel Aviv University,
Tel Aviv, 69978, Israel
{azar, zachutra}@tau.ac.il

Abstract. We study the problem of online packet routing and information gathering in lines, rings and trees. A network consist of n nodes. At each node a buffer of size B . Each buffer can transmit one packet to the next buffer at each time step. The packets injection is under adversarial control. Packets arriving at a full buffer must be discarded. In information gathering all packets have the same destination. If a packet reaches the destination it is absorbed. The goal is to maximize the number of absorbed packets. Previous studies have shown that even on the line topology this problem is difficult to handle by online algorithms. A lower bound of $\Omega(\sqrt{n})$ on the competitiveness of the Greedy algorithm was presented by Aiello et al in [1]. All other known algorithms have a near linear competitive ratio. In this paper we give the first $O(\log n)$ competitive deterministic algorithm for the information gathering problem in lines, rings and trees. We also consider multi-destination routing where the destination of a packet may be any node. For lines and rings we show an $O(\log^2 n)$ competitive randomized algorithms. Both for information gathering and for the multi-destination routing our results improve exponentially the previous results.

1 Introduction

Overview: Packet routing networks, have become dominant platform for carrying data. In this paper we investigate a packet routing and information gathering in lines, rings and trees. In information gathering all injected packets have the same destination. Information gathering is widely used in many networks (e.g sensor networks). We also consider the multi destination routing in which the destination of a packet might be any node in the network.

We model the problem of packet routing on a unidirectional line or ring or tree as follows. A network has n nodes. At each node a buffer of size B . At each time unit, new packets may arrive at the buffers, each has a destination node. A packet can only be stored in a buffer if there is enough space. Since the n nodes have bounded capacity, packet loss may occur. All packets have the same value. At each time step a buffer can transmit one packet to its successor.

* Research supported in part by the German-Israeli Foundation and by the Israel Science Foundation.

When a packet reaches its destination it is absorbed. The goal is to maximize the number of absorbed packet. The definitions can be extended to rings, trees and general graphs.

Traditionally, the performance of a packet routing algorithm was measured within the stability analysis. In such framework either a probabilistic model ([8,11]) or an adversarial model([2,9]) for the packet injection is given, and the goal is to bound the buffer size needed to prevent packet drop. Since it seems impossible to avoid packet drop in practice, approximation analysis which avoid any a priori assumption on the input and compares the performance of algorithms to the optimal solution in the context of throughput has been adopted recently. In particular competitive analysis in which one has to deal with dropped packets becomes a common approach ([1,7]).

To the best of our knowledge, even for a simple topology as the line, all known algorithm either have a near linear competitive ratio or they must use buffers which are much larger than those of the optimal solution, in order to obtain a good competitive ratio. In [1] Aiello *et al.* showed the poor performance of the greedy algorithm for information gathering by proving a lower bound of $\Omega(\sqrt{n})$ on its competitive ratio (actually the upper bound is $n+1$ for FIFO model). They also showed that for the multi-destination routing, algorithm *NTG* (Nearest To Go) is only $O(n^{\frac{2}{3}})$ -competitive. There were no algorithms with poly-logarithmic competitive ratio given the same condition as the optimal solution even for lines, rings and trees.

In this paper we provide the first logarithmic competitive algorithm for information gathering improving exponentially the previous results for lines, rings and trees. For multi destination routing we provide the first poly-logarithmic randomized algorithms for lines and rings. Our results hold even for small buffers of constant size as well as for large buffers independent of the buffer size.

Our Results:

- Our main contribution is an $O(\log n)$ competitive deterministic algorithms for information gathering in lines, rings and trees. For lines and rings we require $B \geq 3$ and for trees we require that B is larger than the maximum degree. We note that for $B = 1$ there is an easy deterministic lower bound of n for the line.
- We provide an $O(\log^2 n)$ competitive randomized algorithm for the multi-destination routing in lines and unidirectional and bidirectional rings topology for any $B \geq 3$.

We use two tools which are of their own interest:

- We present a generic technique to transform any *fractional* algorithm for information gathering into a discrete algorithm. Specifically, we show that given a fractional algorithm for information gathering with buffers of size $B \geq 3$ in a line, we can construct a discrete algorithm whose competitive ratio is larger by the small factor of $\frac{B}{B-2}$.
- We present a generic technique to construct a fractional algorithm for large buffers from an algorithm for smaller buffers. Specifically, we show that given

a fractional algorithm for information gathering for buffers of size n we can construct a fractional algorithm for buffers of size $B > n$ with competitive ratio larger by a factor of 16.

Recently, independently of our work Stanislav *et al.* [3] achieved a slightly weaker result of $O(\log^2 n)$ competitiveness for information gathering in lines and trees and a randomized $O(\log^3 n)$ algorithm for multi-destination routing in lines.

Our Techniques: We start by studying the online fractional call admission and circuit routing problem. By a small modification to the algorithm of Awerbuch *et al.* [4] for the discrete version of this problem with small bandwidth requests, we obtain a fractional $O(\log D)$ competitive algorithm, where D is a bound on the allowed maximum length of path used. We next construct an *online* reduction from the fractional buffers packet routing in a line network with bounded delay problem to the problem of fractional call admission and circuit routing. Thus we obtain an $O(\log(nB))$ competitive algorithm for the fractional information gathering, which immediately supplies an $O(\log n)$ competitive algorithm for buffers of size $B \leq n$. For larger buffers we use a reduction to buffers of size n in order to obtain the $O(\log n)$ competitiveness. Next we use a technique to transform any fractional algorithm for information gathering into a discrete algorithm, and obtain an $O(\log n)$ discrete algorithm for information gathering in a line network. We construct an $O(\log^2 n)$ randomized algorithm for the multi-destination routing in a line network problem using the "Classify and randomly Select" with the algorithm for information gathering. We extend our techniques for rings and trees.

Related Results for Throughput Packet Routing:

- **Line and Tree Topologies:** Aiello *et al.*[1] investigated the unit packet routing on the line topology and proved a lower bound of $\Omega(\sqrt{n})$ on the competitiveness of the greedy algorithm for information gathering in a line. Algorithm *NTG* (Nearest To Go) is shown in [1] to be $O(n^{\frac{2}{3}})$ competitive for the multi-destination routing in a line. Azar *et al.* [7] showed that the greedy algorithm for the multi-destination routing is $(n + 1)$ competitive. In [10] Kesselman *et al.* investigated the routing problem under the *work conserving* assumption on directed lines and directed trees where packets are injected at the leaves and are destined to the root.
- **General Graphs:** In [5] Awerbuch *et al.* presented a load balance algorithm for anycasting packet routing in general topologies. For the line problem this algorithm is $\frac{1}{1-\epsilon}$ competitive using buffers which are larger by factor of $O(\frac{n}{\epsilon})$ than those of the optimal solution. In [1] Aiello *et al.* proved that algorithm *NTG* (Nearest To Go) is $O(md)$ competitive for any network, where m is the number of edges in the network and d is the maximal length of a path traversed by any packet. They also showed that on DAGs any greedy algorithm is $O(md)$ competitive.

Other Related Work: For various switching model there are constant competitive algorithms for the throughput. There is a lot of work on adversarial

queuing theory where the adversary never overload the network. There is also a lot on online call admission and routing for circuit routing. We use [4] for our work.

2 Problem Definition and Notations

There are two major routing types in communication networks: packet routing and virtual circuit routing. In circuit routing paths connections are constructed while in packet routing packets are traversed in the network. In this paper we consider packet routing defined below. Interestingly, our results are based also on virtual circuit routing defined in section 3.

In the online Packet Routing problem on a line we have a network organized in a line topology of length n , i.e. node $i = 0, \dots, n - 1$ is connected to node $i + 1$ via a unidirectional link with unit capacity. Node $i = 0, \dots, n - 1$ contains a buffer of size B , which is initially empty, to buffer the packets waiting to be transmitted via its outgoing link. In information gathering we may assume without loss of generality that node $n - 1$ is the destination of all packets while in the multi-destination routing the destination of a packet may be any node. We assume time proceeds in discrete steps, and each time step $t \geq 0$ is divided into two phases: at the first phase new packets may arrive to nodes $i = 0, \dots, n - 1$, each packet is associated with a destination node. During the second phase of time t , node $i = 0, \dots, n - 1$ may transmit a packet from its buffer to node $i + 1$. If a packet reaches its destination it is absorbed. Otherwise, online arriving packets (from both phase 1 and phase 2) can be buffered without exceeding the buffers capacities. Remaining packets must be discarded. The goal is to maximize the number of packets that reach their destination. We consider the non-preemptive model, in which stored packets cannot be discarded from the buffers.

Given an online algorithm A we denote by $A(\sigma)$ the value of A given the sequence σ . We denote the optimal (offline) algorithm by OPT , and use similar notation for it.

3 Online Fractional Call Admission and Circuit Routing

Our routing algorithm for information gathering is based on an online fractional call admission and circuit routing algorithm. Thus we start by considering the online version of the call admission and circuit routing problem, which is defined as follows. A network is represented by a capacitated graph $G(V, E, u)$ and a bound D on the allowed maximum length of path used. The capacity $u(e)$ assigned to each edge $e \in E$ represents the bandwidth available on this edge. The online input sequence consists of a call requests for paths: $\beta_1, \beta_2, \dots, \beta_k$, where the i th call is represented by: $\beta_i = \{s_i, t_i, r_i\}$. Node s_i is the origin of the call β_i , node t_i is its destination, and r_i is the bandwidth it requires. Upon receiving a call request β_i an algorithm either routes it by assigning it a path of maximum length D from s_i to t_i with r_i bandwidth, or rejects it. If it routes the call then the available bandwidth of each edge on the path decreases by r_i and

the throughput of the algorithm increases by r_i . The goal of an algorithm is to maximize its throughput while maintaining the capacity constraints.

The fractional version of the problem is defined as follows: an algorithm can split a call into smaller bandwidth calls and treat each one of them as a separate call. Thus an algorithm can route different fractions of a call β_i in different paths from s_i to t_i , and reject the remaining fraction. The throughput of the algorithm is the total bandwidth of routed fractions.

Awerbuch *et al.* [4] investigated the integral version of the problem. They proved $O(\log D)$ competitive algorithm for the case that the bandwidth of each request is relatively small compared to the capacity of the edges. Specifically they assumed that for each call β_i , $r_i \leq \frac{\min_e \{u(e)\}}{\log(2D+2)}$. We call their algorithm *AAP*.

The low bandwidth assumption is required to achieve a poly-logarithmic competitive algorithm. We show how to easily overcome this assumption by modifying the *AAP* algorithm and allowing it to route fractional call. We call this fractional algorithm *FAAP* (presented in figure 1).

Algorithm *FAAP*

Upon arrival of the call β_i :

1. Split β_i into calls of bandwidth $\frac{\min_e \{u(e)\}}{\log(2D+2)}$. (The last fraction of β_i might be of smaller bandwidth).
2. Run *AAP* in sequence on β_i fractions , until it rejects a fraction or all fractions have been routed.

Fig. 1. Algorithm *FAAP*

Theorem 1. *Algorithm FAAP is $O(\log D)$ competitive for maximizing the throughput (even compared to a fractional OPT).*

4 Fractional Packet Routing

In this section we consider a fractional version of the packet routing problem described in section 2. I.e., we allow an algorithm to accept fractional packets as well as transmit fractional packets from one node to its successor. Each packet fraction that reaches its destination increases the algorithm throughput by its size. The purpose of this section is to construct an $O(\log n)$ competitive fractional algorithm for information gathering. We also assume that input sequence σ consists of *integral* packets. However, this restriction is not obligatory for this section. The restriction is relevant for the transformation of a fractional algorithm for the packet routing problem into a discrete routing algorithm (see subsection 5.1).

4.1 Fractional Packet Routing with Bounded Delay

In this section we consider a fractional variant of the multi-destination routing in which a packet fraction must not stay more than T time steps in the network.

We begin by introducing a translation of this problem into the problem of fractional call admission and circuit routing which was described in section 3. The graph $G = (V, E, u)$ for the fractional call admission problem is described in figure 2.

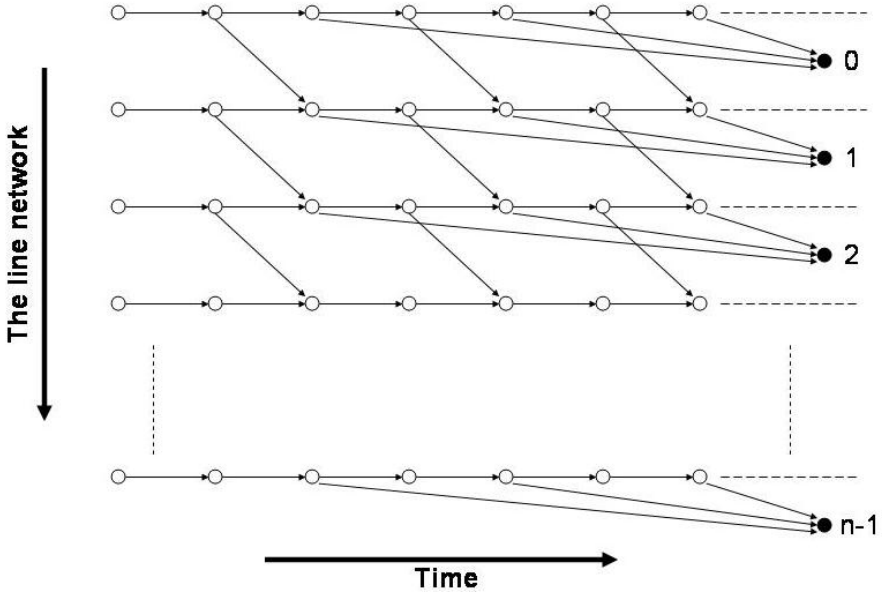


Fig. 2. The graph G for the call admission and circuit routing problem

Exploiting the matrix shape in which the white nodes in figure 2 are positioned, we refer to a white node according to its coordinates in the matrix. The node at the left top corner is node $\{0, 0\}$.

The i 'th row in the graph represents the i 'th buffer over all times. Each time step is represented by two consequent columns. The beginning of the arrival phase of time step t is represented by column number $2t$. The consequent column represents the beginning of the transmission phase of time step t . We assign a capacity B to each edge $(\{2t, i\}, \{2t + 1, i\})$ for every $t \geq 0$ and $0 \leq i \leq n - 1$. Those edges represent the buffers resources. We assign a capacity 1 to all diagonal edges between two white nodes. They represent the links resources. All other edges capacity is ∞ .

We set the bound D on the allowed maximum length of a path used in the translated fractional call admission and circuit routing problem to be $2T$ (since each time step corresponds to two consequent edges).

Based on the graph description, we translate the input sequence σ . Suppose a packet $p \in \sigma$ has arrived at node i at time step t and its destination is node j . We translate p to the following call request c on G : The origin of c is the white node $\{2t, i\}$. The destination of c is the black node j . The bandwidth c requires is 1.

It is not hard to show the equivalence between the fractional packet routing with bounded delay problem and the translated fractional call admission and circuit routing problem.

In figure 3 we present the online algorithm BPR for the fractional packet routing with bounded delay T .

Algorithm BPR (Bounded delay Packet Routing)

- Maintain a running simulation of $FAAP$ on G with $D = 2T$ on the translated input sequence σ .
- If $FAAP$ accepted a call fraction, accept its corresponding packet fraction and route it according to the path of the call fraction.

Fig. 3. Algorithm BPR

Theorem 2. *Algorithm BPR is feasible and $O(\log T)$ competitive for the fractional packet routing with bounded delay.*

Remark 1. The technique presented in subsection 4.1 can be generalized to reduce fractional packet routing in general graphs to circuit routing.

4.2 An $O(\log(nB))$ Algorithm for Fractional Information Gathering

In this subsection we consider the fractional version of information gathering with no bound on the delay. It can be shown that any feasible routing solution with unbounded delay can be transformed into a feasible solution with bounded delay $T = 2nB$ by losing only a constant factor in the throughput. This implies $O(\log(nB))$ competitive fractional algorithm PR for information gathering (presented in figure 4) based on the BPR algorithm.

Algorithm PR (Packet Routing)

Apply algorithm BPR with bounded delay $T = 2nB$.

Fig. 4. The PR algorithm

Theorem 3. *Algorithm PR is feasible and $O(\log(nB))$ competitive for fractional information gathering with no bounded delay assumptions.*

4.3 Reduction from Buffers of Size $B > n$ to Buffers of Size n

In this subsection we give a generic technique to construct a fractional algorithm for information gathering with large buffers from an algorithm for small buffers. Specifically, given a c -competitive algorithm for buffers of size n , we can construct a $16c$ -competitive fractional algorithm for buffers of size $B > n$. We call this technique *GR* (Generic Reduction). We assume throughout this subsection that $n|B$ otherwise we use only the biggest portion of the buffers space which is divisible by n .

Note that given a c -competitive algorithm for buffers of size n immediately implies that we are given a $2c$ -competitive fractional algorithm for buffers of size $\frac{n}{2}$ by halving each accepted/transmitted packet fraction. Furthermore, we can construct $2c$ -competitive algorithm A for buffers of size $\frac{B}{2}$ and links of bandwidth $\frac{B}{n}$ by scaling. We will show that applying the *GR* technique on algorithm A generates a $16c$ -competitive algorithm for information gathering with buffers of size $B > n$.

The following definition unite every $\frac{B}{n}$ consequent time steps.

Definition 1. We define the l 'th time interval ($l \geq 0$) as time steps $l \cdot \frac{B}{n}$ upto $(l + 1) \cdot \frac{B}{n} - 1$.

Definition 2. We define the l 'th border time as the time between the end of the l 'th time interval and the beginning of time interval $l + 1$.

We denote by $\hat{\sigma}$ the input sequence σ in which for each buffer we concatenate packets arriving during the same time interval. Informally, the idea of the technique is to simulate algorithm A which runs in time intervals on the sequence $\hat{\sigma}$, by transmitting in the original sequence σ during the time steps contained in the time interval. We denote by $R_{i,l}$ the quantity of the packet fractions which was injected to buffer i at time interval l and accepted by the simulation of algorithm A on $\hat{\sigma}$. We denote by $T_{i,l}$ the quantity of packet fractions which was transmitted from buffer i at time interval l by the simulation of algorithm A on $\hat{\sigma}$. In figure 5 we give the exact definition of *GR* with algorithm A as a parameter.

Theorem 4. Let $B' \leq B$ the largest number such that $n|B'$ and let A be a $2c$ -competitive algorithm for fractional information gathering with bandwidth $\frac{B'}{n}$ and buffers of size $\frac{B'}{2}$, then algorithm GR^A is $16c$ -competitive for fractional information gathering with buffers of size B (and links of bandwidth 1).

Corollary 1. Theorem 4 implies that given a c -competitive algorithm for buffers of size n , we can construct a $16c$ -competitive algorithm for fractional information gathering with buffers of size $B > n$ by applying GR^A as above.

4.4 Fractional Information Gathering – An $O(\log n)$ Competitive Algorithm

In this section we present an $O(\log n)$ competitive algorithm for fractional information gathering. For that purpose we use algorithm *PR* for information gathering, presented in subsection 4.2, and the reduction to information gathering

Algorithm GR^A

- Virtually partition each buffer of GR^A to *upper* buffer and *lower* buffer. Each of size $\frac{B}{2}$.
- Run a simulation of algorithm A in time intervals on the sequence $\hat{\sigma}$.
- For each node $i = 0, \dots, n - 1$ at every time step t :
 Let $l = \lfloor \frac{t}{\binom{t}{n}} \rfloor$.
 1. **Arrival phase:** Accept packet fractions into the upper buffer unless it's full.
 2. **Transmission phase:** Transmit $\frac{T_{i,l-1}}{\binom{t}{n}}$ fractions of packets from the lower buffer to the lower buffer of the next node.
- At border time l move $R_{i,l}$ fractions of packets from the upper buffer to the lower buffer.

Fig. 5. Algorithm GR^A

with size of buffers n presented in subsection 4.3. Algorithm FIG for fractional information gathering is presented in figure 6.

Algorithm FIG (Fractional Information Gathering)

- if $B \leq n$: Use algorithm PR .
- else
 1. Let A be algorithm PR for buffers of size $\frac{n}{2}$ scaled up by $\lfloor \frac{B}{n} \rfloor$.
 2. Use algorithm GR^A .

Fig. 6. Algorithm FIG

Theorem 5. *Algorithm FIG is feasible and $O(\log n)$ competitive for fractional information gathering.*

5 Discrete Information Gathering

In this section we consider discrete information gathering. Given a sequence σ which consists of *integral* packets, we present a generic local technique for buffers of size $B \geq 3$ to transform any online fractional algorithm for information gathering into a discrete algorithm with competitiveness multiplied by $\frac{B}{B-2}$. In particular, we use the fractional algorithm FIG presented in subsection 4.4, to construct a discrete algorithm with $O(\log n)$ competitiveness.

5.1 Discretization of a Fractional Packet Routing Algorithm

Given a sequence σ which consists of *integral* packets, we present a generic local technique to transform any fractional algorithm A for information gathering, that uses buffers of size B , into a discrete algorithm with the same throughput,

but uses buffers of size $B + 2$. This implies that, for $B \geq 3$ this technique can be used to transform any c -competitive fractional algorithm for the problem into a discrete algorithm with a competitive ratio of $c \cdot \frac{B}{B-2}$ which doesn't require additional buffer space. Before we proceed we introduce some notations. Given a sequence σ which consists of *integral* packets and a fractional (or discrete) algorithm Alg , we denote by $T_i^{Alg}(t)$ the accumulated sum of packet fractions Alg has transmitted from node i until time step t inclusive. We denote by $R_i^{Alg}(t)$ the accumulated sum of packet fractions that were injected from outside to node i until time step t inclusive and were accepted by Alg . In figure 7 we present the definition of RU with algorithm A as a parameter. This technique rounds up quantities from algorithm A . Let $A' = RU^A$.

Algorithm $A' = RU^A$

Run a simulation of algorithm A (fractional model) with the input sequence σ .
 For each node $i = 0, \dots, n - 1$ at every time step t :

1. **Arrival phase:** Accept packets until $R_i^{A'}(t) = \lceil R_i^A(t) \rceil$.
2. **Transmission phase:** Transmit a packet only if it is necessary to hold the equality : $T_i^{A'}(t) = \lceil T_i^A(t) \rceil$.

Fig. 7. Algorithm $A' = RU^A$

Theorem 6. *Suppose A' uses buffers larger than those used by A by two slots each. Then for every input sequence σ , A' is feasible and $A'(\sigma) \geq A(\sigma)$.*

Next we show how to use RU to transform any c -competitive fractional algorithm for buffers of size B into a discrete algorithm with buffers of size B whose competitive ratio is $c \cdot \frac{B}{B-2}$ for $B \geq 3$. We call this transformation D (Discretization). Let A be a c -competitive fractional algorithm for information gathering, when using buffers of size B . In figure 8 we present the definition of D with algorithm A as a parameter.

Algorithm D^A

1. Run a simulation of algorithm A with the input sequence σ .
2. Let S be the algorithm obtained by shrinking by $\frac{B-2}{B}$ each accepted/transmitted packet fraction of the simulation of A .
3. Apply RU^S .

Fig. 8. Algorithm D^A

Theorem 7. *Algorithm D^A uses buffers of size B and is $c \cdot \frac{B}{B-2}$ competitive for information gathering.*

5.2 An $O(\log n)$ Algorithm for Information Gathering

In this subsection we apply the discretization technique D from subsection 5.1 on algorithm FIG presented in subsection 4.4 in order to construct an $O(\log n)$ competitive discrete algorithm for information gathering. We construct the discrete algorithm IG for information gathering for buffers of size $B \geq 3$ by applying D^{FIG} .

Theorem 8. *Algorithm $IG = D^{FIG}$ is feasible and $O(\log n)$ competitive for information gathering (even against a fractional OPT).*

Remark 2. In information gathering, packets at a specific buffer are exchangeable. Thus transmitting the packet at the head of the buffer, instead of transmitting an arbitrary packet, will modify our information gathering algorithm to hold in a model in which FIFO queues are imposed.

6 Multi-destination Routing

In this section we give an $O(\log^2 n)$ randomized algorithm for multi-destination routing. For this algorithm we use the known technique "Classify and Randomly Select" (e.g [6]) and our result from subsection 5.2. We classify a packet p according to its source to destination distance d_p , and its source location s_p , by the following rule: let $l = \lfloor \log(d_p) \rfloor$ then classify p into class number $3l + \lfloor \frac{s_p}{2^l} \rfloor \bmod 3$. The above rule classifies the packets into $3 \log n$ disjoint classes.

Definition 3. *Nodes $j, \dots, j+3d-1$ are an interval of class number i if $d = 2^{\lfloor \frac{j}{3} \rfloor}$ and $j \bmod 3d = d(i \bmod 3)$.*

Definition 4. *A super exit of an interval of a class is the first node after the first third of the interval.*

In figure 9 we present algorithm MD for the multi-destination routing:

Algorithm MD (Multi-destination)

1. Randomly select a class number i^* uniformly in $\{0, 1, \dots, 3 \log n - 1\}$.
2. Reject all packets which are not from class i^* .
3. For each interval of the class i^* :
 - Run a simulation of IG in the first third of the interval while changing the destination of all the packets to be the super exit of the interval.
 - Apply the actions of the simulation in the first third of the interval.
 - Apply a greedy transmission in the last two-thirds of the interval.

Fig. 9. Algorithm MD

Theorem 9. *The randomized algorithm MD is $O(\log^2 n)$ competitive for the multi-destination routing.*

Remark 3. We can easily modify our multi-destination routing algorithm so that it uses FIFO queues with the same competitive ratio (see remark 2).

7 Extension to Trees and Rings

In this section we extend our results to rings and trees.

1. Information gathering in unidirectional ring is the same as in the line. For the multi-destination routing in a unidirectional ring we can also get $O(\log^2 n)$ competitive algorithm. This can be done in a similar way described in section 6 with the small modification that the definition of the interval should be considered under the modulo operation.
2. For the multi-destination routing on a bidirectional ring we can also get $O(\log^2 n)$ competitive algorithm. We pick one direction with probability of $\frac{1}{2}$ and apply the algorithm for the unidirectional case.
3. Information gathering in trees, i.e., routing packets to a single node of a tree, can be done in the same way described in section 4 and section 5. The discretization in the case of a tree results in the multiplication of the competitiveness of the fractional algorithm by a factor of $\frac{B}{B-d}$ (instead of $\frac{B}{B-2}$ as shown for the line topology), where d is the maximum input degree of a node. Thus for $B \geq (1 + \epsilon)d$ we obtain an $O(\log n)$ competitive algorithm. For example for binary tree ($d = 3$) we need $B \geq 4$.

References

1. W. Aiello, R. Ostrovsky, E. Kushilevitz, and A. Rosén. Dynamic routing on networks with fixed-size buffers. In *Proc. 14th SODA*, pages 771–780, 2003.
2. M. Andrews, B. Awerbuch, A. Fernández, J. Kleinberg, T. Leighton, and Z. Liu. Universal stability results for greedy contention-resolution protocols. In *Proc. 37th IEEE Symp. on Found. of Comp. Science*, pages 380–389, 1996.
3. S. Angelov, S. Khanna, and K. Kunal. The network as a storage device: Dynamic routing with bounded buffers. To appear in *APPROX*, 2005.
4. B. Awerbuch, Y. Azar, and S. Plotkin. Throughput competitive on-line routing. In *Proc. 34th IEEE Symp. on Found. of Comp. Science*, pages 32–40, 1993.
5. B. Awerbuch, A. Brinkmann, and C. Scheideler. Anycasting and multicasting in adversarial systems: Routing and admission control. In *Proc. 30 ICALP*, pages 1153–1168, 2003.
6. Baruch Awerbuch, Yair Bartal, Amos Fiat, and Adi Rosén. Competitive non-preemptive call control. In *Proc. 5th ACM-SIAM Symp. on Discrete Algorithms*, pages 312–320, 1994.
7. Y. Azar and Y. Richter. The zero-one principle for switching networks. In *Proc. 36th ACM Symp. on Theory of Computing*, pages 64–71, 2004.
8. A. Birman, H. R. Gail, S. L. Hantler, Z. Rosberg, and M. Sidi. An optimal service policy for buffer systems. *Journal of the Association Computing Machinery (JACM)*, 42(3):641–657, 1995.
9. A. Borodin, J. Kleinberg, P. Raghavan, M. Sudan, and D. Williamson. Adversarial queuing theory. In *Proc. 28th ACM STOC*, pages 376–385, 1996.
10. A. Kesselman, Z. Lotker, Y. Mansour, and B. Patt-Shamir. Buffer overflows of merging streams. In *Proc. 11th Annual ESA*, pages 349–360, 2003.
11. M. May, J. C. Bolot, A. Jean-Marie, and C. Diot. Simple performance models of differentiated services for the internet. In *Proceedings of the IEEE INFOCOM 1999*, pages 1385–1394.

Jitter Regulation for Multiple Streams

(Extended Abstract)

David Hay and Gabriel Scalosub

Computer Science Department, Technion,
Technion City, Haifa 32000, Israel
{hdavid, gabriels}@cs.technion.ac.il

Abstract. For widely-used interactive communication, it is essential that traffic is kept as smooth as possible; the smoothness of a traffic is typically captured by its *delay jitter*, i.e., the difference between the maximal and minimal end-to-end delays. The task of minimizing the jitter is done by jitter regulators that use a limited-size buffer in order to shape the traffic. In many real-life situations regulators must handle multiple streams simultaneously and provide low jitter on each of them separately. This paper investigates the problem of minimizing jitter in such an environment, using a fixed-size buffer.

We show that the offline version of the problem can be solved in polynomial time, by introducing an efficient offline algorithm that finds a release schedule with optimal jitter. When regulating M streams in the online setting, we take a *competitive analysis* point of view and note that previous results in [1] can be extended to an online algorithm that uses a buffer of size $2MB$ and obtains the optimal jitter possible with a buffer of size B . The question arises whether such a resource augmentation is essential. We answer this question in the affirmative, by proving a lower bound that is tight up to a factor of 2, thus showing that jitter regulation does not scale well as the number of streams increases unless the buffer is sized-up proportionally.

1 Introduction

Contemporary network applications call for connections with stringent Quality-of-Service (QoS) demands. This gives rise to QoS networks that are able to provide guarantees on various parameters, such as the end-to-end delay, loss ratio, bandwidth, and jitter. The need for efficient mechanisms to provide smooth and continuous traffic is mostly motivated by the increasing popularity of interactive communication and in particular video/audio streaming. The smoothness of the traffic is captured by the notion of *delay jitter* (or *Cell Delay Variation* [2]); namely, the difference between the maximal and minimal end-to-end delays of different fixed-size packets, henceforth referred to as *cells*.

Controlling traffic distortions within the network, and in particular jitter control, has the effect of moderating the traffic throughout the network [3]. This is important when a service provider in a QoS network must meet service level

agreements (SLA) with its customers. In such cases, moderating high congestion states in switches along the network results in the provider's ability to satisfy the guarantees to all its customers [4].

Jitter control mechanisms have been extensively studied in recent years (see a survey in [3]). These are usually modelled as *jitter regulators* [1,5,6] that use internal buffers in order to shape the traffic, so that cells leave the regulator in the most periodic manner possible. Generally, such regulators calculate a hypothetical periodic schedule, and try to release cells accordingly. Upon arrival, cells are stored in the buffer until their planned release time, or until a buffer overflow occurs. This indicates a tradeoff between the buffer size and the best attainable jitter, i.e., as buffer space increases, one can expect to obtain a lower jitter.

This paper investigates the problem of finding an optimal jitter release schedule, given a predetermined buffer size. This problem was first raised by Mansour and Patt-Shamir [1], who considered only a single-stream setting. However, in practice jitter regulators handle multiple streams simultaneously and must provide low jitter for each stream separately and independently.

In the *multi-stream* model, the traffic arriving at the regulator is an interleaving of M streams originating from M independent abstract sources (see Figure 1). Each abstract source i sends a stream of fixed-size cells in a fully periodic manner, with inter-release time X^i , which arrive at a jitter regulator after traversing the network. Variable end-to-end delay caused by transient congestion throughout the network may result in such a stream arriving at the regulator in a non-periodic fashion. The regulator knows the value of X^i , and strives to release consecutive cells X^i time units apart, thus re-shaping the traffic into its original form. Furthermore, the order in which cells are released by each abstract source is assumed to be respected throughout the network. This implies that the cells from the same stream arrive at the regulator in order (but not necessarily equally spaced), and the regulator should also maintain this order. We refer to this property as the *FIFO constraint*.

Note that the FIFO constraint should be respected in each stream independently, but not necessarily on all incoming traffic. This implies that in the multi-stream model, the order in which cells are released is not known *a priori*. This lack of knowledge is an inherent difference from the case where there is only one abstract source, and it poses a major difficulty in devising algorithms for multi-stream jitter regulation (as we describe in detail in Section 4).

Our Results

This paper presents algorithms and tight lower bounds for jitter regulation in this multiple-streams environment, both in offline and online settings. This answers a primary question posed in [1].

We evaluate the performance of a regulator in the multi-stream environment by considering the maximum jitter obtained on any stream. We show that surprisingly, the offline problem can be solved in polynomial time. This is done by characterizing a collection of optimal schedules, and showing that their prop-

erties can be used to devise an offline algorithm that efficiently finds a release schedule that attains the optimal jitter.

We use a *competitive analysis* [7,8] approach in order to examine the online problem. In this setting, by sizing up the buffer to a size of $2MB$ and statically partitioning the buffer equally among the M streams, applying the algorithm described in [1–Algorithm B] on each stream separately yields an algorithm that obtains the optimal max-jitter possible with a buffer of size B . We show that such a resource augmentation cannot be avoided, by proving that any online algorithm needs a buffer of size at least $M(B - 1) + B + 1 = \Omega(MB)$ in order to obtain the optimal jitter possible with a buffer of size B . We further show that these tight results also apply when the objective is to minimize the *average* jitter attained by the M streams. These results indicate that online jitter regulation does not scale well as the number of streams increases unless the buffer is sized up proportionally.

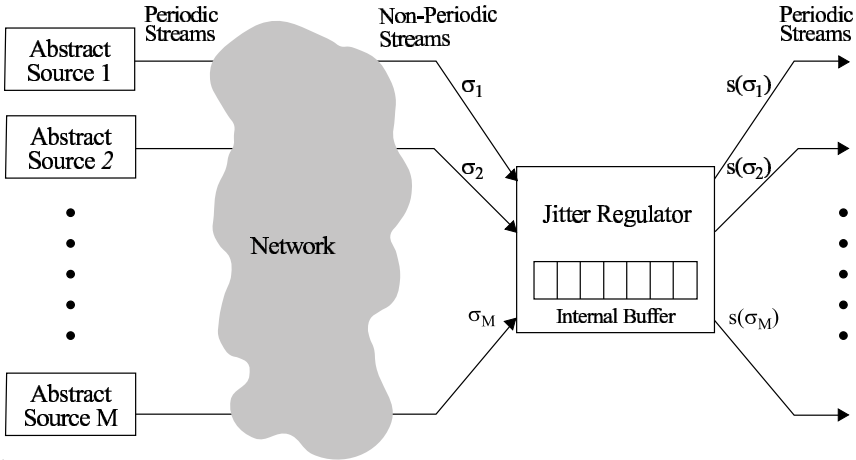


Fig. 1. The multi-stream jitter regulation model

Previous Work

Mansour and Patt-Shamir [1] consider a simplified *single-stream* model in which there is only a single abstract source. They present an efficient offline algorithm, which computes an optimal release schedule in these settings. They further devise an online algorithm, which uses a buffer of size $2B$, and produces a release schedule with the optimal jitter attainable with buffer of size B , and then show a matching lower bound on the amount of resource augmentation needed, proving that their online algorithm is optimal in this sense.

This model is later discussed by Koga [9] that deals with jitter regulation of a single stream with delay consideration. An optimal offline algorithm, and a nearly optimal online algorithm are presented for the case where a cell cannot be stored in the buffer for more than a predetermined amount of time.

2 Model Description, Notation, and Terminology

We adopt the following definitions from [1]:

Definition 1. Given a sequence of cells $\sigma = (p_i^\sigma)_{i=0}^n$ and a non-decreasing arrival function $a : \sigma \rightarrow \mathbb{R}^+$ such that cell p_i^σ arrives at time $a(p_i^\sigma)$:

1. A release schedule for σ is a function $s : \sigma \rightarrow \mathbb{R}^+$ satisfying for every $p_i^\sigma \in \sigma$, $a(p_i^\sigma) \leq s(p_i^\sigma)$.
2. A release schedule s for σ is B -feasible if at any time t ,

$$|\{p_i^\sigma \in \sigma | a(p_i^\sigma) \leq t < s(p_i^\sigma)\}| \leq B.$$

That is, there are never more than B cells in the buffer simultaneously.

3. The delay jitter of σ under a release schedule s is

$$J^\sigma(s) = \max_{0 \leq i, k \leq n} \{s(p_i^\sigma) - s(p_k^\sigma) - (i - k)X\}$$

where X is the inter-release time of σ (i.e., X is the difference between the release times of any two consecutive cells from the abstract source).¹

We first extend Definition 1 to an arrival sequence σ that is an interleaving of M streams $\sigma_1, \dots, \sigma_M$. We denote by X^{σ_i} the inter-release time of stream σ_i , and assume for simplicity that all streams have the same inter-release time X ; all our results extend immediately to the case where this does not hold. Let p_j^σ denote the j 'th cell (in order of arrival) of the interleaving of the streams σ , and let $p_j^{\sigma_i}$ denote the j 'th cell of the single stream σ_i . A release schedule should obey a per-stream FIFO discipline, in which cells of the same stream are released in the order of their arrival.

Let $J^{\sigma_i}(s)$ be the jitter of a single stream σ_i obtained by a release schedule s . We use the following metric to evaluate multi-stream release schedules:

Definition 2. The max-jitter of a multi-stream sequence $\sigma = \{\sigma_1, \dots, \sigma_M\}$ obtained by a release schedule s is the maximal jitter obtained by any of the streams composing the sequence; that is, $MJ^\sigma(s) = \max_{1 \leq k \leq M} J^{\sigma_k}(s)$.

2.1 Geometric Intuition

One can take a geometric view of delay jitter by considering a two dimensional plane where the x -axis denotes time and the y -axis denotes the cell number. We first consider the case of a single stream σ . Given a release schedule s , a point at coordinates (t, j) is marked if $s(p_j^\sigma) = t$ (see Figure 2(a)). The *release band* is the band with slope $1/X$ that encloses all the marked points and has minimal width. The jitter obtained by s is the width of its release band, and therefore our objective is to find a schedule with the narrowest release band.

Under the multi-stream model, we associate every stream σ_i with a different color i . A point at coordinates (t, j) is colored with color i if $s(p_j^{\sigma_i}) = t$. Any schedule s induces a separate release band for each stream σ_i in σ that encloses all points with color i . Schedule s is therefore characterized by M release bands.

¹ Since the abstract source generates perfectly periodic traffic, this definition of delay jitter coincides with the notion of Cell Delay Variation.

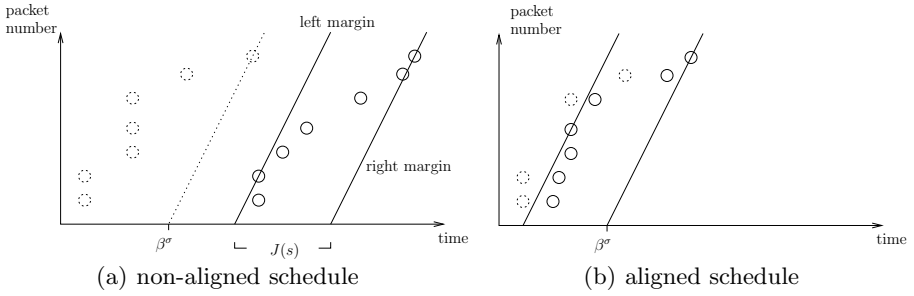


Fig. 2. Outline of arrivals (dotted circles) and marked releases (full circles)

3 Online Multi-stream Max-Jitter Control

As mentioned previously, there exists an online algorithm with buffer size $2MB$, which obtains the optimal max-jitter possible with a buffer of size B . In this section we show that this result is tight up to a factor of 2, by showing that in order to obtain the optimal max-jitter possible with a buffer of size B , any online algorithm needs a buffer of size at least $M(B - 1) + B + 1$. Hence, in order to maintain the same jitter performance, it is necessary to increase the buffer size in a linear proportion to the number of streams.

Theorem 1. *For every online algorithm ALG with an internal buffer of size $< M(B - 1) + B + 1$, there exists an arrival sequence consisting of M streams, such that ALG attains max-jitter strictly greater than the optimal jitter possible with a buffer of size B .*

Proof. Let ALG be an online algorithm with a buffer of size at most $M(B - 1) + B$. Consider the following arrival sequence σ : For every $0 \leq i \leq B - 1$, M cells arrive at the regulator at time $i \cdot X$, one for every stream. The sequence stops if ALG releases a cell before time $t' = (B + 1)X$.

If ALG releases a cell before time t' , say of stream σ_i , consider the following continuation for σ : In time $T > t'$ which can be arbitrarily large, one cell of stream σ_i arrives at the regulator.

Since ALG releases the first cell of stream σ_i before time t' , and the last cell of stream σ_i cannot be sent prior to time T , then $j^{\sigma_i}(\text{ALG}) \geq T - t' - (B + 1)X \geq T - (B + 1)X - (B + 1)X = T$, which can be arbitrarily large. It follows that $\text{MJ}^\sigma(\text{ALG})$ is strictly greater than zero. On the other hand, note that for any choice of T , the optimal max-jitter possible with a buffer of size B is zero: Every cell of a stream other than σ_i is released immediately upon its arrival, and for every $0 \leq j \leq B$, cell $p_j^{\sigma_i}$ is released in time $T - (B - j)X$. Since every stream other than σ_i does not consume any buffer space, it is easy to verify that at every time t , there are at most B cells in the buffer. Clearly, every stream obtains a zero jitter by this release schedule.

Assume now that ALG does not release any cells before time t' , implying that in time t' there are MB cells in the buffer. Consider the following continuation for σ : In time t' , $B + 1$ cells of stream σ_1 arrive at the regulator.

Since ALG has a buffer of size at most $M(B-1)+B = (M+1)B-M$, it must release at least $M+1$ cells in time t' . By the pigeonhole principle it follows that two of the released cells correspond to the same stream. This stream attains a jitter of at least $t'-t'-(0-1)X = X$, and therefore $MJ^\sigma(\text{ALG})$ is strictly greater than zero. On the other hand, the optimal max-jitter possible with a buffer of size B is zero: Every cell of a stream other than σ_1 is released immediately upon its arrival, and for every $0 \leq j \leq B$, cell $p_j^{\sigma_1}$ is released in time $t' - (B-j)X$. Similarly to the previous case, every stream obtains a zero jitter by this release schedule, and no more than B cells are stored simultaneously in the buffer. \square

Note that this lower bound for the case $M = 1$ exactly coincides with the result of the single stream model [1]. Theorem 1 further implies that in case the buffer size is $< M(B-1) + B + 1$, there are scenarios in which an optimal schedule attains zero jitter for all streams, while any online algorithm produces a schedule where at least one stream has a strictly positive jitter. This fact immediately implies that even if the objective is to minimize the average jitter obtained by the different streams, the same lower bound holds. Since the online algorithm, which statically partitions the buffer, minimizes the jitter of each stream independently, it clearly minimizes the overall average jitter as well, thus providing a matching upper bound.

4 An Efficient Offline Algorithm

This section presents an efficient offline algorithm that generates a release schedule with optimal max-jitter.

Given a sequence σ that is an interleaving of M streams, consider a total order $\pi = (p'_0, \dots, p'_n)$ on the release schedule of cells in σ that respects the FIFO order in each stream separately. The release schedule, which attains the optimal max-jitter and respects π , can be found using similar arguments to the ones in [1-Algorithm A]: Cell p'_j can be stored in the buffer only until cell p'_{j+B} arrives, imposing strict bounds on the release time of each cell. In particular, it follows that for every sequence σ , there exists an optimal release schedule. Unfortunately, it is computationally intractable to enumerate over all possible total orders, hence a more sophisticated approach should be considered.

We first discuss properties of schedules that achieve optimal max-jitter. We then show that these properties allow to find an optimal schedule in polynomial time, based solely on the cells' arrival times, and the parameters X and B .

For every cell p_j^σ , one can intuitively consider $t = a(p_j^\sigma) - jX$ as the time at which p_0^σ should be sent, so that p_j^σ is sent immediately upon its arrival, in a perfectly periodic release schedule. For any stream σ , denote by $\beta^\sigma = \max_j \{a(p_j^\sigma) - jX\}$. From a geometric point of view, β^σ is a lower bound on the intersection between the time axis and the right margin of any release band (see Figure 2(a)), since otherwise the cell defining β^{σ_i} would have to be released prior to its arrival.

Given a release schedule s for a sequence σ , a stream $\sigma_i \subseteq \sigma$ is said to be *aligned* in s if there is no cell $p_k^{\sigma_i} \in \sigma_i$ such that $s(p_k^{\sigma_i}) > \beta^{\sigma_i} + kX$. Clearly,

if σ_i is aligned in s , then the cell $p_j^{\sigma_i}$ that defines β^{σ_i} satisfies $s(p_j^{\sigma_i}) = a(p_j^{\sigma_i})$. Geometrically, the right margin of a release band corresponding to an aligned stream σ_i intersects the time axis in point $(\beta^{\sigma_i}, 0)$ (see Figure 2(b)).

A release schedule s for max-jitter is said to be *aligned*, if every stream is aligned in s . The following simple lemma shows that one can iteratively align the streams of an optimal schedule without increasing the overall jitter:

Lemma 1. *For every sequence σ , there exists an optimal aligned schedule s .*

Proof. Given an optimal schedule s' for sequence σ with at least ℓ aligned streams, we prove that s' can be changed into an aligned schedule (i.e. with M aligned streams), maintaining its optimality.

We first show that s' can be altered into an optimal schedule with $\ell + 1$ aligned streams. Let σ_i be one of the non-aligned streams in s' , and consider the following schedule \bar{s} :

$$\bar{s}(p_j^{\sigma_k}) = \begin{cases} \min \{s'(p_j^{\sigma_k}), \beta^{\sigma_k} + jX\} & k = i \\ s'(p_j^{\sigma_k}) & k \neq i \end{cases}$$

Clearly for every stream other than σ_i , the schedule remains unchanged, therefore it suffices to consider only stream σ_i . Since $s'(p_j^{\sigma_i}) \geq a(p_j^{\sigma_i})$ and $\beta^{\sigma_i} + jX \geq a(p_j^{\sigma_i})$, \bar{s} is a release schedule and it can easily be verified that \bar{s} satisfies the FIFO constraint. Schedule \bar{s} is B -feasible, since s' is B -feasible and for any cell $p_j^{\sigma_i}$, $\bar{s}(p_j^{\sigma_i}) \leq s'(p_j^{\sigma_i})$. Stream σ_i is aligned in \bar{s} , since clearly every cell $p_j^{\sigma_i}$ satisfies $\bar{s}(p_j^{\sigma_i}) \leq \beta^{\sigma_i} + jX$. Hence, \bar{s} has $\ell + 1$ aligned stream.

In order to prove that \bar{s} is optimal, it suffices to show that $\bar{s}(p_j^{\sigma_i}) - \bar{s}(p_m^{\sigma_i}) - (j - m)X \leq J^{\sigma_i}(s')$ for every two cells $p_j^{\sigma_i}, p_m^{\sigma_i} \in \sigma_i$. First note that $\bar{s}(p_j^{\sigma_i}) - \bar{s}(p_m^{\sigma_i}) - (j - m)X \leq s'(p_j^{\sigma_i}) - \bar{s}(p_m^{\sigma_i}) - (j - m)X$, since $\bar{s}(p_j^{\sigma_i}) \leq s'(p_j^{\sigma_i})$. If $\bar{s}(p_m^{\sigma_i}) = s'(p_m^{\sigma_i})$ then trivially $\bar{s}(p_j^{\sigma_i}) - \bar{s}(p_m^{\sigma_i}) - (j - m)X \leq J^{\sigma_i}(s')$. Otherwise, $\bar{s}(p_m^{\sigma_i}) = \beta^{\sigma_i} + mX = a(p_b^{\sigma_i}) - bX + mX$ for the cell $p_b^{\sigma_i}$ that defines β^{σ_i} . Since s' is a release schedule, then $s(p_b^{\sigma_i}) \geq a(p_b^{\sigma_i})$, which yields $\bar{s}(p_j^{\sigma_i}) - \bar{s}(p_m^{\sigma_i}) - (j - m)X \leq s'(p_j^{\sigma_i}) - s'(p_b^{\sigma_i}) - (j - b)X \leq J^{\sigma_i}(s')$.

Applying the same arguments repeatedly alters schedule s' into an aligned schedule and preserves its optimality. □

Next we show that the optimality of a schedule s is maintained even if cells that are stored in the buffer are released earlier, as long as their new release time satisfies FIFO order and remains within a release band of width $MJ^\sigma(s)$:

Lemma 2. *Let s be an optimal schedule for sequence σ . Then, for every stream $\sigma_i \subseteq \sigma$ and for every $J \in [J^{\sigma_i}(s), MJ^\sigma(s)]$, the new schedule*

$$s'(p_j^{\sigma_k}) = \begin{cases} \max \{a(p_j^{\sigma_k}), \beta^{\sigma_k} - J + jX\} & k = i \\ s(p_j^{\sigma_k}) & k \neq i \end{cases}$$

is B -feasible and $MJ^\sigma(s') = MJ^\sigma(s)$. Furthermore, if s is aligned then so is s' .

Proof. Since s' only changes the release schedule of stream σ_i , it clearly preserves the FIFO order and jitter of each stream other than σ_i .

We first show that s' respects the FIFO order of cells in σ_i . Let $p_j^{\sigma_i}$ be any cell in σ_i . If $s'(p_j^{\sigma_i}) = a(p_j^{\sigma_i})$ then its release time is $\leq a(p_{j+1}^{\sigma_i}) \leq s'(p_{j+1}^{\sigma_i})$. Otherwise, $s'(p_j^{\sigma_i}) = \beta^{\sigma_i} - J + jX \leq \beta^{\sigma_i} - J + (j+1)X \leq s'(p_{j+1}^{\sigma_i})$.

In order to bound the max-jitter of s' , it suffices to show that $J^{\sigma_i}(s') \leq \text{MJ}^\sigma(s)$. Consider any pair of cells $p_a^{\sigma_i}, p_b^{\sigma_i} \in \sigma_i$. By the definition of s' , $s'(p_a^{\sigma_i}) \geq \beta^{\sigma_i} - J + aX$. On the other hand, $s'(p_b^{\sigma_i}) = \max\{a(p_b^{\sigma_i}), \beta^{\sigma_i} - J + bX\} \leq \beta^{\sigma_i} + bX$ since $a(p_b^{\sigma_i}) \leq \beta^{\sigma_i} + bX$ by the definition of β^{σ_i} . Hence, $s'(p_b^{\sigma_i}) - s'(p_a^{\sigma_i}) \leq J + (b-a)X$, which implies that $J^{\sigma_i}(s') = \max_{a,b} \{s'(p_b^{\sigma_i}) - s'(p_a^{\sigma_i}) - (b-a)X\} \leq J \leq \text{MJ}^\sigma(s)$.

Assume by way of contradiction that s' is not B -feasible, and let t be any time in which a set P of more than B cells are stored in the buffer. Since the release schedule of any stream σ_k other than σ_i is identical under both s and s' , every cell $p_j^{\sigma_k} \in P$, for $k \neq i$, is also stored in the buffer at time t under schedule s . Note first that any cell in P is not released upon its arrival. Hence,

$$\begin{aligned}
 s'(p_j^{\sigma_i}) &= \beta^{\sigma_i} - J + jX && \text{by the definition of } s' \\
 &\leq \beta^{\sigma_i} - J^{\sigma_i}(s) + jX && \text{since } J \in [J^{\sigma_i}(s), \text{MJ}^\sigma(s)] \\
 &= a(p_k^{\sigma_i}) - kX - J^{\sigma_i}(s) + jX && \text{for } p_k^{\sigma_i} \text{ defining } \beta^{\sigma_i} \\
 &\leq s(p_k^{\sigma_i}) - (k-j)X - J^{\sigma_i}(s) && \text{since } a(p_k^{\sigma_i}) \leq s(p_k^{\sigma_i}) \\
 &\leq s(p_k^{\sigma_i}) - (k-j)X - \\
 &\quad (s(p_k^{\sigma_i}) - s(p_j^{\sigma_i}) - (k-j)X) && \text{by definition of } J^{\sigma_i}(s) \\
 &\leq s(p_j^{\sigma_i})
 \end{aligned}$$

Therefore, all cells $p_j^{\sigma_i} \in P$ are stored in the buffer at time t under schedule s as well, contradicting the B -feasibility of s .

We conclude the proof by showing that if s is aligned then s' is also aligned. Assume s is aligned. For any stream $\sigma_k \neq \sigma_i$ schedules s and s' are identical on σ_k , and therefore σ_k is aligned in s' . Assume by contradiction that σ_i is not aligned, therefore there is a cell $p_j^{\sigma_i}$ such that $s'(p_j^{\sigma_i}) > \beta^{\sigma_i} + jX$. Note that the definition of β^{σ_i} is independent of s and s' . By the definition of s' , $\max\{a(p_j^{\sigma_i}), \beta^{\sigma_i} - J + jX\} > \beta^{\sigma_i} + jX$. It follows that $a(p_j^{\sigma_i}) > \beta^{\sigma_i} + jX$, contradicting the maximality of β^{σ_i} . \square

The new schedule obtained in the above lemma is illustrated by the circled cells in Figure 3. By iteratively applying Lemma 2 with $J = \text{MJ}^\sigma(s)$ on all streams, we get:

Corollary 1. *Given an optimal aligned schedule s for sequence σ , the schedule defined by*

$$s'(p_j^{\sigma_k}) = \max\{a(p_j^{\sigma_k}), \beta^{\sigma_k} - \text{MJ}^\sigma(s) + jX\}$$

is an optimal aligned schedule.

The following lemma bounds from below the release time of cells in an aligned schedule. Intuitively, this lemma defines the left margin of the release band.

Lemma 3. *For any aligned schedule s for sequence σ , every stream $\sigma_i \subseteq \sigma$, and every cell $p_j^{\sigma_i}$, $s(p_j^{\sigma_i}) \geq \beta^{\sigma_i} - J^{\sigma_i}(s) + jX$.*

Proof. Assume by contradiction that there exists a stream σ_i and a cell $p_j^{\sigma_i}$ such that $s(p_j^{\sigma_i}) < \beta^{\sigma_i} - J^{\sigma_i}(s) + jX$. Let $p_k^{\sigma_i}$ be the cell defining β^{σ_i} . Since s is aligned, it follows that $s(p_k^{\sigma_i}) = a(p_k^{\sigma_i})$. Hence,

$$\begin{aligned} J^{\sigma_i}(s) &\geq s(p_k^{\sigma_i}) - s(p_j^{\sigma_i}) - (k - j)X \\ &> a(p_k^{\sigma_i}) - (\beta^{\sigma_i} - J^{\sigma_i}(s) + jX) - (k - j)X \\ &= a(p_k^{\sigma_i}) - (a(p_k^{\sigma_i}) - kX) + J^{\sigma_i}(s) - jX - kX + jX = J^{\sigma_i}(s), \end{aligned}$$

which is a contradiction. □

Lemma 3 indicates an important property of aligned optimal schedules. In such schedules, the jitter of any stream can be characterized by the release time of a single cell, as depicted in the following corollary: (proof omitted)

Corollary 2. *For any aligned schedule s for sequence σ and every stream $\sigma_i \subseteq \sigma$, $J^{\sigma_i}(s) = \max_j \{ \beta^{\sigma_i} - s(p_j^{\sigma_i}) + jX \}$.*

The following lemma shows that at least one of the widest release bands, corresponding to some stream σ_i attaining the max-jitter, has its left margin determined by the following event: An arrival of a cell causing a buffer overflow, which necessitates some cell of σ_i to be released earlier than desired.

Lemma 4. *Let s be an aligned optimal schedule for sequence σ . There exists a stream $\sigma_i \subseteq \sigma$ that attains the max-jitter, and a cell $p_j^{\sigma_i}$ such that $s(p_j^{\sigma_i}) = \beta^{\sigma_i} - MJ^\sigma(s) + jX$ and $s(p_j^{\sigma_i}) = a(p_\ell^\sigma)$ for some cell p_ℓ^σ .*

Proof. We show by contradiction that if the claim does not hold for an optimal aligned schedule, then such a schedule can be altered into a new schedule with max-jitter strictly less than the original schedule. Formally, consider an aligned optimal schedule s for σ . Let $M = \{ \sigma_i \mid J^{\sigma_i}(s) = MJ^\sigma(s) \}$, and for every $\sigma_i \in M$, let $T_i = \{ p_j^{\sigma_i} \mid s(p_j^{\sigma_i}) = \beta^{\sigma_i} - MJ^\sigma(s) + jX \}$. From a geometric point of view, T_i consists of all the cells in σ_i , whose release time lies on the left margin of σ_i 's release band. Finally, let $T = \bigcup_{\sigma_i \in M} T_i$. Assume by contradiction that for every $p_j^\sigma \in T$, there is no cell p_ℓ^σ such that $s(p_j^\sigma) = a(p_\ell^\sigma)$.

Note first that in such a case, $MJ^\sigma(s) > 0$. Otherwise, since s is aligned, for each stream σ_i the cell $p_k^{\sigma_i}$ defining β^{σ_i} satisfies both $s(p_k^{\sigma_i}) = a(p_k^{\sigma_i})$ and $s(p_k^{\sigma_i}) = \beta^{\sigma_i} - 0 + jX$.

The altered schedule s' is obtained by postponing the release of all the cells in T for some positive amount of time. As we shall prove, schedule s' is B -feasible, and has a max-jitter strictly less than $MJ^\sigma(s)$, contradicting the optimality of s .

For each cell $p_k^{\sigma_i} \in T$ which is the j 'th cell of σ (i.e., $p_k^{\sigma_i} = p_j^\sigma$), the exact amount of postponing time is determined by the following constraints:

1. *Avoiding buffer overflow:* Do not postpone further than the first arrival of a cell after $s(p_j^\sigma)$. This constraint is captured by

$$\delta(p_j^\sigma) = \min_{p_\ell^\sigma : a(p_\ell^\sigma) > s(p_j^\sigma)} \{ a(p_\ell^\sigma) - s(p_j^\sigma) \}.$$

2. *Maintaining FIFO order:* Do not postpone further than $s(p_{k+1}^{\sigma_i})$. This constraint is captured by $\varepsilon(p_j^\sigma) = s(p_{k+1}^{\sigma_i}) - s(p_k^{\sigma_i})$.

If p_j^σ is the last cell in σ , $\delta(p_j^\sigma) = \varepsilon(p_j^\sigma) = \infty$. Let $\delta = \min_{p_j^\sigma \in T} \delta(p_j^\sigma)$ and $\varepsilon = \min_{p_j^\sigma \in T} \varepsilon(p_j^\sigma)$, capturing the amounts of time that satisfy these constraints for all cells in T . Since $MJ^\sigma(s) > 0$ and by using the previous lemmas and the assumption, it can be verified that both δ and ε are finite and strictly greater than zero.

For the purpose of analysis, define for every stream $\sigma_i \in M$,

$$\rho(\sigma_i) = \min_{p_k^{\sigma_i} \in \sigma_i \setminus T_i} \{s(p_k^{\sigma_i}) - (\beta^{\sigma_i} - MJ^\sigma(s) + kX)\}.$$

$\rho(\sigma_i)$ comes to capture how far is the rest of the stream from the left margin. Since for any $\sigma_i \in M$, $J^{\sigma_i}(s) > 0$, then $\sigma_i \setminus T_i$ is not empty and $\rho(\sigma_i) > 0$. Let $\rho = \min_{\sigma_i \in M} \rho(\sigma_i)$. It follows that $\rho > 0$.

Let $\Delta = \min\{\delta, \varepsilon, \rho\}$, and consider the following schedule that, as we shall prove, attains a jitter strictly smaller than $MJ^\sigma(s)$:

$$s'(p_j^\sigma) = \begin{cases} s(p_j^\sigma) + \Delta/2 & p_j^\sigma \in T \\ s(p_j^\sigma) & \text{otherwise} \end{cases}$$

We first prove that s' is B -feasible and maintains FIFO order. Assume by way of contradiction that s' is not B -feasible, and let t be the first time the number of cells in the buffer exceeds B . By the minimality of t , there exists a cell that arrives at time t . For every cell $p_j^\sigma \in T$, no cells arrive to the buffer in the interval $[s(p_j^\sigma), s(p_j^\sigma) + \Delta/2]$ because $\Delta \leq \delta(p_j^\sigma)$, implying that t is not in any such interval. But the definition of s' yields that the content of the buffer in such a time t is the same under schedules s and s' , thus contradicting the B -feasibility of s . The FIFO order of s' is maintained since $\Delta \leq \varepsilon(p_j^\sigma)$ for every $p_j^\sigma \in T$.

We conclude the proof by showing that $MJ^\sigma(s') < MJ^\sigma(s)$. Consider any $\sigma_i \in M$, and any $p_k^{\sigma_i}$. If $p_k^{\sigma_i} \in T$ then by the definition of s' and Lemma 3, $s'(p_k^{\sigma_i}) = s(p_k^{\sigma_i}) + \Delta/2 \geq \beta^{\sigma_i} - MJ^\sigma(s) + kX + \Delta/2$. The same holds also for $p_k^{\sigma_i} \notin T$: Since $\rho(\sigma_i) \geq \Delta > \Delta/2$, it follows that $s'(p_k^{\sigma_i}) = s(p_k^{\sigma_i}) \geq \beta^{\sigma_i} - MJ^\sigma(s) + kX + \rho(\sigma_i) > \beta^{\sigma_i} - MJ^\sigma(s) + kX + \Delta/2$. Hence, for every $p_k^{\sigma_i}$,

$$\begin{aligned} \beta^{\sigma_i} - s'(p_k^{\sigma_i}) + kX &\leq \beta^{\sigma_i} - (\beta^{\sigma_i} - MJ^\sigma(s) + kX + \Delta/2) + kX \\ &= MJ^\sigma(s) - \Delta/2 \\ &< J^{\sigma_i}(s). \end{aligned}$$

By Corollary 2, $J^{\sigma_i}(s') < J^{\sigma_i}(s)$ for any stream $\sigma_i \in M$. The jitter of any other stream remains unchanged, therefore $MJ^\sigma(s') < MJ^\sigma(s)$, contradicting the optimality of s . \square

Lemma 4 implies that there is an optimal schedule s and a stream σ_i , such that $MJ(s) = \beta^{\sigma_i} - a(p_l^\sigma) + kX$, for some cells $p_k^{\sigma_i} \in \sigma_i$ and $p_l^\sigma \in \sigma$. Note that for any stream σ_i , the value of β^{σ_i} can be computed in linear time using only

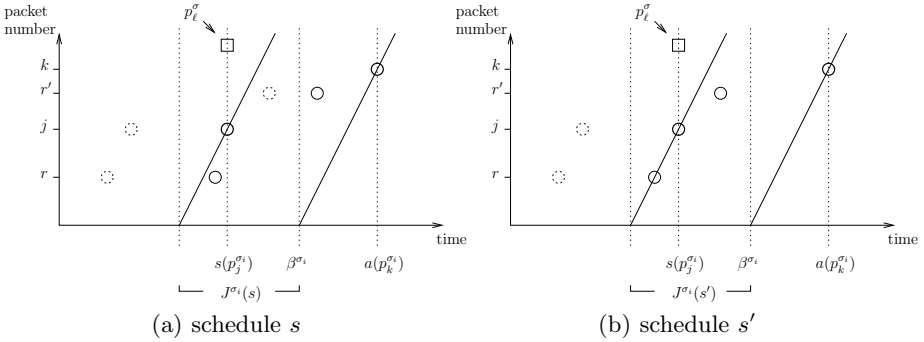


Fig. 3. Outline of arrivals (dotted circles) and releases (full circles) for cells of the stream σ_i that attains the max-jitter, in an aligned release schedule, as discussed in Corollary 1 and in Lemma 4. The square represents an arrival of some cell in σ causing buffer overflow.

the arrival sequence σ_i . It follows that by enumerating over all possible choices of the pair $(p_k^{\sigma_i}, p_l^{\sigma_i})$, one can find the collection of possible values of the optimal jitter. For every such value J , verifying that there is a B -feasible release schedule attaining jitter J can be done in linear time by checking the feasibility of the schedule defined in Corollary 1 assuming $MJ(s) = J$. This yields the following result:

Theorem 2. *There exists a polynomial-time algorithm that finds an optimal schedule for the multi-stream max-jitter problem.*

5 Discussion

This paper examines the problem of jitter regulation and specifically, the tradeoff between the buffer size available at the regulator and the optimal jitter attainable using such a buffer. We deal with the realistic case where the regulator must handle many streams concurrently, thus answering a primary question posed in [1].

We focus our attention on regulating the jitter of multiple streams with the objective of minimizing the maximum jitter attained by any of these streams. We show that the offline problem of finding a schedule that attains the optimal max-jitter can be solved in polynomial time, by a time-efficient algorithm which produces an optimal schedule. We observe that existing single-stream online algorithms can be used to devise an online algorithm for the multi-stream jitter regulation problem, at a cost of multiplying the buffer size linearly by the number of streams. We prove that such a resource augmentation is essential by providing an asymptotically matching lower bound. Our results for the online setting apply also to the problem of finding a release schedule with optimal average jitter.

Note that our offline algorithm suggests an interesting heuristic for improving the value of the jitter for an online algorithm using a buffer of size considerably less than MB , compared to the optimal jitter attainable with a buffer of size B .

One can calculate an optimal schedule of a prefix of the traffic using our offline algorithm, and then prolong the schedule by attempting to send consecutive cells as equally spaced as possible. Although there are traffics in which this approach fails, as shown by our lower bound, it may prove a useful heuristic in situations where the overall traffic in the network does not change radically over time.

Since real-life networks clearly have finite capacity links, it is also interesting to investigate the behavior of a jitter regulator that handles multiple streams simultaneously and its outgoing link is of bounded capacity. In addition, since regulators might be allowed to drop cells, it is of interest to examine the correlations between buffer size, optimal jitter, and drop ratio.

Acknowledgements. We would like to thank Hagit Attiya, Seffi Naor, Adi Rosén, and Shmuel Zaks for their useful comments.

References

1. Mansour, Y., Patt-Shamir, B.: Jitter control in Qos networks. *IEEE/ACM Transactions on Networking* **9** (2001) 492–502
2. The ATM Forum: Traffic Management Specification. (1999) Version 4.1, AF-TM-0121.000.
3. Zhang, H.: Service disciplines for guaranteed performance service in packet switching networks. *Proceedings of the IEEE* **83** (1995) 1374–1396
4. Tanenbaum, A.: *Computer Networks*. Fourth edn. Prentice Hall (2003)
5. Keshav, S.: *An Engineering Approach to Computer Networking*. Addison-Wesley Publishing Co. (1997)
6. Zhang, H., Ferrari, D.: Rate-controlled service disciplines. *Journal of High-Speed Networks* **3** (1994) 389–412
7. Borodin, A., El-Yaniv, R.: *Online Computation and Competitive Analysis*. Cambridge University Press (1998)
8. Sleator, D., Tarjan, R.: Amortized efficiency of list update and paging rules. *Communications of the ACM* **28** (1985) 202–208
9. Koga, H.: Jitter regulation in an internet router with delay constraint. *Journal of Scheduling* **4** (2001) 355–377

Efficient c -Oriented Range Searching with DOP-Trees

Mark de Berg^{*}, Herman Haverkort^{**}, and Micha Streppel^{***}

Department of Computer Science,
TU Eindhoven, P.O.Box 513, 5600 MB Eindhoven, The Netherlands
mberg@win.tue.nl
cs.herman@haverkort.net
m.w.a.streppel@tue.nl

Abstract. A c -DOP is a c -oriented convex polytope, that is, a convex polytope whose edges have orientations that come from a fixed set of c orientations. In this paper we study DOP-trees—bounding-volume hierarchies that use c -DOPs as bounding volumes—in the plane. We prove that for any set S of n disjoint c -DOPs in the plane, one can construct a DOP-tree such that a range query with a c -DOP as query range can be answered in $O(n^{1/2+\varepsilon} + k)$ time, where k is the number of reported answers. This is optimal up to the factor $O(n^\varepsilon)$. If the c -DOPs in S may intersect, the query time becomes $O(n^{1-1/c} + k)$, which is optimal.

1 Introduction

The range-searching problem is to preprocess a set S of objects in \mathbb{R}^d such that the objects intersecting a query range Q can be found quickly. The range-searching problem is a fundamental problem in computational geometry, and it arises in numerous applications in practice. Hence, it has been widely studied both from a theoretical and from an experimental point of view [2,3,8,10].

The goal of most of the research in computational geometry on range searching is to develop data structures with (close to) optimal performance guarantees for each specific setting. Thus there are data structures for range searching with triangles in point sets in the plane, for range searching with disks in point sets, for range searching with rectangles in line segments, and so on. Unfortunately, many of these data structures are rather involved. Moreover, it would be preferable to have a single *multi-functional geometric data structure*: a data structure

^{*} MdB was supported by the Netherlands' Organisation for Scientific Research (NWO) under project no. 639.023.301.

^{**} This research was done while HH was working at Karlsruhe University, supported by the European Commission, FET open project DELIS (IST-001907), and subsequently at the University of Aarhus, supported by a grant from the Danish National Science Research Council.

^{***} MS was supported by the Netherlands' Organisation for Scientific Research (NWO) under project no. 612.065.203.

that can store different types of data and answer various types of queries. Indeed, this is what is usually done in practice. The goal of our work is to investigate what can be said about such practical data structures from a theoretical point of view.

One of the most widely used practical solutions to the range-searching problem is to use a *bounding-volume hierarchy* (BVH). A BVH on a set S of objects is a tree structure whose leaves are in a one-to-one correspondence with the objects in S and where each node ν stores some bounding volume of the set of objects corresponding to the leaves in the subtree of ν . A BVH has linear size by definition, and it can store any kind of object. A query with a range Q is answered by traversing the BVH in a top-down manner, only proceeding to those nodes whose bounding volume is intersected by Q . For each leaf that is reached, the corresponding object needs to be checked against Q . In principle one can perform the search with any kind of range Q . To speed up the test whether the range Q intersects the bounding volume of some node, however, the range itself is often also replaced by a bounding volume. Hence, the possibly expensive test with the original range Q only has to be performed with the objects found at the leaf level. Note that there is a trade-off in the type of bounding volume used: a simple bounding volume will make the intersection test fast, but it often makes the bounding volume fit less well to the underlying objects so that more nodes in the tree will be visited.

A very popular bounding volume is the axis-aligned bounding box. The reason for this is, of course, that intersection tests between bounding boxes are very fast and easy to implement. BVHs that use bounding boxes as bounding volumes—such BVHs are called box-trees—have been investigated from a theoretical point of view by Agarwal *et al.* [1] and by Haverkort *et al.* [5]. Agarwal *et al.* showed how to construct a box-tree for a set S of n input boxes in \mathbb{R}^d such that a range query with a query box Q can be answered in time $\Theta(n^{1-1/d} + k)$, where k is the number of input boxes intersecting Q . They also showed that this is optimal in the worst case, even if the input boxes are disjoint. If the input bounding boxes are disjoint and $d = 2$, they present another box-tree, which achieves a query time of $O(\sqrt{n} \log n + k)$ for queries with axis-parallel rectangles, and $O(\log^2 n)$ for point queries. In Haverkort's thesis [4], the bound for rectangle queries is improved to the optimal $O(\sqrt{n} + k)$.

As noted above, a simple bounding volume such as a bounding box may not fit the underlying objects or the query range very well. Suppose, for instance, that we want to find all objects intersecting a query line segment. If the slope of the segment is close to 1, then its bounding box will fit badly, possibly increasing the number of visited nodes dramatically. Indeed, the box-trees of Agarwal *et al.* [1] (or any box-tree, for that matter) cannot give any (sublinear) worst-case guarantees for queries with non-rectangular ranges. The fact that bounding boxes may not always fit well, inspired research on BVHs that use different, more tightly fitting bounding volumes [3,11]. One of the bounding volumes that has been suggested are so-called *discretely oriented polytopes* [6,7]. These are polytopes whose facets have orientations that come from a fixed set of c orientations. We

call such a bounding volume a c -DOP. For instance, a bounding box in the plane is a 2-DOP, since its edges are horizontal and vertical. The larger the value of c , the better the c -DOP will fit the underlying object. We call a BVH that uses c -DOPs as bounding volumes a *DOP-tree*. As far as we know, there are no DOP-trees with worst-case performance guarantees. In this paper we develop such DOP-trees for range searching in the plane, where the query range is also a c -DOP. We call such queries *DOP queries*. We will also consider *point queries*, where the query range is a point. Point queries—sometimes also called *inverse range queries*—are interesting in their own right. Moreover, if point queries can be answered very efficiently, one may expect that range queries with small ranges can be answered efficiently as well.

Our Results. We describe a general strategy to construct a DOP-tree on a set S of c -DOPs in the plane from a c -oriented binary space partition (BSP) on a set of representative points for the c -DOPs. (A BSP for a set of points in the plane is a recursive partitioning of the plane using lines, until each subspace contains a single point. A BSP is c -oriented if its splitting lines come from a fixed set of c orientations.) This strategy is described and analyzed in Section 3.

To obtain an efficient DOP-tree, we thus need a BSP on a set of points in the plane that has a good query time for range searching with c -DOPs. Developing such a BSP is the topic of Section 2, where we present such a BSP whose query time is $O(n^{1/2+\varepsilon} + k)$.¹ This result generalizes to higher dimensions; the query time in \mathbb{R}^d is $O(n^{1-1/d+\varepsilon} + k)$. We remark that the same (in fact, slightly better) bounds can be obtained for more general queries, namely arbitrary simplices, using partition trees [9]. Our BSP structure is much simpler, however, and its time for point queries is $O(\log n)$, which is not the case for partition trees. Moreover, partition trees cannot serve as a basis for our general strategy for constructing DOP-trees.

By combining our BSP result with our general strategy, we obtain a DOP-tree on a set S of disjoint c -DOPs in the plane such that DOP queries can be answered in $O(n^{1/2+\varepsilon} + k)$ time. This is optimal up to the factor $O(n^\varepsilon)$: any BVH that uses convex bounding volumes has $\Omega(\sqrt{n})$ query time, even for point data and rectangle queries [1]. Our DOP-tree answers point queries in $O(\log^c n)$ time. Note that our query times are almost as good as those of Agarwal *et al.* [1]. The advantages of our structure are twofold: (i) it works on more tightly fitting bounding volumes, so that less bounding volumes will be reported whose underlying objects do not intersect the range, and (ii) there are performance guarantees for a larger class of query ranges, namely c -DOPs instead of rectangles.

We also prove that if the c -DOPs in S may intersect, then any DOP-tree must have $\Omega(n^{1-1/c})$ query time in the worst case, even for point queries where the query point does not lie in any of the input c -DOPs, and we describe a DOP-tree that matches this bound. Interestingly, this discrepancy between intersecting and disjoint input does not occur for rectangular input and rectangular ranges:

¹ More precisely stated, for any constant $\varepsilon > 0$ we can construct the BSP in such a way that the query time is $O(n^{1/2+\varepsilon} + k)$. The constant in the O -notation will depend on ε and c .

here one can obtain $O(\sqrt{n} + k)$ query time both for disjoint and for intersecting input [1] (which can also be seen by plugging $c = 2$ into our results.)

Finally, we consider the case where the input set S is “almost” disjoint, in the sense that no point in the plane is contained in more than σ c -DOPs. (The number σ is called the stabbing number of S .) We prove that if σ is a constant, then the same bounds can be obtained as for disjoint objects. Due to lack of space several of the proofs are omitted in this extended abstract.

Terminology. Let \mathcal{C} be a set of c non-parallel hyperplanes in \mathbb{R}^d . We say that two hyperplanes have the same orientation if they are parallel. Thus the set \mathcal{C} defines c orientations. We say that a hyperplane, or a $(d-1)$ -dimensional facet of a polytope, is c -oriented if it has one of the c orientations defined by \mathcal{C} . We call a convex polytope a c -DOP if all of its facets are c -oriented. Hence, a c -DOP has at most $2c$ facets. The set \mathcal{C} is fixed throughout the paper, and terms like c -DOPs, c -oriented, and so on, always refer to this set \mathcal{C} . Moreover, when we speak of a DOP, we always mean a c -DOP. Finally, we define $bdop(o)$ to be the bounding DOP of an object o , that is, the smallest c -DOP containing o .

2 A c -Oriented bsp for dop Queries

In this section we describe a BSP on a set S of n points in \mathbb{R}^d that has a good query time for range searching with c -DOPs.

A *simplicial partition of size r* for a set S of n points in \mathbb{R}^d is a set of pairs $\Psi(S) = \{(S_1, \Delta_1), \dots, (S_r, \Delta_r)\}$, where the S_i form a (disjoint) partition of S , each Δ_i is a simplex containing S_i , and $|S_i| \leq n/r$ for all i . The *crossing number* of $\Psi(S)$ is the maximum number of simplices intersecting any hyperplane. Matoušek [9] has shown that any set of points admits a simplicial partition of size r with crossing number $O(r^{1-1/d})$. We will obtain basically the same result, except that we only bound the crossing number with respect to c -oriented hyperplanes. Thus our result is less general than the result of Matoušek. Our construction, however, has one important additional property, which will be crucial in later sections: it is a BSP. The simplices in Matoušek’s partitioning, on the other hand, are arbitrary and can even intersect. This means that the crossing number of a point—the maximum number of simplices containing the point—is 1 for our partitioning, whereas no better bounds than $O(r^{1-1/d})$ are known for Matoušek’s partitioning. Moreover, our partitioning algorithm is much simpler.

Lemma 1. *Let S be a set of n points in \mathbb{R}^d . For any $r \leq n$ there exists a c -oriented BSP for S with the following properties:*

- (i) *each cell in the partitioning contains at most n/r points,*
- (ii) *the size of the partitioning is $O(r)$,*
- (iii) *any c -oriented hyperplane h intersects $O(r^{1-1/d})$ cells.*

The partitioning can be constructed in $O(n \log n)$ time.

Proof. Consider an orientation in \mathcal{C} . Take a set of $r^{1/d} - 1$ splitting hyperplanes with that orientation, such that there are $n/r^{1/d}$ points from S in each of the

slabs defined by the hyperplanes.² Do this for each of the c orientations. This gives us a “grid” with $O(r)$ cells. Next, partition each cell that contains more than n/r points into cells with at most n/r and at least $n/(2r)$ points, for example using parallel hyperplanes with an orientation from C .

This partitioning is a BSP (or rather, can easily be converted into a BSP), and it has property (i) by construction. Property (ii) follows from the fact that the first stage produces $O(r)$ cells, and the second stage produces only cells with at least $n/(2r)$ points. To prove property (iii), we note that the number of cells intersected by any c -oriented hyperplane h after the first stage is bounded by the complexity of the arrangement on h induced by the partitioning, which is $O(((c-1)r^{1/d})^{d-1}) = O((c-1)^{d-1}r^{1-1/d})$. It remains to account for the increase in the number of intersected cells due to the second stage. To this end, we observe that the total number of points in the intersected cells is at most $n/r^{1/d}$, since h lies inside one of the slabs induced by the splitting hyperplanes parallel to h . Because the second stage only produces cells with at least $n/(2r)$ points, this implies that there cannot be more than $2r^{1-1/d}$ such cells.

Computing the BSP in $O(n \log n)$ time is not difficult, so we omit the details in this extended abstract. \square

To obtain a BSP that can answer DOP queries efficiently, we recursively apply Lemma 1 (with a suitable value of r) to each cell in the partitioning that contains more than one point. This is similar to the way simplicial partitions are used to construct partition trees. This leads to the following result.

Theorem 1. *Let S be a set of n points in \mathbb{R}^d . For any $\varepsilon > 0$, there is a linear-size c -oriented BSP tree \mathcal{T} whose depth is $O(\log n)$ and that can answer DOP queries in $O(n^{1-1/d+\varepsilon} + k)$ time, where k is the number of reported points.*

3 From bsp-Trees to dop-Trees

In this section we describe and analyze a general method for creating a DOP-tree on a set S of c -DOPs in the plane from a c -oriented BSP on a set of representative points. Our method will create a DOP-tree of branching degree at most $2c + 3$, which can be turned into a binary DOP-tree as a postprocessing step. The method works as follows.

1. Pick an arbitrary representative point in each DOP in S . Let $P := P(S)$ be the resulting set of representative points.
2. Construct a c -oriented BSP \mathcal{T}_P on the set P .
3. Next, transform the BSP \mathcal{T}_P into a BVH \mathcal{T}_S by inserting the DOPs from S into \mathcal{T}_P in a top-down manner, starting at the root of \mathcal{T}_P with the complete input set S . A recursive call proceeds as follows. We get as input a set of DOPs $S_\nu \subseteq S$ which is to be inserted into the subtree rooted at a node ν . Instead of the splitting line ℓ_ν stored at ν , we store at ν the bounding DOP

² For simplicity we assume here and in the sequel that the set of points is in general position.

$bdop(\nu) := bdop(S_\nu)$ of all DOPs in S_ν , thus converting a BSP-node into a BVH-node. We then split S_ν into three sets. The set S_ν^- contains all DOPs in S_ν that lie completely to the left of ℓ_ν , the set S_ν^+ contains all DOPs in S_ν to the right of ℓ_ν , and S_ν^\times contains the DOPs in S_ν that are intersected by ℓ_ν .

- (i) The set S_ν^- (if non-empty) is inserted recursively into the left child ν^- of ν , and the set S_ν^+ (if non-empty) is inserted recursively into the right child ν^+ of ν .
- (ii) Recall that \mathcal{C} is the set of lines defining the c orientations of the splitting lines used by the BSP \mathcal{T}_P (and the DOPs). Remove the line parallel to ℓ_ν from \mathcal{C} , to obtain a set \mathcal{C}^* . Construct a DOP-tree for the set S_ν^\times (if non-empty) by calling a subroutine $CreateSubTree(S_\nu^\times, \mathcal{C}^*)$ and make this tree a subtree of ν .

$CreateSubTree$ is a recursive subroutine, which works as follows.

$CreateSubTree(S^*, \mathcal{C}^*)$

1. If $\mathcal{C}^* = \emptyset$, create a subtree for S^* —we call such a subtree a 0 -tree—using a special subroutine $CreateZeroTree$ to be described later. Otherwise, proceed with steps 2–4.
2. Create a root node μ for the tree to be constructed. Define $S_\mu := S^*$ and $\mathcal{C}_\mu := \mathcal{C}^*$. Store the bounding dop $bdop(S_\mu)$ at μ .
3. Let $\{\vec{n}_1, \dots, \vec{n}_{2c}\}$ be the normals to the lines in \mathcal{C} (not only the lines in \mathcal{C}^*). Note that for every line we have normals in both directions. For each normal \vec{n}_i in turn, we remove from S_μ the DOP D_i extending furthest in the direction \vec{n}_i , and store it in a leaf μ_i directly below μ . We call such leaves *priority leaves*. Note that a DOP stored in a priority leaf because it is extreme in some direction \vec{n}_i will not be considered when we look for extreme DOPs in subsequent directions.
4. Let $S' := S_\mu \setminus \{D_1, \dots, D_{2c}\}$ be the remaining DOPs in S_μ . If S' is not empty, we find a splitting line ℓ_μ parallel to the first line in \mathcal{C}_μ , such that ℓ_μ splits the set of representative points of the DOPs in S' in two sets of roughly equal size. Now we create three sets of DOPs S_μ^- , S_μ^+ , and S_μ^\times that contain the DOPs in S' that lie completely to the left, completely to the right, or across ℓ_μ , respectively.
 - (i) Create the left and right subtree of μ by calling $CreateSubTree(S_\mu^-, \mathcal{C}_\mu)$ and $CreateSubTree(S_\mu^+, \mathcal{C}_\mu)$, respectively. (If S_μ^- or S_μ^+ is empty the corresponding call can be skipped.)
 - (ii) Let \mathcal{C}_μ^* be the set \mathcal{C}_μ with its first line, which is parallel to ℓ_μ , removed. Create a subtree of μ for S_μ^\times (if this set is non-empty) by calling $CreateSubTree(S_\mu^\times, \mathcal{C}_\mu^*)$.

Note that the algorithm above can create nodes of degree one, when only one of the subsets for which a subtree is created is non-empty. We remove these nodes in a postprocessing step.

This finishes the description of the construction algorithm; the subroutine $CreateZeroTree$ that creates the 0-trees depends on the application—whether or

not the input DOPs are disjoint—and we defer its description to the relevant part of the next section. In the remainder of this section we will prove bounds on the performance of the DOP-tree created with the algorithm above. In the analysis, we will make one assumption on the 0-trees created by *CreateZeroTree*: every non-leaf node in a 0-tree has priority leaves, similar to the priority leaves of the nodes created by *CreateSubTree*. Moreover, we define $q_0()$ to be a function that specifies the point-query time in a 0-tree, that is, $q_0(m) + O(k)$ is the number of nodes visited when a 0-tree with m objects is queried with a point and k answers are reported.

We first need to introduce some definitions. Any node which was already present in the original BSP \mathcal{T}_P will be called a c -node. For $0 < m < c$, an m -node ν will be any node constructed in a call to *CreateSubTree*(S^*, \mathcal{C}^*) with $|\mathcal{C}^*| = m$. An m -tree is a subtree rooted at an m -node; thus an m -tree only contains m' -nodes for $m' \leq m$. The k -parent of an m -node ν , for some $m < k \leq c$, is its lowest ancestor that is a k -node.

Lemma 2. *The BVH \mathcal{T}_S uses $O(n)$ storage and its depth, excluding 0-trees, is $O(\text{depth}(\mathcal{T}_P))$. Given \mathcal{T}_P , the DOP-tree \mathcal{T}_S , excluding the 0-trees, can be constructed in $O(n \cdot \text{depth}(\mathcal{T}_P))$ time.*

Next we prove the bounds on the number of nodes in \mathcal{T}_S visited by a point query and a DOP query. To this end, we associate a region of the plane with every node ν in \mathcal{T}_S . Consider all the ancestors of ν . At each such ancestor μ , we used a splitting line ℓ_μ to guide the construction; the node ν can either lie in the subtree corresponding to the subset lying completely in one of the half-planes defined by ℓ_μ , or not. (In the latter case the ν is a priority leaf below μ , or ν lies in the subtree created for S_μ^\times .) The region of ν , denoted $\text{region}(\nu)$, is defined as the intersection of all half-planes corresponding to ancestors of the former type. Note that for any node ν in \mathcal{T}_S the bounding dop $bdop(\nu)$ is a subset of $\text{region}(\nu)$.

Observation 2. The number of visited c -nodes in \mathcal{T}_S is at most the number of visited nodes in \mathcal{T}_P .

We now analyse the number of nodes visited by a point query.

Lemma 3. *The number of nodes visited by a point query in \mathcal{T}_S , including visited nodes in its 0-trees, is $O(q_0(n) \cdot \text{depth}(\mathcal{T}_P) \cdot \log^{c-1} n + k)$.*

Proof. In \mathcal{T}_P a point query q visits $O(\text{depth}(\mathcal{T}_P))$ nodes, so q is contained in the bounding DOPs of at most $O(\text{depth}(\mathcal{T}_P))$ c -nodes in \mathcal{T}_S . For each such node, its $(c - 1)$ -subtree may also contain bounding DOPs that contain q . A point query in an m -tree ($0 < m < c$) is contained in the bounding DOPs of $O(\log n)$ m -nodes, since the depth of an m -tree, excluding its 0-trees, is $O(\log n)$. At each m -node, its $(m - 1)$ -subtree may also contain bounding DOPs that contain q . Thus, an m -tree on n DOPs, excluding the 0-trees, contains at most $T(n, m) = O(\log n)(1 + T(n, m - 1))$ bounding DOPs that contain q in total, where $T(n, 1) = O(\log n)$. This solves to $T(n, m) = O(\log^m n)$, and the total number of nodes

visited in \mathcal{T}_S , excluding 0-trees, is therefore $O(\text{depth}(\mathcal{T}_P) \cdot \log^{c-1} n)$. For each visited 1-node ν , we may need to visit $q_0(n) + O(k_\nu)$ nodes in its 0-subtree \mathcal{T}_ν , where k_ν is the number of answers found in that structure. Thus we visit $O(q_0(n) \cdot \text{depth}(\mathcal{T}_P) \cdot \log^{c-1} n + k)$ nodes in total. \square

Now we are going to analyse the number of internal nodes visited by a DOP query in \mathcal{T}_S , including its 0-trees; the number of leaf-nodes is at most a constant factor more. To this end we introduce the notion of *defining segments*. The set of defining segments of an m -node ν , denoted $\text{DefSeg}(\nu)$, is the intersection of $bdop(\nu)$ with the splitting lines ℓ_μ of all k -parents μ of ν (for every $k \in \{m + 1, \dots, c\}$). Note that any DOP in the subtree rooted at ν intersects all defining segments of that node.

We also make the following definition. We say that two edges of a query range Q (or some other c -DOP) are *adjacent* if their orientations are adjacent in the cyclic ordering of the set \mathcal{C} of all orientations. Hence, if Q has $2c$ edges then this corresponds to the usual definition. If, however, some orientations are not used by Q then two edges may be non-adjacent even when they are incident to the same vertex.

We distinguish between the following types of visited nodes:

- Inner nodes** are nodes ν such that $bdop(\nu)$ is completely contained in Q ;
- Side nodes** are nodes ν such that $bdop(\nu)$ intersects only one edge of Q ;
- Stabbing nodes** are nodes ν such that $bdop(\nu)$ intersects at least two edges of Q (but no vertex), and have a defining segment that intersects at most one edge of Q ;
- Embracing nodes** are nodes ν such that $bdop(\nu)$ and all defining segments $\text{DefSeg}(\nu)$ intersect at least two non-adjacent edges of Q ;
- Corner nodes** are nodes ν such that $bdop(\nu)$ contains at least one vertex of Q ;

Lemma 4. *The number of inner nodes, side nodes, and stabbing nodes visited by a DOP-query Q in \mathcal{T}_S , including such nodes in its 0-trees, is $O(k)$. Furthermore the number of visited corner nodes is $O(q_0(n) \cdot \text{depth}(\mathcal{T}_P) \cdot \log^{c-1} n + k)$.*

We will now bound the number of embracing nodes. Observe that all ancestors of embracing nodes are embracing nodes.

Lemma 5. *An m -tree ($0 \leq m < c$) on n DOPs whose root is an embracing node contains $O(\log^m n)$ embracing nodes—excluding embracing nodes in the associated 0-trees.*

Proof. Let ν be an embracing m -node. Since $m < c$, the node ν has at least one defining segment s , and since ν is an embracing node, s intersects two edges of Q . Let $s' := s \cap Q$. Since the bounding DOPs $bdop(\nu^-)$ and $bdop(\nu^+)$ of the left and right child of ν are disjoint, they cannot both contain s' completely. Any child that does not contain s' completely, has a defining segment that intersects less than two edges of Q and therefore cannot be an embracing node. Hence, at most one of ν^- and ν^+ can be an embracing node. In addition, the child created for S_ν^\times may be an embracing $(m - 1)$ -node. The maximum number of embracing

nodes in an m -tree on n DOPs is therefore $T(n, m) \leq 1 + T(n/2, m) + T(n, m - 1)$, where $T(n, m) = 1$ for $n \leq 2c$, and $T(n, 0) = 0$ for any n . This recurrence solves to $T(n, m) = O(\log^m n)$. \square

Lemma 6. *The number of embracing nodes visited by a DOP-query Q in \mathcal{T}_S , excluding its 0-trees, is $O(\text{tna}(\mathcal{T}_P, Q) + \text{tna_disj}(\mathcal{T}_P, Q) \log^{c-1} n)$, where $\text{tna}(\mathcal{T}_P, Q)$ is the number of cells in \mathcal{T}_P that intersect at least two non-adjacent edges of Q , and $\text{tna_disj}(\mathcal{T}_P, Q)$ is the maximum size of a set of such cells that are pairwise disjoint.*

Proof. By Observation 2, the number of embracing c -nodes is at most the number of cells in \mathcal{T}_P that intersect at least two non-adjacent edges of Q , which is $O(\text{tna}(\mathcal{T}_P, Q))$ by definition.

Now consider the subgraph \mathcal{T}' of \mathcal{T}_P that consists of the nodes for all such cells. Since the cells at the leaves of \mathcal{T}' are disjoint, \mathcal{T}' has $O(\text{tna_disj}(\mathcal{T}_P, Q))$ leaves. Suppose ν is an embracing c -node that has an embracing $(c - 1)$ -child ν^\times . Then the cutting line ℓ_ν used at ν must intersect two edges of Q . Since no cutting line can intersect edges of adjacent orientations, it must, in fact, intersect two non-adjacent edges of Q . Then the left and right child of ν in \mathcal{T}_P also intersect those two non-adjacent edges of Q , and therefore ν is a node of degree two in \mathcal{T}' . That tree \mathcal{T}' has $O(\text{tna_disj}(\mathcal{T}_P, Q))$ nodes of degree two, and therefore there can be only $O(\text{tna_disj}(\mathcal{T}_P, Q))$ embracing $(c - 1)$ -children of c -nodes. By Lemma 5 they may have $O(\text{tna_disj}(\mathcal{T}_P, Q) \log^{c-1} n)$ embracing descendants in total. Adding up the bounds proves the lemma. \square

Lemma 7. *The total number of embracing nodes in 0-trees visited by a DOP-query Q is $O(q_0(n) \cdot \text{depth}(\mathcal{T}_P) \cdot \log^{c-1} n + k)$.*

The following lemma, which follows immediately from the preceding lemmas, summarizes the performance of our DOP-tree construction algorithm.

Lemma 8. *Let S be a set of n c -DOPs, and let \mathcal{T}_P be a c -oriented BSP on the set of representative points of S . Then there is a BVH \mathcal{T}_S on S such that:*

- (i) *a point query visits $O(q_0(n) \cdot \text{depth}(\mathcal{T}_P) \cdot \log^{c-1} n + k)$ nodes;*
- (ii) *a DOP query with a DOP Q visits $O(\text{tna}(\mathcal{T}_P, Q) + (\text{tna_disj}(\mathcal{T}_P, Q) + q_0(n) \cdot \text{depth}(\mathcal{T}_P)) \cdot \log^{c-1} n + k)$ nodes,*

where k is the number of DOPs in S that intersect Q ; $\text{tna}(\mathcal{T}_P, Q)$ is the number of cells in \mathcal{T}_P that intersect at least two non-adjacent edges of Q ; $\text{tna_disj}(\mathcal{T}_P, Q)$ is the maximum size of a set of such cells that are pairwise disjoint; the number of nodes visited by a point query in any 0-tree on n DOPs is $q_0(n) + O(k)$.

4 Applications

By combining the c -oriented BSP developed in Section 2 with the BSP-to-DOP-tree conversion algorithm of the previous section we will obtain the main results of this paper. We consider two cases, depending on whether or not the DOPs in the input set S are disjoint.

Disjoint Input. We start with the case where the input DOPs are disjoint. The efficiency of our solution is based on the following lemma.

Lemma 9. *If the DOPs in the input set S are disjoint, then a 0-tree stores only a single DOP.*

Proof. Suppose there are two DOPs in a 0-tree. Since the DOPs are disjoint, they can be separated by a line ℓ parallel to an edge of one of the DOPs and, hence, by a line parallel to a line in \mathcal{C} . However, the set of defining segments of a 0-tree contains a defining segment for each direction, in particular one parallel to ℓ . This defining segment cannot intersect both DOPs, contradicting that each DOP stored in the 0-tree intersects all defining segments by construction. \square

Together with Theorem 1 and Lemma 8, the lemma above implies the following result.

Theorem 3. *Let S be a set of n disjoint c -DOPs in the plane. There is a DOP-tree for S such that DOP queries can be answered in time $O(n^{1/2+\varepsilon} + k)$, where k is the number of reported answers. Point queries can be answered in $O(\log^c n)$ time.*

Intersecting Input. If the input DOPs can intersect, then a 0-tree may have to store more than one DOP. In fact, if all input DOPs have a non-empty common intersection then it can even happen that all input DOPs end up in the same 0-tree. We therefore need to develop an efficient DOP-tree for intersecting input. Fortunately, the so-called cs-boxtree described by Agarwal *et al.* [1] can be generalized to obtain a DOP-tree with optimal query time.

Theorem 4. *For any set of (possibly intersecting) c -DOPs in the plane, there is a DOP-tree such that DOP queries can be answered in time $O(n^{1-1/c} + k)$, where k is the number of reported answers. Moreover, the bound on the query time is optimal, even for point queries: for any n , there is a set S of n c -DOPs such that for any DOP-tree \mathcal{T} on S there is a query point p not contained in any DOP from S such that a query with p visits $\Omega(n^{1-1/c})$ nodes in \mathcal{T} .*

We now consider the setting that is probably most relevant in practice: the DOPs may intersect, but not too much. To quantify this we use the so-called *stabbing number* of the input set S . This is the smallest number σ such that no point in the plane is contained in more than σ DOPs from S . For example, if the DOPs in S are disjoint, then $\sigma = 1$. In practice, especially when the DOPs from S are bounding DOPs of an underlying set of disjoint objects, one may expect that σ is some small constant. We construct the DOP-tree by combining our previous results: we use the general construction algorithm of the previous section with the BSP from Theorem 1 and a subroutine *CreateZeroTree* that builds 0-trees according to Theorem 4.

To analyse the performance of this structure, we need to bound the number of DOPs that can end up in any 0-tree. Below we will prove that this number is bounded by $O(\sigma)$. This leads to the following, almost optimal, result.

Theorem 5. *For any set of c -DOPs in the plane with stabbing number σ , there is a DOP-tree such that c -oriented range queries can be answered in $O(n^{1/2+\varepsilon} + \sigma^{1-1/c} \log^c n + k)$ time, where k is the number of reported DOPs. The time for point queries is $O(\sigma^{1-1/c} \log^c n + k)$.*

It remains to prove the claim that the maximum number of DOPs in any 0-tree is $O(\sigma)$. Recall that all DOPs ending up in the 0-tree rooted at some node ν intersect all defining segments of ν . Thus we need to bound the maximum size of any set \mathcal{D} of c -DOPs with the following properties.

- (P1) There is a set L of c lines such that every edge of any DOP in \mathcal{D} is parallel to some line in L ;
- (P2) (the interior of) each DOP in \mathcal{D} intersects every line in L ;
- (P3) no point in the plane is within the interior of more than σ DOPs from \mathcal{D} .

We begin with a simple proof for a special, but interesting case.

Lemma 10. *Let \mathcal{D} be a set of DOPs that are bounding DOPs of some underlying set of disjoint objects and satisfying properties (P1)–(P3). Then $|\mathcal{D}| \leq 4c\sigma + 1$.*

Proof. It follows from Lemma 9 that any two DOPs in \mathcal{D} intersect. Furthermore, for any two intersecting DOPs $D, D' \in \mathcal{D}$ there must be a vertex from D inside D' , or vice versa. This follows since D and D' are bounding DOPs of disjoint objects. We charge the intersection between D and D' to this vertex. By property (P3), any vertex can be charged at most σ times. Since a DOP has at most $2c$ vertices we can have at most $2c|\mathcal{D}|\sigma$ intersections, otherwise a vertex would be charged too often. On the other hand, any two DOPs in \mathcal{D} intersect, so there are $\binom{|\mathcal{D}|}{2} = |\mathcal{D}|(|\mathcal{D}| - 1)/2$ pairwise intersections. Hence, $|\mathcal{D}|(|\mathcal{D}| - 1)/2 \leq 2c|\mathcal{D}|\sigma$, which implies $|\mathcal{D}| \leq 4c\sigma + 1$. □

Bounding the size of \mathcal{D} for the general case, where the DOPs in \mathcal{D} can intersect in an arbitrary manner, is a lot more difficult, and we have not been able to get a linear dependency on c , as in Lemma 10. Nevertheless we can prove a bound that is linear in σ .

Theorem 6. *Let \mathcal{D} be a set of DOPs satisfying properties (P1)–(P3). Then $|\mathcal{D}| = O(c^4\sigma)$.*

Proof (sketch): For each DOP $D \in \mathcal{D}$ we can prove that there is a cell in the arrangement \mathcal{A}_L induced by the lines in L such that D intersects at least three edges of that cell. It follows that it intersects three edges of a cell in the arrangement \mathcal{A}' of the three lines from L that contain those edges of \mathcal{A}_L . For any such arrangement \mathcal{A}' , we show there is a set of $O(c)$ points such that if a DOP intersects all edges of any three-edge cell in \mathcal{A}' , it must contain at least one point. Hence the total number of DOPs in \mathcal{D} cannot exceed $\binom{c}{3}O(c)\sigma = O(c^4\sigma)$. □

5 Concluding Remarks

Our paper suggests several interesting open problems. The first is to see whether any set of n points in \mathbb{R}^d admits a simplicial partition of size r with crossing number $O(r^{1-1/d})$ that consists of *disjoint* simplices or that is perhaps even a BSP subdivision. We have shown this for the crossing number of c -oriented hyperplanes, but it would be very interesting to see if this is possible in general. Another interesting open problem is to improve the dependency on c in Theorem 6.

References

1. P.K. Agarwal, M. de Berg, J. Gudmundsson, M. Hammar, H.J. Haverkort, Box-trees and R-trees with near-optimal query time, *Discrete Comput. Geom.*, 28, 291–312, 2002.
2. P.K. Agarwal and J. Erickson. Geometric range searching and its relatives. In: B. Chazelle, J. Goodman, and R. Pollack (Eds.), *Advances in Discrete and Computational Geometry*, Vol. 223 of *Contemporary Mathematics*, pages 1–56, American Mathematical Society, 1998.
3. S. Gottschalk, M.C. Lin, and D. Manocha, OBB-Tree: a hierarchical structure for rapid interference detection, In *Proc. Computer Graphics (SIGGRAPH)*, 171–180, 1996.
4. H.J. Haverkort, Results on Geometric Networks and Data Structures. Ph.D. Thesis, Utrecht University, 2004.
5. H.J. Haverkort, M. de Berg, and J. Gudmundsson. Box-Trees for Collision Checking in Industrial Installations. In *Proc. 18th ACM Symp. on Computational Geometry*, pages 53–62, 2002
6. H.V. Jagadish. Spatial Search with Polyhedra. In *Proc. Int. Conf. Data Engineering (ICDE)*, 311–319, 1990
7. J.T. Klosowski, M. Held, J.S.B. Mitchell, H. Sowizral, and K. Zikan, Efficient Collision Detection Using Bounding Volume Hierarchies of k -DOPs *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36, 1998.
8. Y. Manolopoulos, Y. Theodoridis, and V. Tsotras, *Advanced Database Indexing*, Kluwer Academic Publishers, 1999.
9. J. Matoušek, Efficient partition trees *Discrete Comput. Geom.*, 8:315–334, 1992.
10. J. Nievergelt and P. Widmayer. Spatial data structures: concepts and design choices. In: M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer (eds.), *Algorithmic Foundations of Geographic Information Systems*, LNCS 1340, Springer-Verlag, 153–197, 1997
11. I. Sitzmann and P.J. Stuckey. The O-TreeA Constraint-Based Index Structure, technical report, University of Melbourne, 1999.

Matching Point Sets with Respect to the Earth Mover's Distance

Sergio Cabello^{1,*}, Panos Giannopoulos², Christian Knauer²,
and Günter Rote²

¹ IMFM, Department of Mathematics, Jadranska 19, SI-1000 Ljubljana, Slovenia
`sergio.cabello@imfm.uni-lj.si`

² Institut für Informatik, Freie Universität Berlin,
Takustraße 9, D-14195 Berlin, Germany
`{panos, knauer, rote}@inf.fu-berlin.de`

Abstract. The Earth Mover's Distance (EMD) between two weighted point sets (point distributions) is a distance measure commonly used in computer vision for color-based image retrieval and shape matching. It measures the minimum amount of work needed to transform one set into the other one by weight transportation.

We study the following shape matching problem: Given two weighted point sets A and B in the plane, compute a rigid motion of A that minimizes its Earth Mover's Distance to B . No algorithm is known that computes an exact solution to this problem. We present simple FPTAS and polynomial-time $(2 + \epsilon)$ -approximation algorithms for the minimum Euclidean EMD between A and B under translations and rigid motions.

1 Introduction

Shape matching is a fundamental problem in computational geometry: given two shapes A and B and a distance measure, one wants to determine a transformed — such as rotated and/or translated — version of, say, A that attains the minimum possible distance to B ; see Alt and Guibas [2] for a survey.

In a typical application such as content-based image retrieval, a shape, or pattern in general, is usually given by a set of feature *weighted* points in some metric space, e.g., Euclidean space or CIE-Lab color space [15]. The weight of a point normally represents its significance, that is, the larger the weight, the more important the point for the whole pattern.

The Earth Mover's Distance (EMD) is a similarity measure for weighted point sets. Informally, it measures the minimum amount of work needed to transform one set into the other one by weight transportation; a formal definition will be given shortly. The EMD is the discrete version of the *Monge-Kantorovich mass transportation distance* whose potential use for measuring shape similarity was first proposed by Mumford [13]. Since then, the EMD has turned into a popular

* Research partially supported by the European Community Sixth Framework Programme under a Marie Curie Intra-European Fellowship.

similarity measure in computer vision with applications in color-based image retrieval [10,12,15], shape matching [7,8,9] and music score matching [16].

Let $A = \{a_1, \dots, a_m\}$ and $B = \{b_1, \dots, b_n\}$ be two planar weighted point sets with $m \leq n$. A weighted point $a_i \in A$ is defined as $a_i = \{(x_{a_i}, y_{a_i}), w_i\}$, $i = 1, \dots, m$, where $(x_{a_i}, y_{a_i}) \in \mathbb{R}^2$ and $w_i \in \mathbb{R}^+ \cup \{0\}$ is its weight. A weighted point $b_j \in B$ is defined similarly as $b_j = \{(x_{b_j}, y_{b_j}), u_j\}$, $j = 1, \dots, n$. Let $W = \sum_{i=1}^m w_i$ and $U = \sum_{j=1}^n u_j$ be the total weight, or simply weight, of A and B respectively.

We study the following problem: Given two weighted point sets A and B find a rigid motion – sometimes referred to as isometry – of A that minimizes its Earth Mover's Distance (EMD) to B . Note that we are interested in transformations that change only the position of the points, not their weights. We consider B to be fixed, while A can be translated and/or rotated relative to B . We assume some initial positions for both sets, denoted simply by A and B . We denote by \mathcal{I} the set of all possible rigid motions in the plane, by R_θ a rotation about the origin by some angle $\theta \in [0, 2\pi)$, and by $T_{\vec{t}}$ a translation by $\vec{t} \in \mathbb{R}^2$. Any rigid motion $I \in \mathcal{I}$ can be uniquely defined as a translation followed by a rotation, that is, $I = I_{\vec{t}, \theta} = R_\theta \circ T_{\vec{t}}$, for some $\theta \in [0, 2\pi)$ and $\vec{t} \in \mathbb{R}^2$. In general, transformed versions of A are denoted by $A(\vec{t}, \theta) = \{a_1(\vec{t}, \theta), \dots, a_m(\vec{t}, \theta)\}$ for some $I_{\vec{t}, \theta} \in \mathcal{I}$. For simplicity, translated only versions of A are denoted by $A(\vec{t}) = \{a_1(\vec{t}), \dots, a_m(\vec{t})\}$. Similarly, rotated only versions of A are denoted by $A(\theta) = \{a_1(\theta), \dots, a_m(\theta)\}$.

The EMD between $A(\vec{t}, \theta)$ and B , is a function $\text{EMD} : \mathcal{I} \rightarrow \mathbb{R}^+ \cup \{0\}$ defined as

$$\text{EMD}(\vec{t}, \theta) = \min_{F \in \mathcal{F}(A, B)} \frac{\sum_{i=1}^m \sum_{j=1}^n f_{ij} d_{ij}(\vec{t}, \theta)}{\min\{W, U\}},$$

where $d_{ij}(\vec{t}, \theta)$ is the distance of $a_i(\vec{t}, \theta)$ to b_j , and $F = \{f_{ij}\} \in \mathcal{F}(A, B)$ with $\mathcal{F}(A, B)$ being the set of all feasible flows between A and B defined by the constraints: (i) $f_{ij} \geq 0$, $i = 1, \dots, m$, $j = 1, \dots, n$, (ii) $\sum_{j=1}^n f_{ij} \leq w_i$, $i = 1, \dots, m$, (iii) $\sum_{i=1}^m f_{ij} \leq u_j$, $j = 1, \dots, n$, and (iv) $\sum_{i=1}^m \sum_{j=1}^n f_{ij} = \min\{W, U\}$. In case that \vec{t} or θ or both are constant, we simply write $\text{EMD}(\theta)$, $\text{EMD}(\vec{t})$ and EMD respectively. We deal with the Euclidean EMD where d_{ij} is given by the L_2 -norm. Our problem can be now stated as follows:

Given two weighted point sets A, B in the plane, compute a rigid motion $I_{\vec{t}_{\text{opt}}, \theta_{\text{opt}}}$ that minimizes $\text{EMD}(\vec{t}, \theta)$.

The problem was first studied by Cohen and Guibas [7] who presented a Flow – Transformation iteration which alternates between finding the optimum flow for a given transformation, and the optimum transformation for a given flow. They showed that this iterative procedure converges, but not necessarily to the global optimum. Computing the EMD for a given transformation is actually the transportation problem, a special minimum cost network flow problem [1], for the solution of which there is a variety of polynomial time algorithms; see Section 2. However, the task of finding the optimal transformation for a given flow is not trivial: for translations, it reduces to the *Fermat-Weber problem* [6] where one wants to find a point that minimizes the sum of weighted distances to a set of given points. No exact solution to the latter problem is known even

in the real RAM model of computation [4]. Cohen and Guibas gave also simple algorithms that compute the optimum translation for the special case where $W = U$ and d_{ij} is the squared Euclidean distance. This case is quite restrictive since, in general, the sets need not have the same weight, and the use of squared Euclidean distance is statistically less robust than Euclidean distance [4]. To the best of the authors' knowledge, no algorithm that computes the optimal translation and/or rotation is known for the Euclidean EMD.

The EMD is a metric when d_{ij} is a metric and $W = U$ [15]. When $W \neq U$ the EMD inherently performs partial matching since a portion of the weight of the 'heavier' set remains unmatched. The case where $w_i = u_j = 1, i = 1, \dots, m, j = 1, \dots, n$ deserves special attention: the integer solutions property of the minimum cost flow problem and the fact that $0 \leq f_{ij} \leq 1$ imply that there is a minimum cost flow from A to B that results in a *partial assignment* between A and B , that is, a perfect matching between A and a subset of B ; when $n = m$ the problem is simply referred to as the *assignment* problem.

We give simple polynomial-time $(1 + \epsilon)$ and $(2 + \epsilon)$ -approximation algorithms for the minimum EMD of two weighted point sets in the plane under translations and rigid motions. The algorithms for translations are given in Section 4 and for rigid motions in Section 5. In the general case where the sets have unequal total weights we compute a $(1 + \epsilon)$ -approximation in $O((n^3 m / \epsilon^4) \log^2(n/\epsilon))$ time for translations and a $(2 + \epsilon)$ -approximation in $O((n^4 m^2 / \epsilon^4) \log^2(n/\epsilon))$ time for rigid motions. When the sets have equal total weights, the respective running times decrease to $O((n^2 / \epsilon^4) \log^2(n/\epsilon))$ and $O((n^3 m / \epsilon^4) \log^2(n/\epsilon))$.

We also show how to compute a $(1 + \epsilon)$ -approximation of the minimum cost assignment under translations and rigid motions in $O((n^{3/2} / \epsilon^{7/2}) \log^5 n)$ and $O((n^{7/2} / \epsilon^{9/2}) \log^6 n)$ time respectively. Finally, we give probabilistic $(1 + \epsilon)$ -approximations of the minimum cost partial assignment under translations in $O((n^3 / \epsilon^4) \log^2(n/\epsilon) \log n)$ time and under rigid motions in $O((n^4 m / \epsilon^5) \log^2(n/\epsilon) \log n \log m)$ time; both algorithms succeed with high probability.

In Section 3, we give two simple lower bounds on the EMD that are vital to our approximation algorithms. These algorithms need to compute the EMD for a given transformation. Computing the EMD exactly is expensive, and unnecessary since we opt for approximations for our original problem. We begin by showing how to get a $(1 + \epsilon)$ -approximation of the EMD in almost quadratic time.

2 Approximating the EMD

The fastest known strongly polynomial-time algorithm for the minimum cost flow problem on a graph $G(V, E)$ is due to Orlin [14], and runs in $O((|E| \log |V|)(|E| + |V| \log |V|))$ time. Several weakly polynomial-time algorithms exist that assume integer edge costs and/or integer weights [1]. For the transportation problem in the plane, Atkinson and Vaidya [3] gave a weakly polynomial-time algorithm that assumes integer supplies and demands and runs in $O(|V|^{2.5} \log(|V|) \log W)$ time, where W is the largest supply or demand.

Consider the complete bipartite graph $G(V, E)$ with $V = A \cup B$ and $E = \{(a_i, b_j) : a_i \in A, b_j \in B\}$. Using the algorithm of Callahan and Kosaraju [5], we can construct, in $O(n \log n + (n/\epsilon^2) \log 1/\epsilon)$ time, a linear size $(1 + \epsilon)$ -spanner G_s , i.e., a graph $G_s(V, E_s)$ with $|E_s| = O(n/\epsilon)$ such that the shortest path between any two points in G_s is at most $(1 + \epsilon)$ times the Euclidean distance of the points. Running the algorithm of Orlin on G_s produces an approximate value EMD_s . For convenience, this procedure is referred to as $APXEMD(A, B, \epsilon)$ and given in Figure 1. Using the fact that the distances in the spanner approximate the distances in the complete graph, it is not hard to prove the following result.

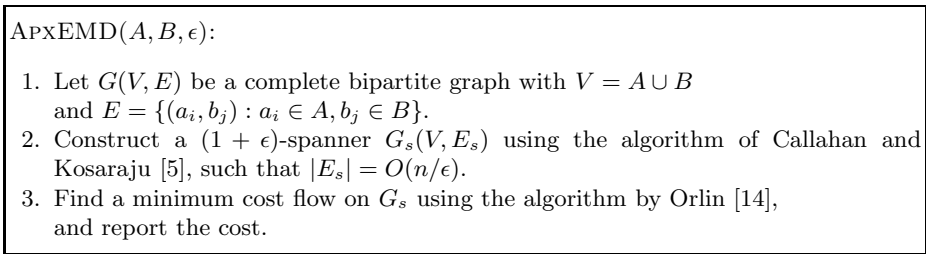


Fig. 1. Algorithm $APXEMD(A, B, \epsilon)$

Lemma 1. *For any given $\epsilon > 0$, $APXEMD(A, B, \epsilon)$ computes a value EMD_s such that $EMD \leq EMD_s \leq (1 + \epsilon)EMD$ in $O((n^2/\epsilon^2) \log^2(n/\epsilon))$ time.*

For the assignment or else minimum cost Euclidean bipartite matching problem, Varadarajan and Agarwal [17] presented an algorithm that finds a matching with cost at most $(1 + \epsilon)$ times that of an optimal matching in $O((n/\epsilon)^{3/2} \log^5 n)$ time; we refer to this algorithm as $APXMATCH(A, B, \epsilon)$.

Theorem 1. *[17–Theorem 3.1] Let A and B be two sets of points in the plane with $|A| = |B| = n$. For any given $\epsilon > 0$, a perfect matching between A and B with cost at most $(1 + \epsilon)$ times that of an optimal perfect matching can be computed in $O((n/\epsilon)^{3/2} \log^5 n)$ time.*

3 Lower Bounds on the EMD

The following simple lower bound comes directly from the definition of the EMD.

Observation 1. *Given two weighted point sets A and B , $EMD \geq \min_{i,j} d_{ij}$.*

The next lower bound is due to Rubner et al. [15], and applies to sets with equal weights. It is based on the notion of the *center of mass* of a weighted point set. The center of mass $C(A)$ of a planar weighted point set $A = \{(x_{a_i}, y_{a_i}), w_i\}, i = 1, \dots, m$ is defined as $C(A) = (\sum_{i=1}^m w_i \cdot (x_{a_i}, y_{a_i})) / \sum_{i=1}^m w_i$.

Theorem 2. *[15] Let A and B be two weighted point sets with equal weights. Then $EMD \geq d(C(A), C(B))$.*

4 Approximation Algorithms for Translations

We denote by $\vec{t}_{i \rightarrow j}$ the translation which matches a_i and b_j ; we call such a translation a *point-to-point* translation. Observation 1 implies that the point-to-point translation that is closest to \vec{t}_{opt} gives a 2-approximation of $\text{EMD}(\vec{t}_{\text{opt}})$. Hence, we have the following:

Lemma 2. $\text{EMD}(\vec{t}_{\text{opt}}) \leq \min_{i,j} \text{EMD}(\vec{t}_{i \rightarrow j}) \leq 2\text{EMD}(\vec{t}_{\text{opt}})$.

According to Observation 1, the point-to-point translation which is closest to \vec{t}_{opt} can be at most $\text{EMD}(\vec{t}_{\text{opt}})$ away from \vec{t}_{opt} . This bound is crucial for the $(1 + \epsilon)$ -approximation algorithm given in Figure 2. Using a uniform square grid of suitable size we compute the EMD for a limited number of grid translations within a neighborhood of size $\text{EMD}(\vec{t}_{\text{opt}})$ of every translation $\vec{t}_{i \rightarrow j}$. Note that we do not know $\text{EMD}(\vec{t}_{\text{opt}})$ but we can compute $\min_{i,j} \text{EMD}(\vec{t}_{i \rightarrow j})$ which, according to Lemma 2, approximates $\text{EMD}(\vec{t}_{\text{opt}})$ well-enough. In order to save time, rather than computing EMD exactly, we will approximate it using the procedure APXEMD.

TRANSLATION(A, B, ϵ):

1. Let $\alpha = \min_{i,j} \text{APXEMD}(A(\vec{t}_{i \rightarrow j}), B, 1)$;
 let G be a uniform square grid of spacing $c\epsilon\alpha$, where $c = 1/\sqrt{72}$.
2. For each pair of points $a_i \in A$ and $b_j \in B$ do:
 - (a) Place a disk D of radius α around $\vec{t}_{i \rightarrow j}$.
 - (b) For every grid point \vec{t}_g of any cell of G that intersects D
 compute a value $\widetilde{\text{EMD}}(\vec{t}_g) = \text{APXEMD}(A(\vec{t}_g), B, \epsilon/3)$.
3. Report the grid point \vec{t}_{apx} that minimizes $\widetilde{\text{EMD}}(\vec{t}_g)$.

Fig. 2. Algorithm TRANSLATION(A, B, ϵ)

Theorem 3. Let $A = \{a_1, \dots, a_m\}$ and $B = \{b_1, \dots, b_n\}$ be two weighted point sets in the plane with $m \leq n$. For any given $\epsilon > 0$, TRANSLATION(A, B, ϵ) computes a translation \vec{t}_{apx} such that $\text{EMD}(\vec{t}_{\text{apx}}) \leq (1 + \epsilon)\text{EMD}(\vec{t}_{\text{opt}})$ in $O((n^3 m / \epsilon^4) \log^2(n/\epsilon))$ time.

Proof. From Lemma 2, we have $\text{EMD}(\vec{t}_{\text{opt}}) \leq \min_{i,j} \text{EMD}(\vec{t}_{i \rightarrow j}) \leq 2\text{EMD}(\vec{t}_{\text{opt}})$. From Lemma 1, we have $\text{EMD}(\vec{t}_{i \rightarrow j}) \leq \text{APXEMD}(A(\vec{t}_{i \rightarrow j}), B, 1) \leq 2\text{EMD}(\vec{t}_{i \rightarrow j})$. Hence, since $\alpha = \min_{i,j} \text{APXEMD}(A(\vec{t}_{i \rightarrow j}), B, 1)$, we have that $\text{EMD}(\vec{t}_{\text{opt}}) \leq \alpha \leq 4\text{EMD}(\vec{t}_{\text{opt}})$. Also, according to Observation 1, there is a point-to-point translation $\vec{t}_{i \rightarrow j}$ for which $|\vec{t}_{i \rightarrow j} - \vec{t}_{\text{opt}}| \leq \text{EMD}(\vec{t}_{\text{opt}}) \leq \alpha$. Algorithm TRANSLATION will, at some stage, consider the α -neighborhood of such a translation, and thus, compute a value $\widetilde{\text{EMD}}(\vec{t}_g)$ for some grid point \vec{t}_g for which $|\vec{t}_g - \vec{t}_{\text{opt}}| \leq \sqrt{2(\epsilon\alpha/\sqrt{72})^2}/2 \leq (\epsilon/3)\text{EMD}(\vec{t}_{\text{opt}})$ and, thus, $d_{ij}(\vec{t}_g) \leq d_{ij}(\vec{t}_{\text{opt}}) + (\epsilon/3)\text{EMD}(\vec{t}_{\text{opt}})$; see Figure 3. If $\{f_{ij}\}$ is the optimal flow at \vec{t}_{opt} , we have

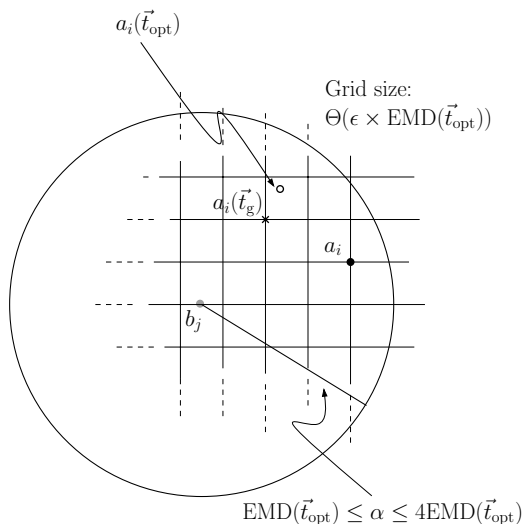


Fig. 3. A pair of points a_i, b_j for which $d_{ij}(\vec{t}_{\text{opt}}) \leq \text{EMD}(\vec{t}_{\text{opt}})$, and a grid translation \vec{t}_g of a_i for which $|\vec{t}_g - \vec{t}_{\text{opt}}| \leq (\epsilon/3)\text{EMD}(\vec{t}_{\text{opt}})$

$$\begin{aligned} \text{EMD}(\vec{t}_g) &\leq \frac{\sum_{i=1}^m \sum_{j=1}^n f_{ij} d_{ij}(\vec{t}_g)}{\min\{W, U\}} \\ &\leq \frac{\sum_{i=1}^m \sum_{j=1}^n f_{ij} (d_{ij}(\vec{t}_{\text{opt}}) + (\epsilon/3)\text{EMD}(\vec{t}_{\text{opt}}))}{\min\{W, U\}} = (1 + \epsilon/3)\text{EMD}(\vec{t}_{\text{opt}}). \end{aligned}$$

From Lemma 1 we have that $\text{EMD}(\vec{t}_g) \leq \widetilde{\text{EMD}}(\vec{t}_g) \leq (1 + \epsilon/3)\text{EMD}(\vec{t}_g)$. Hence, the algorithm reports a translation \vec{t}_{apx} such that

$$\text{EMD}(\vec{t}_{\text{apx}}) \leq \widetilde{\text{EMD}}(\vec{t}_{\text{apx}}) \leq \widetilde{\text{EMD}}(\vec{t}_g) \leq (1 + \epsilon/3)\text{EMD}(\vec{t}_g) \leq (1 + \epsilon)\text{EMD}(\vec{t}_{\text{opt}}),$$

for every $\epsilon \leq 3$. There are nm point-to-point translations, around each of which APXEMD is run for $O(\alpha^2/(\alpha^2\epsilon^2)) = O(1/\epsilon^2)$ grid points. Hence, the algorithm runs in $O((nm/\epsilon^2)(n^2/\epsilon^2) \log^2(n/\epsilon)) = O((n^3m/\epsilon^4) \log^2(n/\epsilon))$ time. \square

4.1 Equal Weight Sets

We consider now the case of sets with equal total weights. Let $\vec{t}_{C(A) \rightarrow C(B)}$ be the translation that matches the centers of mass $C(A)$ and $C(B)$. Theorem 2 suggests the following trivial 2-approximation algorithm: compute $\text{EMD}(\vec{t}_{C(A) \rightarrow C(B)})$; this has been noted also by Klein and Veltpamp [11].

Since \vec{t}_{opt} is at most $\text{EMD}(\vec{t}_{\text{opt}})$ far away from $\vec{t}_{C(A) \rightarrow C(B)}$, we need to search for \vec{t}_{opt} only within a small neighborhood of $\vec{t}_{C(A) \rightarrow C(B)}$. We modify algorithm TRANSLATION as follows: First we compute $C(A)$ and $C(B)$. Then, we run $\text{APXEMD}(A(\vec{t}_{C(A) \rightarrow C(B)}), B, 1)$ and set α to the value returned. Next, we use

the same grid size as in TRANSLATION, and run $\text{APXEMD}(A(\vec{t}_g), B, \epsilon/3)$ for all the grid points \vec{t}_g which are at most α away from $\vec{t}_{C(A) \rightarrow C(B)}$. The minimum over all these approximations gives the desired approximation bound. Since the total number of grid points to be tested is $O(1/\epsilon^2)$, we have saved a nm term from the time bound of Theorem 3.

Theorem 4. *Let $A = \{a_1, \dots, a_m\}$ and $B = \{b_1, \dots, b_n\}$ be two weighted point sets in the plane with equal total weights and $m \leq n$. Then, for any given $\epsilon > 0$, a translation \vec{t}_{apx} such that $\text{EMD}(\vec{t}_{\text{apx}}) \leq (1 + \epsilon)\text{EMD}(\vec{t}_{\text{opt}})$ can be computed in $O((n^2/\epsilon^4) \log^2(n/\epsilon))$ time.*

For the assignment problem under translations we can use the above algorithm running APXMATCH instead of APXEMD, reducing the running time further.

Theorem 5. *For any given $\epsilon > 0$, a $(1 + \epsilon)$ -approximation of the minimum cost assignment under translations can be computed in $O((n^{3/2}/\epsilon^{7/2}) \log^5 n)$ time.*

Note that the latter algorithm can be also applied to equal weight sets with bounded integer point weights by replacing each point by as many points as its weight.

4.2 Partial Assignment

In Observation 1, we saw that there is at least one pair of points a_i, b_j whose distance is at most EMD . It is not hard to see that for the partial assignment case there is a linear number of pairs of points whose distance is at most 2EMD .

Lemma 3. *Given two weighted point sets $A = \{a_1, \dots, a_m\}$, $B = \{b_1, \dots, b_n\}$ with $m \leq n$ and $w_i = u_j = 1, i = 1, \dots, m, j = 1, \dots, n$, there are at least $m/2$ distances d_{ij} with $d_{ij} \leq 2\text{EMD}$.*

Algorithm TRANSLATION tests all possible nm pairs of points a_i, b_j in order to find at least one for which $d_{ij}(\vec{t}_{\text{opt}}) \leq \text{EMD}(\vec{t}_{\text{opt}})$. Using Lemma 3, we can prove that testing a linear number of pairs suffices in order to find one for which $d_{ij}(\vec{t}_{\text{opt}}) \leq 2\text{EMD}(\vec{t}_{\text{opt}})$ with high probability. Algorithm RANDOMTRANSLATION is given in Figure 4; it is a probabilistic version of algorithm TRANSLATION.

Theorem 6. *Let $A = \{a_1, \dots, a_m\}$ and $B = \{b_1, \dots, b_n\}$ be two weighted point sets with $m \leq n$ and $w_i = u_j = 1, i = 1, \dots, m, j = 1, \dots, n$. For any given $\epsilon > 0$, $\text{RANDOMTRANSLATION}(A, B, \epsilon)$ computes a translation \vec{t}_{apx} such that $\text{EMD}(\vec{t}_{\text{apx}}) \leq (1 + \epsilon)\text{EMD}(\vec{t}_{\text{opt}})$ in $O((n^3/\epsilon^4) \log^2(n/\epsilon) \log n)$ time. The algorithm succeeds with probability at least $1 - 2n^{-1}$.*

Proof. According to Lemma 3, $d_{ij}(\vec{t}_{\text{opt}}) \leq 2\text{EMD}(\vec{t}_{\text{opt}})$ for at least $m/2$ distances $d_{ij}(\vec{t}_{\text{opt}})$. Since there are in total nm possible distances $d_{ij}(\vec{t}_{\text{opt}})$, we have that

$$\Pr[d_{ij}(\vec{t}_{\text{opt}}) > 2\text{EMD}(\vec{t}_{\text{opt}})] \leq 1 - m/(2nm) = 1 - 1/(2n)$$

for a random pair a_i, b_j . Thus, the probability that K random draws of a pair a_i, b_j will all fail to give a pair for which $d_{ij}(\vec{t}_{\text{opt}}) \leq 2\text{EMD}(\vec{t}_{\text{opt}})$ is at most

RANDOMTRANSLATION(A, B, ϵ):

1. Repeat $(2/\log e)n \log n$ times:
 - (a) Choose a random pair $(a_i, b_j) \in A \times B$.
 - (b) Let $\alpha_{ij} = \text{APXEMD}(A(\vec{t}_{i \rightarrow j}), B, 1)$.
 Let $\alpha = 2 \min_{i,j} \alpha_{ij}$;
 let G be a uniform square grid of spacing $c\epsilon\alpha$, where $c = 1/\sqrt{288}$.
2. Repeat $(2/\log e)n \log n$ times:
 - (a) Choose a random pair $(a_i, b_j) \in A \times B$.
 - (b) Place a disk D of radius α around $\vec{t}_{i \rightarrow j}$.
 - (c) For every grid point \vec{t}_g of any cell of G that intersects D
 compute a value $\widetilde{\text{EMD}}(\vec{t}_g) = \text{APXEMD}(A(\vec{t}_g), B, \epsilon/3)$.
3. Report the grid point \vec{t}_{apx} that minimizes $\widetilde{\text{EMD}}(\vec{t}_g)$.

Fig. 4. Algorithm RANDOMTRANSLATION(A, B, ϵ)

$(1 - 1/2n)^K$. By choosing $K = (2/\log e)n \log n$ the latter probability is at most $e^{-(\log n)/\log e} = n^{-1}$.

The rest of the proof is almost identical to the proof of Theorem 3. That is, if a pair a_i, b_j for which $d_{ij}(\vec{t}_{\text{opt}}) \leq 2\text{EMD}(\vec{t}_{\text{opt}})$ is tested, then the algorithm will compute in step 1 a value α such that $2\text{EMD}(\vec{t}_{\text{opt}}) \leq \alpha < 8\text{EMD}(\vec{t}_{\text{opt}})$. Similarly, step 2 will report a translation \vec{t}_{apx} such that $\text{EMD}(\vec{t}_{\text{opt}}) \leq \text{EMD}(\vec{t}_{\text{apx}}) \leq (1 + \epsilon)\text{EMD}(\vec{t}_{\text{opt}})$ in $O(((n \log n)/\epsilon^2)(n/\epsilon)^2 \log^2(n/\epsilon)) = O((n^3/\epsilon^4) \log^2(n/\epsilon) \log n)$ time. The algorithm fails to report such a translation if and only if any of its two random steps fail. That is, the algorithm fails with probability at most $2n^{-1}$. \square

5 Approximation Algorithms for Rigid Motions

We first give $(2+\epsilon)$ and $(1+\epsilon)$ -approximation algorithms for rotations for the general and partial assignment case respectively. Then, we combine these algorithm with the $(1 + \epsilon)$ -approximation algorithms for translations to get approximation algorithms for rigid motions.

5.1 Rotations

Let $\angle a_i o b_j$ be the angle between the segments $\overline{o a_i}$ and $\overline{o b_j}$ such that $0 \leq \angle a_i o b_j \leq \pi$. Also, let $\theta_{i \rightarrow j}$ be the rotation by $\angle a_i o b_j$ that aligns the origin o and points a_i and b_j such that both a_i and b_j are on the same side of o . Note that this is the rotation that minimizes $d_{ij}(\theta)$; we call such a rotation an *alignment rotation*.

Lemma 4. *Let a_i and b_j be two points in the plane with $\angle a_i o b_j = \phi$. If a_i is rotated by an angle $\theta \leq \phi$, then $d_{ij}(\theta) < 2d_{ij}$.*

Consider the angle $\angle a_i(\theta_{\text{opt}}) o b_j$ for every pair of points $a_i(\theta_{\text{opt}})$ and b_j and let $\angle a_{i_0}(\theta_{\text{opt}}) o b_{j_0}$ be the smallest of all these angles. Then $\theta_{i_0 \rightarrow j_0}$ is the alignment rotation that is *closest* to θ_{opt} . Similarly to Lemma 2, and using Lemma 4, we can

now prove that this alignment rotation gives a 2-approximation of $\text{EMD}(\theta_{\text{opt}})$. Hence, we have the following:

Lemma 5. $\text{EMD}(\theta_{\text{opt}}) \leq \min_{i,j} \text{EMD}(\theta_{i \rightarrow j}) \leq 2\text{EMD}(\theta_{\text{opt}})$.

By approximating $\min_{i,j} \text{EMD}(\theta_{i \rightarrow j})$ with $\min_{i,j} \text{APXEMD}(A(\theta_{i \rightarrow j}), B, \epsilon/2)$ we can get a $(2 + \epsilon)$ -approximation of $\text{EMD}(\theta_{\text{opt}})$. We refer to this algorithm as $\text{ROTATION}(A, B, \epsilon)$. Apart from the cost value, ROTATION returns the corresponding rotation $\theta_{i \rightarrow j}$ as well.

Lemma 6. *For any given $\epsilon > 0$, a rotation θ_{apx} such that $\text{EMD}(\theta_{\text{apx}}) \leq (2 + \epsilon)\text{EMD}(\theta_{\text{opt}})$ can be computed in $O((n^3m/\epsilon^2) \log^2(n/\epsilon))$ time.*

Partial Assignment. For the partial assignment case we can provide a $(1 + \epsilon)$ -approximation. Let $a_1b_{j_1}, \dots, a_mb_{j_m}$ be the matching corresponding to an optimal integer flow at an optimal rotation θ_{opt} for the problem. Observe that $d_{ij_i}(\theta_{\text{opt}}) \leq m\text{EMD}(\theta_{\text{opt}})$ since $m\text{EMD}(\theta_{\text{opt}}) = \sum_i d_{ij_i}(\theta_{\text{opt}})$. This means that for finding an optimal rotation we only need to consider the rotations $\{\theta \in [0, 2\pi) : d_{ij}(\theta) \leq m\text{EMD}(\theta_{\text{opt}})\}$ for all i, j . Of course, since we do not know the value $\text{EMD}(\theta_{\text{opt}})$, we will instead consider the rotations $R_{ij}(\alpha) = \{\theta \in [0, 2\pi) : d_{ij}(\theta) \leq m\alpha\}$, for some value α such that $\text{EMD}(\theta_{\text{opt}}) \leq \alpha \leq 3\text{EMD}(\theta_{\text{opt}})$.

Inside each R_{ij} we will consider sample rotations Θ_{ij} according to the following. We divide $R_{ij}(\alpha)$ into two parts, $R_{ij}^<(\alpha) = \{\theta \in [0, 2\pi) : d_{ij}(\theta) \leq \alpha\}$ and $R_{ij}^>(\alpha) = \{\theta \in [0, 2\pi) : \alpha \leq d_{ij}(\theta) \leq m\alpha\}$. To handle $R_{ij}^<(\alpha)$, we consider the set of distances $D_{ij}^<(\alpha) = \{k \cdot \frac{\alpha}{18} \in [0, \alpha] \mid k \in \mathbb{N}\}$, which consists of $O(1/\epsilon)$ values. To handle $R_{ij}^>(\alpha)$, we consider the set of distances $D_{ij}^>(\alpha) = \{\alpha(1 + \epsilon/6)^k \in [\alpha, m\alpha] \mid k \in \mathbb{N}\}$, which contains $O(\log_{1+\epsilon} \frac{m\alpha}{\alpha}) = O(\log_{1+\epsilon} m) = O(\epsilon^{-1} \log m)$ values. Let $D_{ij}(\alpha) = \{0, \alpha, m\alpha\} \cup D_{ij}^<(\alpha) \cup D_{ij}^>(\alpha)$, and consider the set of angles $\Theta_{ij} = \{\theta \in [0, 2\pi) \mid d_{ij}(\theta) \in D_{ij}(\alpha)\}$. This finishes the description of Θ_{ij} , and therefore Θ_{ij} contains $O(\epsilon^{-1} \log m)$ angles.

We claim that the best rotation in $\bigcup_{i,j} \Theta_{ij}$ provides a $(1 + \epsilon)$ -approximation for $\text{EMD}(\theta_{\text{opt}})$. The main idea is that the angles from Θ_{ij_i} that are in $R_{ij}^<(\alpha)$ take care for the case when $d_{ij_i}(\theta_{\text{opt}})$ is at most $\alpha \geq \text{EMD}(\theta_{\text{opt}})$ by controlling the absolute error this pair produces in the approximation, while the angles from Θ_{ij_i} that are in $R_{ij}^>(\alpha)$ take care for the case when $d_{ij_i}(\theta_{\text{opt}})$ is between α and $m\text{EMD}(\theta_{\text{opt}}) \leq m\alpha$ by controlling the relative error that the pair $a_ib_{j_i}$ produces. A detailed description of the algorithm, referred to as PARTROTATION , is given in Figure 5. The algorithm shown runs APXEMD for the general case where $m < n$; when $m = n$, APXMATCH can be used instead.

Lemma 7. *Let $A = \{a_1, \dots, a_m\}$ and $B = \{b_1, \dots, b_n\}$ be two point sets with $m \leq n$ and $w_i = u_j = 1, i = 1, \dots, m, j = 1, \dots, n$. For any given $\epsilon \in (0, 1)$, $\text{PARTROTATION}(A, B, \epsilon)$ computes a rotation θ_{apx} such that $\text{EMD}(\theta_{\text{apx}}) \leq (1 + \epsilon)\text{EMD}(\theta_{\text{opt}})$ in $O((n^3m/\epsilon^3) \log^2(n/\epsilon) \log m)$ time. When $m = n$, the same approximation can be computed in $O((n^{7/2}/\epsilon^{5/2}) \log^6 n)$ time.*

PARTROTATION(A, B, ϵ):

1. Let $\alpha = \min_{i,j} \text{APXEMD}(A(\theta_{i \rightarrow j}), B, 1)$.
2. For each pair of points $a_i \in A$ and $b_j \in B$ do:
 - (a) Compute $D_{ij}(\alpha) = \{0, \alpha, m\alpha\} \cup D_{ij}^<(\alpha) \cup D_{ij}^>(\alpha)$.
 - (b) Let $\Theta_{ij} = \{\theta \in [0, 2\pi) \mid d_{ij}(\theta) \in D_{ij}(\alpha)\}$.
 - (c) For each rotation $\theta \in \Theta_{ij}$ compute a value $\widetilde{\text{EMD}}(\theta) = \text{APXEMD}(A(\theta), B, \epsilon/3)$.
3. Report the sample rotation θ_{apx} that minimizes $\widetilde{\text{EMD}}(\theta)$.

Fig. 5. Algorithm PARTROTATION(A, B, ϵ)

5.2 Rigid Motions

We can combine algorithm ROTATION with the 2-approximation algorithm for translations in Lemma 6 to get a $(4 + \epsilon)$ -approximation of the minimum EMD under rigid motions in the following way: for each point-to-point translation $\vec{t}_{i \rightarrow j}$, compute a $(2 + \epsilon/2)$ -approximation of the optimum EMD between $A(\vec{t}_{i \rightarrow j})$ and B under rotations about b_j . The minimum over all these approximations gives a $2(2 + \epsilon/2)$ -approximation of $\text{EMD}(\vec{t}_{\text{opt}}, \theta_{\text{opt}})$.

Lemma 8. *For any given $\epsilon > 0$, a $(4 + \epsilon)$ -approximation of the minimum EMD under rigid motions can be computed in $O((n^4 m^2 / \epsilon^2) \log^2(n/\epsilon))$ time.*

The $(2 + \epsilon)$ -approximation algorithm for rigid motions is based on similar ideas. According to Observation 1, there exist two points a_i, b_j whose distance at $I_{\vec{t}_{\text{opt}}, \theta_{\text{opt}}}$ is at most $\text{EMD}(\vec{t}_{\text{opt}}, \theta_{\text{opt}})$. We place a grid of suitable size around each $\vec{t}_{i \rightarrow j}$. For each grid point \vec{t}_g that is at most $\text{EMD}(\vec{t}_{\text{opt}}, \theta_{\text{opt}})$ away from $\vec{t}_{i \rightarrow j}$ we compute a $(2 + \epsilon)$ -approximation of the optimum EMD between $A(\vec{t}_g)$ and B under rotations about b_j . The minimum over all these approximations is within a factor of $(2 + \epsilon)$ of $\text{EMD}(\vec{t}_{\text{opt}}, \theta_{\text{opt}})$. Since we do not know $\text{EMD}(\vec{t}_{\text{opt}}, \theta_{\text{opt}})$, we compute a 6-approximation of it as shown above. Algorithm RIGIDMOTION is shown in Figure 6; for the partial assignment problem, a $(1 + \epsilon)$ -approximation can be achieved by running PARTROTATION instead of ROTATION.

Theorem 7. *For any given $\epsilon > 0$, RIGIDMOTION(A, B, ϵ) computes a rigid motion $I_{\vec{t}_{\text{apx}}, \theta_{\text{apx}}}$ such that $\text{EMD}(\vec{t}_{\text{apx}}, \theta_{\text{apx}}) \leq (2 + \epsilon) \text{EMD}(\vec{t}_{\text{opt}}, \theta_{\text{opt}})$ in $O((n^4 m^2 / \epsilon^4) \log^2(n/\epsilon))$ time. For the minimum cost partial assignment problem under rigid motions, a $(1 + \epsilon)$ -approximation can be computed in $O((n^4 m^2 / \epsilon^5) \log^2(n/\epsilon) \log m)$ time.*

As in the case of translations, for equal weight sets we need to search for the optimal translation only around $\vec{t}_{C(A) \rightarrow C(B)}$. We set the center of rotation to be $C(B)$. Computing the 6-approximation of $\text{EMD}(\vec{t}_{\text{opt}}, \theta_{\text{opt}})$ can be done simply by running ROTATION($A(\vec{t}_{C(A) \rightarrow C(B)}), B, 1$). Similarly, we need to run ROTATION($A(\vec{t}_g), B, \epsilon/3$) only for grid points \vec{t}_g that are close to $\vec{t}_{C(A) \rightarrow C(B)}$. For the assignment problem, instead of using ROTATION, we can use the version of PARTROTATION that runs APXMATCH to achieve a $(1 + \epsilon)$ -approximation.

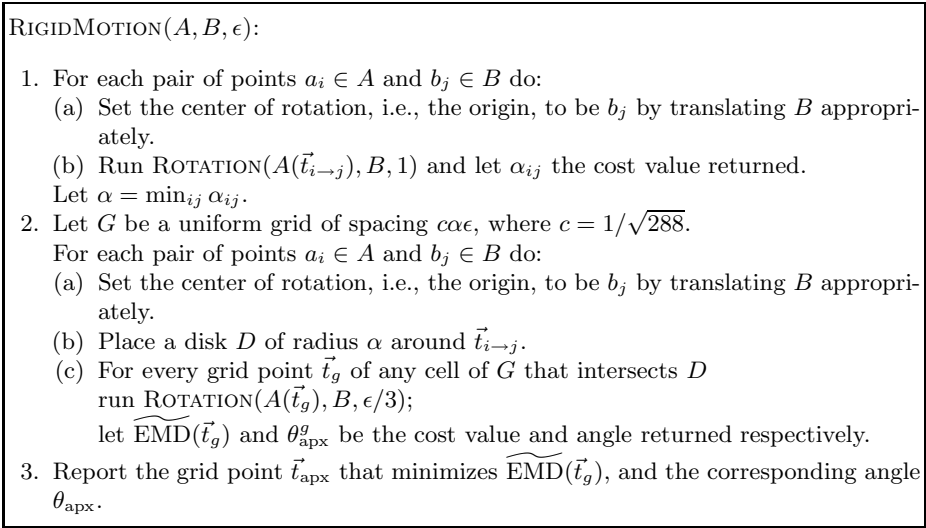


Fig. 6. Algorithm RIGIDMOTION(A, B, ϵ)

Theorem 8. *If A and B have equal total weights, then, for any given $\epsilon > 0$, a rigid motion $I_{\vec{t}_{\text{apx}}, \theta_{\text{apx}}}$ such that $\text{EMD}(\vec{t}_{\text{apx}}, \theta_{\text{apx}}) \leq (2 + \epsilon)\text{EMD}(\vec{t}_{\text{opt}}, \theta_{\text{opt}})$ can be computed in $O((n^3m/\epsilon^4) \log^2(n/\epsilon))$ time. For the minimum cost assignment problem under rigid motions a $(1 + \epsilon)$ -approximation can be computed in $O((n^{7/2}/\epsilon^{9/2}) \log^6 n)$ time.*

For the partial assignment problem under rigid motions, we can use the same arguments as in the translational case to convert algorithm RIGIDMOTION—that will now use PARTROTATION—into a randomized one where its two first steps are executed only for a random selection of $\Theta(n \log n)$ pairs of points. We conclude with the following:

Theorem 9. *Let $A = \{a_1, \dots, a_m\}$ and $B = \{b_1, \dots, b_n\}$ be two weighted point sets with $m \leq n$ and $w_i = u_j = 1, i = 1, \dots, m, j = 1, \dots, n$. For any given $\epsilon > 0$, a rigid motion $I_{\vec{t}_{\text{apx}}, \theta_{\text{apx}}}$ such that $\text{EMD}(\vec{t}_{\text{apx}}, \theta_{\text{apx}}) \leq (1 + \epsilon)\text{EMD}(\vec{t}_{\text{opt}}, \theta_{\text{opt}})$ can be computed in $O((n^4m/\epsilon^5) \log^2(n/\epsilon) \log n \log m)$ time. The algorithm succeeds with probability at least $1 - 2n^{-1}$.*

6 Concluding Remarks

We have presented polynomial-time $(1 + \epsilon)$ and $(2 + \epsilon)$ -approximation algorithms for the minimum Euclidean EMD under translations and rigid motions. We are currently working on extending the $(1 + \epsilon)$ -approximation for rotations to the general case of arbitrary weights. Another interesting and non-trivial task is to give lower and upper bounds of the complexity of the function $\text{EMD}(\vec{t}, \theta)$, i.e., the total number of its local optima.

Acknowledgments. The authors would like to thank Sariel Har-Peled for helpful discussions.

References

1. R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. *Network Flows: Theory, Algorithms and Applications*. Prentice-Hall, 1993.
2. H. Alt and L. Guibas. Discrete geometric shapes: Matching, interpolation, and approximation. In J.R. Sack and J. Urrutia, editors, *Handbook of Comp. Geom.*, pages 121–153. Elsevier Science Publishers B.V. North-Holland, 1999.
3. D.S. Atkinson and P.M. Vaidya. Using geometry to solve the transportation problem in the plane. *Algorithmica*, 13:442–461, 1995.
4. P. Bose, A. Maheshwari, and P. Morin. Fast approximations for sums of distances clustering and the Fermat-Weber problem. *Comp. Geom. Theory & Appl.*, 24:135–146, 2003.
5. P.B. Callahan and S.R. Kosaraju. Faster algorithms for some geometric graph problems in higher dimensions. In *Proc. of the 4th ACM-SIAM SODA*, pages 291–300, 1993.
6. R. Chandrasekaran and A. Tamir. Algebraic optimization: The Fermat-Weber location problem. *Math. Programming*, 46(2):219–224, 1990.
7. S. Cohen and L. Guibas. The Earth Mover's Distance under transformation sets. In *Proc. of the 7th IEEE ICCV*, pages 173–187, 1999.
8. P. Giannopoulos and R. C. Veltkamp. A pseudo-metric for weighted point sets. In *Proc. of the 7th ECCV*, volume 2352 of *LNCS*, pages 715–731, 2002.
9. K. Grauman and T. Darell. Fast contour matching using approximate Earth Mover's Distance. In *Proc. of the IEEE CVPR*, pages 220–227, 2004.
10. P. Indyk and N. Thaper. Fast image retrieval via embeddings. In *3rd Int. Workshop on Statistical and Computational Theories of Vision*, 2003.
11. O. Klein and R.C. Veltkamp. Approximation algorithms for the Earth Mover's Distance under transformations using reference points. Technical Report UU-CS-2005-003, IICS, Utrecht University, The Netherlands, 2005.
12. Q. Lv, M. Charikar, and K. Li. Image similarity search with compact data structures. In *Proc. of the 13th ACM CIKM*, pages 208–217, 2004.
13. D. Mumford. Mathematical theories of shape: Do they model perception? In *SPIE vol. 1570 Geometric Methods in Comp. Vision*, pages 2–10, 1991.
14. J. B. Orlin. A faster strongly polynomial minimum cost flow algorithm. *Operations Research*, 41(2):338–350, 1993.
15. Y. Rubner, C. Tomasi, and L.J. Guibas. The Earth Mover's Distance as a metric for image retrieval. *Int. Journal of Computer Vision*, 40(2):99–121, 2000.
16. R. Typke, P. Giannopoulos, R. C. Veltkamp, F. Wiering, and R. van Oostrum. Using transportation distances for measuring melodic similarity. In *Proc of 4th Int. Symp. on Music Inf. Retrieval (ISMIR)*, pages 107–114, 2003.
17. K.R. Varadarajan and P.K. Agarwal. Approximation algorithms for bipartite and non-bipartite matching in the plane. In *Proc. of the 10th ACM-SIAM SODA '99*, pages 805–814, 1999.

Small Stretch Spanners on Dynamic Graphs*

Giorgio Ausiello¹, Paolo G. Franciosa², and Giuseppe F. Italiano³

¹ Dipartimento di Informatica e Sistemistica,
Università di Roma “La Sapienza”, Roma, Italy
ausiello@dis.uniroma1.it

² Dipartimento di Statistica, Probabilità e Statistiche Applicate,
Università di Roma “La Sapienza”, Roma, Italy
paolo.franciosa@uniroma1.it

³ Dipartimento di Informatica, Sistemi e Produzione,
Università di Roma “Tor Vergata”, Roma, Italy
italiano@disp.uniroma2.it

Abstract. We present fully dynamic algorithms for maintaining 3- and 5-spanners of undirected graphs. For unweighted graphs we maintain a 3- or 5-spanner under insertions and deletions of edges in $O(n)$ amortized time per operation over a sequence of $\Omega(n)$ updates. The maintained 3-spanner (resp., 5-spanner) has $O(n^{3/2})$ edges (resp., $O(n^{4/3})$ edges), which is known to be optimal. On weighted graphs with d different edge cost values, we maintain a 3- or 5-spanner in $O(n)$ amortized time per operation over a sequence of $\Omega(d \cdot n)$ updates. The maintained 3-spanner (resp., 5-spanner) has $O(d \cdot n^{3/2})$ edges (resp., $O(d \cdot n^{4/3})$ edges). The same approach can be extended to graphs with real-valued edge costs in the range $[1, C]$. All our algorithms are deterministic and are substantially faster than recomputing a spanner from scratch after each update.

1 Introduction

Graph spanners arise in many applications, including communication networks, computational biology, computational geometry, robotics and distributed computing [1, 2, 3, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15]. Intuitively, a spanner of a graph is a subgraph that preserves approximate distances between all pairs of vertices. More formally, given $t \geq 1$, a t -spanner of a graph G is a subgraph S of G such that for each pair of vertices the distance in S is at most t times the distance in G : t is referred to as the *stretch factor* of the spanner. The best time bound for computing a t -spanner of a weighted graph with n vertices and m edges is $O(m + n)$, and is given by Baswana and Sen [4]. Their algorithm is randomized and computes spanners of size $O(t \cdot n^{1+2/(t+1)})$; a derandomization of this algorithm has been proposed in [16]. In the case of unweighted graphs, it is possible to compute a t -spanner in $O(m + n)$ time with a deterministic algorithm [20]. For weighted graphs, a deterministic algorithm is given in [1]; the best known implementation of this algorithm has running time $O(n^{2+2/(t+1)})$ [18].

* Partially supported by the Italian MIUR Project ALGO-NEXT “Algorithms for the Next Generation Internet and Web: Methodologies, Design and Experiments”.

Small stretch spanners offer a good compromise between sparsity and distance stretch: maintaining a t -spanner may be practical in the case of very large graphs, whose edges must be stored in external memory, while spanner edges could fit into main memory. A graph with million vertices could need TeraBytes of memory to store its edges, while the edges in its 3-spanner or 5-spanner only need order of GigaBytes, at the cost of a limited distance stretch. While there has been a lot of progress in the area of dynamic graph problems, to the best of our knowledge no fully dynamic algorithm for maintaining a t -spanner of a general weighted graph under edge insertions and/or deletions is known, and for this problem only partially dynamic solutions were announced in [4]. A related direction of research is concerned with the computation and maintenance of approximate distances, i.e., a query on the distance between two vertices is answered with a guaranteed approximation factor (see [16, 17] for recent results and references). Those algorithms typically require $\Omega(n^2)$ space, while in the case of t -spanners we are interested in a much sparser structure that still maintains distances in the original graph.

In this paper, we contribute a first step towards the maintenance of dynamic graph spanners by presenting a fully dynamic deterministic algorithm for maintaining 3- and 5-spanners of unweighted graphs. Our algorithm supports an intermixed sequence of $\Omega(n)$ edge insertions and deletions in $O(n)$ amortized time per operation. The maintained 3-spanner has $O(n^{3/2})$ edges, while the 5-spanner has $O(n^{4/3})$ edges. Wenger [19] showed how to build graphs with $\Theta(n^{3/2})$ edges having no cycles of length less than 5, or with $\Theta(n^{4/3})$ edges having no cycles of length less than 7: for such graphs, no proper subgraphs preserve distances within stretch factor 3 (resp. 5). This implies that the size of our spanners is asymptotically optimal. The same approach can be extended to weighted graphs with d different edge cost values. On a sequence of $\Omega(d \cdot n)$ intermixed edge insertions and deletions the amortized time per operation is still $O(n)$. The maintained 3-spanner has $O(d \cdot n^{3/2})$ edges, while the 5-spanner has $O(d \cdot n^{4/3})$ edges. This is optimal for constant d . On graphs with real-valued edge costs in $[1, C]$, for $t > 3$ we can maintain a t -spanner with $O(n^{3/2} \cdot \log_{t/3} C)$ edges in $O(n)$ amortized time per operation over a sequence of $\Omega(n \cdot \log_{t/3} C)$ edge insertions and edge deletions. For $t > 5$, a t -spanner with $O(n^{4/3} \cdot \log_{t/5} C)$ edges can be maintained in $O(n)$ amortized time per operation over a sequence of $\Omega(n \cdot \log_{t/5} C)$ edge insertions and edge deletions. All our algorithms require $O(n^2)$ worst-case space, are deterministic and are substantially faster than re-computing a spanner from scratch. To achieve our results, we dynamize the static randomized technique of Baswana and Sen [4], and make the resulting algorithm deterministic rather than randomized. Our algorithms use simple data structures, and thus seem amenable to practical implementations.

2 Definitions

We assume that the reader is familiar with the standard graph terminology, as contained for instance in [8]. Let $G = (V, E)$ be an undirected graph, with V

being the set of vertices and E the set of edges. Throughout the paper, we denote by n the number of vertices and by m the number of edges in a graph. If the graph is weighted, there is a real-valued cost $c(e) \geq 0$ associated with each edge $e \in E$. Given a vertex x , its *neighborhood* is the set $N(x) = \{x\} \cup \{y \mid (x, y) \in E\}$ (note that $x \in N(x)$ by definition). Given two vertices $u, v \in V$, a *path* π in $G = (V, E)$ *connecting vertex u to vertex v* is a sequence of vertices $u = v_0, v_1, \dots, v_\ell = v$ such that $(v_{i-1}, v_i) \in E$, for $0 < i \leq \ell$. We say that each edge (v_{i-1}, v_i) is in path π , for $0 < i \leq \ell$. The *length of a path* π is given by the number of edges in π . If the graph is weighted, the *cost of a path* π is the sum of the costs of edges in π : $c(\pi) = \sum_{i=1}^{\ell} c(v_{i-1}, v_i)$. In the case of unweighted graphs the cost of a path is simply its length. The *distance* $dist_G(u, v)$ from u to v in G is given by the minimum cost of a path in G from u to v (or $+\infty$ if there is no such path). A *shortest path* from u to v is then defined as any path π from u to v with $c(\pi) = dist_G(u, v)$. A graph $G' = (V', E')$ is a *subgraph* of graph $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. Given a graph G , a t -*spanner* S is a subgraph of G that preserves distances up to a factor t (the *stretch factor*). More formally,

Definition 1. Let $G = (V, E)$ be a weighted graph, and let t be a real value, with $t \geq 1$. A t -spanner of G is a graph $S = (V, E')$ with $E' \subseteq E$ such that:

$$\forall u, v \in V \quad dist_S(u, v) \leq t \cdot dist_G(u, v).$$

The following lemma is due to Peleg and Shäffer [13]:

Lemma 1. [13] A subgraph $S = (V, E')$ of $G = (V, E)$, is a t -spanner of G if and only if the following holds:

$$\forall (x, y) \in E \quad dist_S(x, y) \leq t \cdot c(x, y). \tag{1}$$

Definition 2. Let x_1, \dots, x_k , with $k \geq 1$, be distinct vertices in V , and let $\Gamma = \{Cl(x_1), \dots, Cl(x_k)\}$ be a family of subsets of V . Given a real number $\ell \geq 1$, Γ is an ℓ -clustering of $G = (V, E)$ if the following properties hold:

1. $Cl(x_i) \subseteq N(x_i)$ for $1 \leq i \leq k$;
2. $Cl(x_i) \cap Cl(x_j) = \emptyset$, for each $i \neq j$;
3. $\bigcup_{i=1}^k Cl(x_i) = \bigcup_{i=1}^k N(x_i)$;
4. $|Cl(x_i)| \geq \ell$ for $1 \leq i \leq k$.

Set $Cl(x_i)$ is called *cluster*, and x_i is denoted as its *center*.

According to the previous definition, the center x_i of cluster $Cl(x_i)$ may belong to another cluster $Cl(x_j)$, with $i \neq j$. As a special case, we define an *empty clustering* to be a clustering with no clusters. We assume that the empty clustering is an ℓ -clustering, for any ℓ . An example of 5-clustering is shown in the left part of Figure 1. Given an ℓ -clustering, a vertex is called *clustered* if it belongs to a cluster, and *free* otherwise; if vertex y is clustered, $center(y)$ denotes the center of the cluster containing y . For each $v \in V$, we define its *free neighborhood* $FN(v)$ as $FN(v) = N(v) \setminus \left(\bigcup_{i=1}^k Cl(x_i)\right)$. Note that an ℓ -clustering contains at most n/ℓ clusters, each of size at least ℓ . We say that an ℓ -clustering is *maximal*

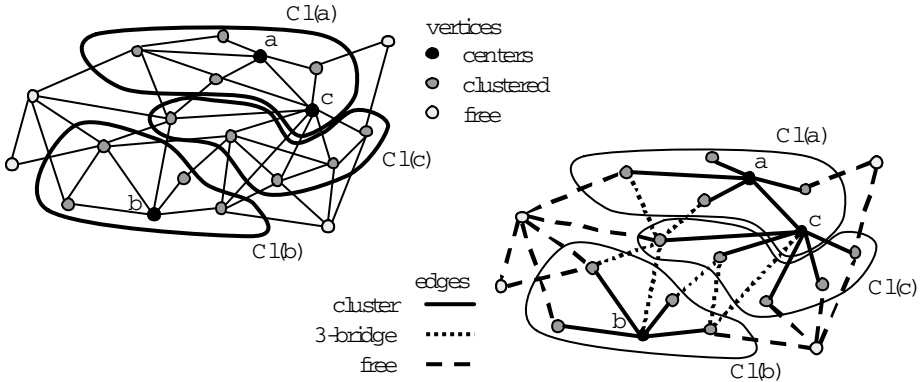


Fig. 1. A 5-clustering of a graph and an associated 3-spanner

if $|FN(v)| < 2\ell$, for each $v \in V$. We remark that our definition of clustering is more strict than the one given in [4]. In the case of unweighted graphs, the construction of spanners starting from our definition of clustering is simpler and deterministic.

3 Clusterings and Spanners

In this section we show how small stretch spanners of unweighted graphs can be obtained from proper ℓ -clusterings. In particular, we describe how to produce a 3-spanner from a $n^{1/2}$ -clustering, and a 5-spanner from a $n^{1/3}$ -clustering.

3.1 Clusterings for 3-Spanners

Definition 3. Given a $n^{1/2}$ -clustering Γ of $G = (V, E)$, a subgraph $G' = (V, E')$ of G is 3-compatible with Γ if E' is the union of the following sets of edges:

- Cluster Edges:** all edges (x, y) such that y is clustered and $x = \text{center}(y)$;
- Free Edges:** all edges $(x, y) \in E$ such that either x or y is a free vertex;
- 3-Bridge Edges:** for each cluster $Cl(x_i)$ and each vertex $y \in (Cl(x_j) \setminus \{x_i\})$, with $x_j \neq x_i$, one arbitrary edge $(x, y) \in E$ such that $x \in Cl(x_i)$, if one exists. We say that edge (x, y) connects vertex y to cluster $Cl(x_i)$.

Theorem 1. Given a graph $G = (V, E)$ and a $n^{1/2}$ -clustering Γ , if $G' = (V, E')$ is a subgraph of G 3-compatible with Γ then G' is a 3-spanner of G .

Proof. We show that for any edge $(a, b) \in E$ there is a path of length at most 3 in G' . There can be only three cases, depending on a and b .

(Case 1) Both a and b belong to the same cluster: in this case either one among a and b is the center of the cluster, and thus (a, b) is a cluster edge in G' , or a third vertex x is the center of the cluster, and thus (a, x) and (x, b) are cluster edges in G' .

(Case 2) Vertices a and b belong to different clusters. Let $a \in Cl(x_i)$ and $b \in Cl(x_j)$: in such a case there must be a 3-bridge edge (b, y) , where $y \in Cl(x_i)$. If $y \neq a$, then the cluster edges provide a path of length at most 2 from y to a , thus giving a path of length at most 3 from a to b in G' .

(Case 3) Either a or b is a free vertex: in this case (a, b) is a free edge in G' .

Due to Theorem 1, we will refer to a subgraph of G 3-compatible with an $n^{1/2}$ -clustering Γ as a *3-spanner associated with Γ* . The right side of Figure 1 shows a 3-spanner associated with the 5-clustering on the left. If the $n^{1/2}$ -clustering is maximal, we can prove that any associated 3-spanner is sparse:

Theorem 2. *A 3-spanner $G' = (V, E')$ associated to a maximal $n^{1/2}$ -clustering contains $O(n^{3/2})$ edges.*

Proof. There are at most n cluster edges, since each vertex can be in at most one cluster. There is at most one 3-bridge edge for each possible pair $(x, Cl(x_i))$, where $x \in N$ and $Cl(x_i) \in \Gamma$: since there are at most $n^{1/2}$ clusters, there are at most $n^{3/2}$ 3-bridge edges. We finally bound the number of free edges: since Γ is maximal, we have $|FN(v)| < 2n^{1/2}$ for any vertex v . Hence, the total number of free edges is at most $2 \cdot n^{3/2}$.

3.2 Clusterings for 5-Spanners

Definition 4. *Given an $n^{1/3}$ -clustering Γ of $G = (V, E)$, a subgraph $G' = (V, E')$ of G is 5-compatible with Γ if E' is the union of the following sets of edges:*

Cluster Edges: *all edges (x, y) such that y is clustered and $x = center(y)$;*

Free Edges: *all edges $(x, y) \in E$ such that either x or y is a free vertex;*

5-Bridge Edges: *for each pair of clusters $Cl(x_i), Cl(x_j)$, with $x_i \neq x_j$, one arbitrary edge $(x, y) \in E$ such that $x \in Cl(x_i)$ and $y \in Cl(x_j)$, if one exists.*

We say that edge (x, y) connects clusters $Cl(x_i)$ and $Cl(x_j)$.

We remark that the only difference with Definition 3 lies in the set of bridge edges.

Theorem 3. *Given a graph $G = (V, E)$ and an $n^{1/3}$ -clustering Γ , if $G' = (V, E')$ is a subgraph of G 5-compatible with Γ then G' is a 5-spanner of G .*

Proof. We show that for any edge $(a, b) \in E$ there is a path of length at most 5 in G' . There can be only three cases, depending on a and b .

(Case 1) Both a and b belong to the same cluster: in this case either one among a and b is the center of the cluster, and thus (a, b) is a cluster edge in G' , or a third vertex x is the center of the cluster, and thus (a, x) and (x, b) are cluster edges in G' .

(Case 2) Vertices a and b belong to different clusters. Let $a \in Cl(x_i)$ and $b \in Cl(x_j)$: in such a case there must be a 5-bridge edge (x, y) , where $x \in Cl(x_i)$ and $y \in Cl(x_j)$. Since cluster edges provide paths of length at most 2 from a to x and from y to b , we have a path of length at most 5 from a to b in G' .

(Case 3) Either a or b is a free vertex: in this case (a, b) is a free edge in G' .

Due to Theorem 3, we will refer to a subgraph of G 5-compatible with an $n^{1/3}$ -clustering Γ as a *5-spanner associated with Γ* . If the $n^{1/3}$ -clustering is maximal, we can prove that any associated 5-spanner is sparse:

Theorem 4. *A 5-spanner $G' = (V, E')$ associated to a maximal $n^{1/3}$ -clustering contains $O(n^{4/3})$ edges.*

Proof. There are at most n cluster edges, since each vertex can be in at most one cluster. There is at most one 5-bridge edge for each possible pair of clusters: since there are at most $n^{2/3}$ clusters, there are at most $n^{4/3}$ 5-bridge edges. We finally bound the number of free edges: since Γ is maximal, we have $|FN(v)| < 2 \cdot n^{1/3}$ for any vertex v . Hence, the total number of free edges is at most $2 \cdot n^{4/3}$.

4 Decremental Algorithms for 3- and 5-Spanners

The main contribution of this paper is to provide a fully dynamic deterministic algorithm for maintaining 3-spanners and 5-spanners of unweighted graphs. The construction and maintenance of our spanners is based on the maintenance of a maximal ℓ -clustering. Starting from any ℓ -clustering, a maximal ℓ -clustering can be obtained with a simple greedy algorithm, as illustrated in Figure 2. We observe that the same algorithm can be used for computing a maximal ℓ -clustering from scratch, starting from the initial empty clustering $\Gamma = \emptyset$.

Theorem 5. *Procedure MaximalCluster computes a maximal ℓ -clustering of G .*

Proof. The fact that Procedure MaximalCluster computes an ℓ -clustering can be easily seen by induction on the number of clusters added to Γ . We assume Γ is an ℓ -clustering before applying the procedure. In particular, this is true for the empty clustering. We now show that any time a new cluster $Cl(x)$ is added to Γ all the properties of Definition 2 are maintained:

- Property 1: $Cl(x) = FN(x) \subseteq N(x)$ by definition of free neighborhood;
- Property 2: $Cl(x)$ only contains free vertices, hence it is disjoint from all existing clusters;
- Property 3: all free vertices in $N(x)$ are included in $Cl(x)$;
- Property 4: $Cl(x)$ has size at least $2 \cdot \ell$.

Procedure MaximalCluster

input: graph $G = (V, E)$
 ℓ -clustering Γ

output: maximal ℓ -clustering Γ

1. **while** there is a vertex x with $|FN(x)| \geq 2 \cdot \ell$
2. make x a center
3. make $Cl(x) = FN(x)$
4. add $Cl(x)$ to Γ

Fig. 2. Procedure MaximalCluster

Moreover, Γ is maximal, since at the end there is no vertex x with $|FN(x)| \geq 2 \cdot \ell$.

Procedure `MaximalCluster` builds an ℓ -clustering Γ by adding one cluster at a time. We now show how to maintain a 3-spanner associated with Γ when new clusters are added, in the case $\ell = n^{1/2}$. For each vertex x we maintain the following simple data structures, that represent the current spanner plus some auxiliary information:

- The number $|FN(x)|$ of free vertices in $N(x)$.
- For each cluster $Cl(c)$ such that $x \notin Cl(c)$, the list of edges $e = (x, z) \in E$ such that $z \in Cl(c)$. This list represents the candidate edges for connecting x to $Cl(c)$; the 3-bridge edge in the spanner connecting x to $Cl(c)$, for each cluster, is the first edge in the list.
- A flag indicating whether x is clustered. If x is clustered, we maintain a reference to the cluster containing x , a reference to $center(x)$, and the list of all 3-bridge edges (y, x) incident to x , connecting any vertex y to the cluster containing x .
- A flag indicating whether x is a center. If x is a center we maintain the list of vertices in $Cl(x)$. This list implicitly represents all cluster edges of $Cl(x)$.
- The list of all free edges incident to x .

Theorem 6. *It is possible to maintain a 3-spanner associated to a $n^{1/2}$ -clustering, under the addition of new clusters C_1, C_2, \dots, C_h as described in Procedure `MaximalCluster`, in a total of $O(\sum_{i=1}^h \sum_{y \in C_i} |N(y)|)$ worst-case time.*

Proof. Assume that cluster edges, free edges and 3-bridge edges are correct before adding clusters to Γ . A vertex x having $|FN(x)| \geq 2n^{1/2}$ can be found in constant time, provided that the set of vertices x having $|FN(x)| \geq 2n^{1/2}$ is maintained throughout. We now determine the cost of updating the spanner for the different classes of spanner edges. Assume that the new cluster $Cl(x) = FN(x)$ is added to Γ :

- (i) **Cluster Edges:** we add all edges (x, y) , with $y \in Cl(x)$. This can be done in $O(|Cl(x)|)$ worst-case time;
- (ii) **Free Edges:** vertices in $Cl(x)$ are no longer free. For each vertex $y \in Cl(x)$ we explore vertices in $N(y)$: for each vertex $z \in N(y)$ we decrement $|FN(z)|$, moreover, if z is clustered we remove (z, y) from the set of free edges. This can be done in $O(\sum_{i=1}^h \sum_{y \in C_i} |N(y)|)$ worst-case time;
- (iii) **3-Bridge Edges:** each vertex $v \in V \setminus Cl(x)$ must be connected to $Cl(x)$ via a new 3-bridge edge, if one exists. For each vertex $y \in Cl(x)$ we scan vertices $z \in N(y)$: if z is not already connected to $Cl(x)$ then (y, z) becomes a 3-bridge edge. This can be done in $O(\sum_{i=1}^h \sum_{y \in C_i} |N(y)|)$ worst-case time.

All the above operations can be implemented in a total of $O(\sum_{i=1}^h \sum_{y \in C_i} |N(y)|)$ worst-case time.

Theorem 7. *Given a graph G , Procedure `MaximalCluster` computes in $O(m+n)$ worst-case time a 3-spanner of G having $O(n^{3/2})$ edges.*

Proof. We apply Procedure MaximalCluster starting from $\Gamma = \emptyset$. At the beginning, all vertices are free, there are no cluster edges and 3-bridge edges, and the set of free edges is E . By Theorems 1, 2, 5, and 6, we can state that a 3-spanner is computed in $O(m + n)$ worst-case time.

We now show how to deal with edge deletions. When an edge is deleted from the graph we might have to update the clustering and the associated spanner. We first analyze how the clustering can be updated, and then describe how to update the associated spanner. A clustering Γ is affected by the deletion of edge $e = (x, y)$ only if e is a cluster edge; w.l.o.g. assume that x is a center and $y \in Cl(x)$. In this case y can no longer be in $Cl(x)$, due to Property 1 of Definition 2. A more substantial change is due to Property 4 of Definition 2, in the case where, after removing y from $Cl(x)$, this set becomes too small to be a cluster: in this case $Cl(x)$ is removed from Γ . In both cases, in order to preserve Property 3 of Definition 2, we must add y (resp., each vertex $v \in Cl(x)$ in the case $Cl(x)$ is removed from Γ) to some other cluster in Γ , whenever possible. If there are no centers in $N(y)$ (resp., in $N(v)$ for any vertex $v \in Cl(x)$) then y becomes a free vertex. The update algorithm is described by Procedure DeleteClusterEdge in Figure 3. We now show how to update the associated spanner after the deletion of an edge $e = (x, y)$. We distinguish four different cases, depending on the type of edge being deleted:

e is not in the spanner: the spanner $G' = (V, E')$ does not change. Since e is not in the spanner, both x and y must be clustered vertices. Neither $FN(x)$ or $FN(y)$ change their size;

Procedure DeleteClusterEdge

input: graph $G = (V, E)$

$n^{1/2}$ -clustering Γ of $G = (V, E)$

a cluster edge $e = (x, y)$, where $x = center(y)$

output: $n^{1/2}$ -clustering Γ of $G = (V, E \setminus \{e\})$

1. **if** $|Cl(x)| > n^{1/2}$
2. remove y from $Cl(x)$
3. **if** y is the center of a cluster in Γ
4. add y to $Cl(y)$
5. **else if** there exists a center $c \in N(y)$ in Γ
6. add y to $Cl(c)$
7. **else**
8. // vertex x is no longer a center //
9. remove $Cl(x)$ from Γ
10. **for each** $v \in Cl(x)$
11. **if** v is the center of a cluster in Γ
12. add v to $Cl(v)$
13. **else if** there exists a center $c \in N(v)$ in Γ
14. add v to $Cl(c)$

Fig. 3. Deleting a cluster edge

e is a free edge: at least one among x and y is free. Edge e is removed from the spanner; the sizes of $FN(x)$ and/or $FN(y)$ are decremented accordingly;

e is a 3-bridge edge: w.l.o.g. assume that e connects y to $x \in Cl(z)$ (the case where e also connects x to $y \in Cl(w)$ is dealt with analogously): find another edge f connecting y to a vertex $w \in Cl(z)$ (if it exists), and add f to the set of 3-bridge edges;

e is a cluster edge: assume that x is a center and $y \in Cl(x)$, and that the clustering is updated according to Procedure DeleteClusterEdge. The spanner is updated as follows.

- If $|Cl(x)|$ remains at least $n^{1/2}$, $Cl(x)$ is still a cluster, and vertex x is still its center:
 - e is no longer a cluster edge;
 - replace each 3-bridge edge (z, y) connecting a vertex z to $Cl(x)$ via y by a new bridge, if one exists. To this aim, for each vertex $z \in N(y)$, if (z, y) is a 3-bridge edge we remove (z, y) from the set of 3-bridge edges and search for a new 3-bridge edge (z, w) , with $w \in Cl(x)$;
 - if y is added to $Cl(c)$ (where possibly $c = y$):
 - * (c, y) becomes a cluster edge (provided that $c \neq y$);
 - * in case some vertex w was not connected to $Cl(c)$, because there were no edges $(w, z) \in E$ with $z \in Cl(c)$, but $(w, y) \in E$, it is now possible to connect w to $Cl(c)$ via y . This can be done by detecting, for each $w \in N(y)$, whether w is already connected to $Cl(c)$ and, if not, adding edge (w, y) to the set of 3-bridge edges.
 - in case y is now free, for each $z \in N(y)$ we increase $|FN(z)|$ and add (z, y) to the set of free edges.
- If $|Cl(x)|$ drops below $n^{1/2}$, $Cl(x)$ can no longer be a cluster, and thus vertex x can no longer be a center:
 - remove all 3-bridge edges connecting vertices to $Cl(x)$ —provided that the same edge does not connect a vertex to any other cluster;
 - for each $v \in Cl(x)$ we do the following:
 - * (x, v) is no longer a cluster edge;
 - * if v is added to $Cl(c)$ (where possibly $c = v$):
 - (c, v) becomes a cluster edge (provided that $c \neq v$);
 - in case some vertex w was not connected to $Cl(c)$, because there were no edges $(w, z) \in E$ with $z \in Cl(c)$, but $(w, v) \in E$, it is now possible to connect w to $Cl(c)$ via v . As above, this can be done by detecting, for each $w \in N(v)$, whether w is already connected to $Cl(c)$ and, if not, adding edge (w, v) to the set of 3-bridge edges.
 - * in case v is now free, for each $z \in N(v)$ we increase $|FN(z)|$ and add (z, v) to the set of free edges.

This restores a 3-spanner associated to the $n^{1/2}$ -clustering Γ . After this, in order to obtain a maximal $n^{1/2}$ -clustering, we apply Procedure MaximalCluster.

Theorem 8. *Procedure DeleteClusterEdge updates a $n^{1/2}$ -clustering and the associated 3-spanner of G under the deletion of edge (x, y) in either $O(|N(x)| + |N(y)|)$ worst-case time, if no cluster is removed from Γ , or in $O(\sum_{v \in Cl(x)} |N(v)|)$ worst-case time, if (x, y) is a cluster edge and cluster $Cl(x)$ is removed from Γ .*

Proof. If e is not in the spanner or it is a free edge, the above algorithm requires constant time. In the case (x, y) is a 3-bridge edge we need $O(|N(x)| + |N(y)|)$ worst-case time for exploring $N(x)$ and/or $N(y)$. If (x, y) is a cluster edge, we distinguish two cases: if $Cl(x)$ is still a cluster, we only explore $N(x)$ and $N(y)$. Otherwise, if $Cl(x)$ is destroyed, we explore the neighborhood of all vertices in $Cl(x)$, in $O(\sum_{v \in Cl(x)} |N(v)|)$ worst-case time.

We now sketch how to build and maintain 5-spanners. With respect to 3-spanners, we need to maintain an $n^{1/3}$ -clustering rather than a $n^{1/2}$ -clustering, and consequently the only change in the data structures consists in keeping track of 5-bridge edges instead of 3-bridge edges. We omit here for lack of space the details of the data structure. The following theorems are the analogues of Theorems 6, 7 and 8.

Theorem 9. *It is possible to maintain a 5-spanner associated to an $n^{1/3}$ -clustering, under the addition of new clusters C_1, C_2, \dots, C_h as described in Procedure MaximalCluster, in a total worst-case time $O(\sum_{i=1}^h \sum_{y \in C_i} |N(y)|)$.*

Theorem 10. *Given a graph G , Procedure MaximalCluster computes in $O(m + n)$ worst-case time a 5-spanner of G having $O(n^{4/3})$ edges.*

Theorem 11. *Procedure DeleteClusterEdge updates an $n^{1/3}$ -clustering and the associated 5-spanner of G under the deletion of edge $e = (x, y)$ in either $O(|N(x)| + |N(y)|)$ worst-case time, if no cluster is removed from Γ , or in $O(\sum_{v \in Cl(x)} |N(v)|)$ worst-case time, if (x, y) is a cluster edge and cluster $Cl(x)$ is removed from Γ .*

We next analyze the amortized complexity of edge deletions. An edge deletion that does not modify the clustering is performed in $O(n)$ worst-case time. If a vertex is removed from a cluster, the spanner is maintained in $O(n)$ worst-case time (by Theorems 8 and 11), plus possibly the time needed to build a new cluster. Since the size of the new cluster is $O(\ell)$, the new cluster can be built in $O(\ell \cdot n)$ time, due to Theorems 6 and 9. An edge deletion that destroys a cluster is performed in $O(\ell \cdot n)$ worst-case time (by Theorems 8 and 11), plus possibly the time needed to build the new clusters. Again, each new cluster has size $O(\ell)$, and by Theorems 6 and 9 the time needed is $O(\ell \cdot n)$ for each new cluster. Hence, for each edge deletion we need a total of $O(n)$ time plus $O(\ell \cdot n)$ time for each cluster that appears or disappears from the clustering.

In order to bound the number of clusters that appear or disappear during a sequence of edge deletions, we consider how the cluster sizes can be affected by edge deletions. If the edge deletion does not destroy any cluster, then the size of at most one cluster is decreased by one (this happens when a cluster edge

is deleted). Otherwise, only one cluster may be destroyed, and the size of the other clusters does not decrease. During a sequence of edge deletions, the set of destroyed clusters consists at most of the initial clusters, plus some of the clusters created during the sequence of edge deletions. The initial clusters are at most n/ℓ . The initial size of a cluster C created during the sequence is at least $2 \cdot \ell$, and C is destroyed only if its size decreases to less than ℓ during the update sequence. By the above arguments at least ℓ deletions are needed in order to shrink C from its initial size (at least $2 \cdot \ell$) to ℓ . In summary, if the update sequence has length σ , at most $n/\ell + \sigma/\ell$ clusters may be destroyed overall. The number of clusters created during the sequence is at most the number of clusters at the end of the sequence plus the number of destroyed clusters, that is at most $2 \cdot n/\ell + \sigma/\ell$. By Theorems 8 and 11, the total cost over the sequence is thus $O(\sigma \cdot n + (n/\ell + \sigma/\ell) \cdot \ell \cdot n) = O(\sigma \cdot n + n^2)$. Hence, we can state the following:

Theorem 12. *A 3-spanner (resp. a 5-spanner) of an unweighted graph can be maintained in $O(\sigma \cdot n + n^2)$ total time over a sequence of σ edge deletions. The spanner has $O(n^{3/2})$ (resp. $O(n^{4/3})$) edges. This gives $O(n)$ amortized time per operation over a sequence of $\Omega(n)$ edge deletions.*

5 Fully Dynamic 3- and 5-Spanners for Unweighted Graphs

To make the decremental algorithms of Section 4 fully dynamic, we deal with edge insertions in a lazy fashion. Inserted edges are kept in a set E'' , and our 3-spanner consists of the edges induced by the clustering (see Definition 3) plus the edges in E'' . When inserting an edge, we do not update the clustering and the associated spanner. Only when the size of E'' exceeds the size of the spanner, i.e., $n^{3/2}$ or $n^{4/3}$, a new clustering and the associated spanner are built from scratch using Procedure MaximalCluster starting from the empty clustering, and E'' is set to the empty set. This gives the following theorem:

Theorem 13. *A 3-spanner or a 5-spanner of an unweighted graph can be maintained in $O(\sigma \cdot n + n^2)$ total time over a sequence of σ intermixed edge insertions and edge deletions. This gives $O(n)$ amortized time per operation on a sequence of $\Omega(n)$ edge insertions and edge deletions.*

Proof. If the sequence contains less than $n^{3/2}$ (resp., $n^{4/3}$) edge insertions, then the spanner is never rebuilt from scratch, and the theorem derives from Theorem 12. Otherwise, we must rebuild from scratch the spanner, taking $O(n^2)$ worst-case time (see Theorems 7 and 10), but this cost can be amortized over a sequence of length $\Omega(n^{3/2})$ (resp., $\Omega(n^{4/3})$), giving an amortized cost of $O(n^{1/2})$ (resp., $O(n^{2/3})$) per operation. Hence, the amortized cost is dominated by the cost of edge deletions, which is $O(n)$ by Theorem 12.

The algorithms in this paper can be extended to weighted graphs, as shown in the following theorems.

Theorem 14. *A 3-spanner (resp., a 5-spanner) of a graph with d different edge costs can be maintained in $O(n)$ amortized time per operation over a sequence of $\Omega(d \cdot n)$ edge insertions and deletions. The spanner has $O(d \cdot n^{3/2})$ (resp., $O(d \cdot n^{4/3})$) edges.*

Theorem 15. *For any $t > 3$ (resp. $t > 5$), a t -spanner of a graph with real-valued edge costs in $[1, C]$ can be maintained in $O(n)$ amortized time per operation over a sequence of $\Omega(n \cdot \log_{t/3} C)$ (resp. $\Omega(n \cdot \log_{t/5} C)$) edge insertions and edge deletions. The spanner has $O(n^{3/2} \cdot \log_{t/3} C)$ (resp. $O(n^{4/3} \cdot \log_{t/5} C)$) edges.*

References

- [1] I. Althofer, G. Das, D. P. Dobkin, D. Joseph, and J. Soares. On sparse spanners of weighted graphs. *Discrete & Computational Geometry*, 9:81–100, 1993.
- [2] B. Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, 1985.
- [3] H.-J. Bandelt and A. Dress. Reconstructing the shape of a tree from observed dissimilarity data. *Adv. Appl. Math.*, 7:309–343, 1986.
- [4] S. Baswana and S. Sen. A simple linear time algorithm for computing $(2k - 1)$ -spanner of $O(n^{1+1/k})$ size for weighted graphs. In *Proc. 30th ICALP*, 384–396.
- [5] L. Cai. NP-completeness of minimum spanner problems. *Discr. Appl. Math. and Comb. Oper. Research and Comp. Sci.*, 48(2):187–194, 1994.
- [6] L. Cai and J. M. Keil. Degree-bounded spanners. *Parallel Processing Letters*, 3:457–468, 1993.
- [7] L. P. Chew. There are planar graphs almost as good as the complete graph. *Journal of Computer and System Sciences*, 39(2):205–219, 1989.
- [8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.
- [9] G. Das and D. Joseph. Which triangulations approximate the complete graph? In *Proc. Int. Symp. on Optimal Algorithms*, 168–192.
- [10] D. Dobkin, S. J. Friedman, and K. J. Supowit. Delaunay graphs are almost as good as complete graphs. *Discrete Comput. Geom.*, 5:399–407, 1990.
- [11] A. L. Liestman and T. Shermer. Additive graph spanners. *Networks*, 23:343–364, 1993.
- [12] A. L. Liestman and T. Shermer. Grid spanners. *Networks*, 23:122–133, 1993.
- [13] D. Peleg and A. Schäffer. Graph spanners. *J. of Graph Theory*, 13:99–116, 1989.
- [14] D. Peleg and J. D. Ullman. An optimal synchronizer for the hypercube. *SIAM Journal on Computing*, 18(4):740–747, 1989.
- [15] D. Richards and A. L. Liestman. Degree-constrained pyramid spanners. *Journal of Parallel and Distributed Computing*, 25:1–6, 1995.
- [16] L. Roditty and M. Thorup and U. Zwick. Deterministic constructions of approximate distance oracles and spanners. In *Proc. ICALP 2005*, to appear.
- [17] L. Roditty and U. Zwick. Dynamic approximate all-pairs shortest paths in undirected graphs. In *Proc. FOCS 2004*, 499–508.
- [18] L. Roditty and U. Zwick. On dynamic shortest paths problems. In *Proc. ESA 2004*, 580–591.
- [19] R. Wenger. Extremal graphs with no C^4 's, C^6 's, or C^{10} 's. *J. Combin. Theory Ser. B*, 52:113–116, 1991.
- [20] U. Zwick. Personal communication.

An Experimental Study of Algorithms for Fully Dynamic Transitive Closure*

Ioannis Krommidas and Christos Zaroliagis

Computer Technology Institute, P.O. Box 1122, 26110 Patras, Greece;
and Dept of Computer Engineering and Informatics,
University of Patras, 26500 Patras, Greece
{krommudi, zaro}@ceid.upatras.gr

Abstract. We have conducted an extensive experimental study on some recent, theoretically outstanding, algorithms for fully dynamic transitive closure along with several variants of them, and compared them to pseudo fully dynamic and simple-minded algorithms developed in a previous study. We tested and compared these implementations on random inputs, synthetic (worst-case) inputs, and on inputs motivated by real-world graphs. Our experiments reveal that some of the fully dynamic algorithms can really be of practical value in many situations.

1 Introduction

The *transitive closure* (or *reachability*) problem in a digraph G consists in finding whether there is a directed path between any two vertices in G . In this paper, we are concerned with the dynamic version of this (fundamental and extensively studied) problem, namely with the maintenance of transitive closure when G undergoes a sequence of edge insertions and deletions. An algorithm is called *fully dynamic* if it supports both edge insertions and deletions, and *partially dynamic* if either insertions or deletions (but not both) are supported; in the former case the algorithm is called *incremental*, while in the latter *decremental*.

Recently, we have witnessed a number of important theoretical breakthroughs regarding fully dynamic transitive closure [4,11,10,12,15–17]. These fully dynamic algorithms can be roughly divided into two categories: those using combinatorial techniques [10,12,15–17] and those which do not exclusively use such techniques [4,11,16]. Moreover, some of these algorithms apply to DAGs, while some others to general digraphs. In this paper, we concentrate on *fully dynamic algorithms* for maintaining the transitive closure of *general digraphs* only.

Despite the above theoretical progress, we are not aware of any practical assessment of any of the aforementioned algorithms. Our prime goal is to advance our knowledge on the practical aspects of this recent and important theoretical work. Previous experimental studies regarding maintenance of transitive closure [1,6] have mostly focussed on the assessment of partially dynamic algorithms.

* This work was partially supported by the IST Programme (6th FP) of EC under contract No. IST-2002-001907 (integrated project DELIS).

The only experimental comparison regarding fully dynamic transitive closure was made by Frigioni et al [6], where a fully dynamic algorithm in [7] was compared to a new algorithm, called **Ital-Gen**, developed in [6]. **Ital-Gen** is based on a hybridization and extension of Italiano's partially dynamic algorithms [8,9], works in a fully dynamic setting, but update times can be analyzed and bounded only for partially dynamic operation sequences. We shall refer to such algorithms as *pseudo fully dynamic*. The experiments in [6] showed that **Ital-Gen** was considerably faster than the fully dynamic algorithm in [7].

In this work, we have implemented and experimentally compared all the aforementioned combinatorially based fully dynamic algorithms [10,12,15–17], as well as the algorithm of Demetrescu & Italiano [4], along with some new variants of them. In particular, from the former set we have implemented the space-saving version of King's algorithm [10,12] along with two new variants, the algorithm of Roditty & Zwick [16], the algorithm of Roditty [15] along with a new variant, and the very recent algorithm of Roditty & Zwick [17]. In addition, we have implemented the decremental algorithm of Roditty & Zwick [16], which we modified and fine-tuned so that it can work in a fully dynamic environment. We call this pseudo fully dynamic algorithm **RZ-Opt**. A summary of the theoretical bounds of all these algorithms can be found in Fig. 1. We compared the above implementations to **Ital-Gen** and also to the simple-minded algorithms presented in [6]. Our experiments were conducted on three types of inputs: random inputs, synthetic (worst-case) inputs, and real-world inputs.

Our experiments showed that, regardless of the type and size of input, the algorithm of Demetrescu & Italiano [4], the space-saving version of King's algorithm [10,12] and its new variants were by far the slowest, followed by the algorithm of Roditty [15]. The performance of the latter is actually surprising, since at least for some cases it is theoretically better than the algorithm in [16]. For random inputs, the pseudo fully dynamic algorithms **Ital-Gen** and **RZ-Opt** were dramatically faster than any of the fully dynamic ones or their variants, with **RZ-Opt** being usually the fastest. Regarding fully dynamic algorithms, the first interesting outcome is that the theoretically inferior – with respect to [17] – algorithm of Roditty & Zwick in [16] was the fastest. The second interesting outcome is that Demetrescu & Italiano's [4] algorithm exhibits an excellent locality of reference and achieves the smallest ratio of cache misses w.r.t. *any* algorithm in our study (however, its performance degrades, since it requires a vast number of main memory accesses). For synthetic inputs, the fastest algorithm in all cases were the simple-minded ones. Regarding the dynamic algorithms, we observed that the situation is similar to the random inputs as long as the graph consists of strongly connected components (SCCs) of small size. When, however, the size of SCCs increases, the fully dynamic algorithms of Roditty & Zwick [16,17] perform dramatically better than the pseudo fully dynamic ones. This implies that the fully dynamic algorithms demonstrate their theoretical superiority by learning quickly the specific structure of these graphs and benefiting substantially from it. The experimental results with the real-world inputs were similar to those of random inputs. Due to space limitations, some parts are omitted from this version. Full details can be found in [13].

2 Algorithms and Their Implementation

Let $G = (V, E)$ be a digraph with n vertices and an initial number of m_0 edges. If there is a directed path from a vertex u to a vertex v , then u is called an *ancestor* of v , v is called a *descendant* of u , and v is said to be *reachable* from u . The digraph $G^* = (V, E^*)$ that has the same vertex set with G but has an edge $(u, v) \in E^*$ iff v is reachable by u in G is called the *transitive closure* of G . In the following, we denote by m' the number of edges to be inserted and/or deleted, and consequently $m = m_0 + m'$.

2.1 The Algorithms of Italiano and Their Extensions

We start with the partially dynamic algorithms of Italiano [8,9]. The incremental algorithm applies to any digraph, while the decremental applies to DAGs. In [6], a modified and fine-tuned implementation of these algorithms was presented, called **Ital-Opt**. **Ital-Opt** maintains for each $u \in V$ a tree $Desc[u]$, which contains all descendants of u . The $Desc$ trees are maintained implicitly using a $n \times n$ matrix $Parent$ and a Boolean query for vertices i and j is carried out in $O(1)$ time, by checking $Parent[i, j]$. The main idea of the algorithm is that the maintenance of a tree during a sequence of edge deletions can be done efficiently, if the graph is a DAG. More details in [6,8,9].

Based on **Ital-Opt**, Frigioni et al. [6] developed a new algorithm called **Ital-Gen** that can handle edge insertions and deletions in general digraphs. The main idea of **Ital-Gen** is that if every strongly connected component (SCC) is replaced by a single vertex, then the resulting graph G' is a DAG, whose transitive closure can be maintained using **Ital-Opt**. For each SCC C , the algorithm maintains (among other) a *sparse certificate* S (sparse subgraph) of C . If an edge e is removed from C and if e belongs to S , then **Ital-Gen** checks whether C has broken. In addition, the algorithm maintains an $n \times n$ matrix $Index$ such that $Index(i, j)$ is true iff there is an i - j path. A query can be answered in $O(1)$ time by checking this matrix. Both **Ital-Opt** and **Ital-Gen** can be modified so that they can work in a fully dynamic environment [6].

2.2 The Algorithm of King and Its Variants

King's algorithm [10] uses forests of BFS trees and a set of matrices to facilitate query answering in $O(1)$ time. An *Out* (resp. *In*) *BFS* tree of depth d rooted at vertex r is a data structure which maintains vertices reachable from (resp. reaching) r , and whose distance from r is less than or equal to d . Maintenance of this data structure for any sequence of edge deletions can be done in $O(m_0d)$ time. The algorithm maintains $k = \lceil \log_2 n \rceil$ forests F^1, F^2, \dots, F^k , where each F^i contains a pair of BFS trees In_u^i and Out_u^i of depth $d = 2$ rooted at every vertex $u \in V$. The BFS trees of the forest F^1 are constructed for G . The BFS trees of the forest F^i , $i > 1$, are constructed for the graph $G^i = (V, E^i)$, where E^i is defined as follows: if there is a path between two vertices u, w whose length is less than or equal to 2^i , then $(u, w) \in E^i$.

During the deletion of an edge e (or of a set of edges) the edge e is removed from any BFS tree of forest F^i , $1 \leq i \leq k$, it belongs. In the case of an insertion of an edge (or a set of edges) incident to a vertex u the trees In_u^i, Out_u^i , $i = 1, \dots, k$ are built from scratch.

King and Thorup in [12] proposed a space saving version of this algorithm. Graph $G^i = (V, E^i)$ is maintained using an incidence matrix M : if $(u, w) \in E^i$, then $M(u, w) = 1$, otherwise $M(u, w) = 0$. The maintenance of a BFS tree using these matrices costs now $O(n^2d)$ time, however this does not affect the amortized update bound. We shall refer to the implementation of this algorithm as **King-1**.

In addition, we have implemented a variant of this algorithm, called **King-2**. The idea is to maintain BFS trees of depth $d > 2$. In **King-2** we considered $d = 8$, which reduces the number of forests by $2/3$. The asymptotic complexity of the update operations is the same with those of **King-1**. Furthermore, we have implemented another variant, called **King-3**, that maintains BFS trees of depth D , where D is the diameter of the graph, and therefore it requires only one forest of BFS trees.

2.3 The Algorithms of Roditty and Zwick

The Algorithms in [16] and Their Extensions. Roditty and Zwick proposed in [16] a randomized (Las Vegas) decremental algorithm, which is a combination of the decremental part of **Ital-Gen** [6] with a new decremental algorithm [16] for maintaining the SCCs of a digraph. The crucial observation is that **Ital-Gen** requires $O(nm_0)$ time to handle any sequence of edge deletions, if the check whether a SCC has broken is handled by the new decremental algorithm [16] for maintaining the SCCs. The main idea of the decremental algorithm which maintains the SCCs is the following. In each SCC C of the graph it maintains an In-BFS tree $In(w)$ and an Out-BFS tree $Out(w)$ rooted at a random vertex $w \in C$. If an edge (x, y) belonging to C is deleted, then the BFS trees are updated accordingly and, to determine whether C breaks it suffices to check whether $x \in In(w)$ and $y \in Out(w)$. If C has broken, the new SCCs to which C breaks are computed and new BFS trees are constructed, except for the new SCC C' which contains w and inherits the BFS trees rooted at w .

We have modified the decremental transitive closure algorithm of Roditty and Zwick so that it can handle edge insertions, without affecting the performance when it handles edge deletions. Specifically, edge deletions are processed as in the “original” algorithm and an edge insertion is handled as follows. If the new edge connects two different SCCs, then **Ital-Gen** is used. On the other hand, if the new edge belongs to a SCC, then its BFS trees are updated in a recursive manner by lifting up nodes towards the root of the tree. We call this pseudo fully dynamic algorithm **RZ-Opt**. This algorithm handles any sequence of edge deletions in $O(nm_0)$ time and m' edge insertions in $O(m'(n + m_0 + m'))$ time. Consequently, we expect **RZ-Opt** to be faster when handling edge deletions, and **Ital-Gen** to be faster when handling edge insertions.

A final remark concerns the algorithm for maintaining the SCCs. When a SCC C breaks, then the BFS trees rooted at $w \in C$ are inherited by the new

SCC that contains w . In our implementation of **RZ-Opt** these trees are built from scratch, therefore **RZ-Opt** may spend more than $O(nm_0)$ time to handle m' edge deletions. Despite this fact, however, **RZ-Opt** proved to be competitive to the fastest algorithms implemented by Frigioni et al. [6].

We now turn to the combinatorial fully dynamic algorithm in [16]. Initially, a decremental data structure **DD** for maintaining the transitive closure is built (**DD** could be **RZ-Opt** or **Ital-Gen**). The insertion of an edge (or a set of edges) incident to a vertex u is done as follows. Vertex u is added to a set S of vertices and an ancestor (resp. descendant) tree $In(u)$ (resp. $Out(u)$) rooted at u is built. If $|S|$ becomes equal to a predetermined parameter t ($t = \sqrt{n}$ in [16] and in our implementation), then all data structures are re-initialized. The deletion of a set E' of edges is done as follows. First, every $e \in E'$ is removed from **DD**. Then, for every $w \in S$, the trees $In(w)$ and $Out(w)$ are rebuilt. A query for an u - w path is computed as follows. First **DD** is queried and if the answer is yes, then there exists a u - w path in G . If the answer is no and if a vertex z exists such that $u \in In(z)$ and $w \in Out(z)$, then again a u - w path exists. Otherwise, there is no u - w path. We shall refer to the implementation of this algorithm as **RZ-1**.

The Algorithm of Roditty [15]. The recent fully dynamic algorithm proposed by Roditty [15] is inspired by the algorithm of King [10]. It uses a decremental data structure for maintaining paths composed of edges belonging to the initial graph and an algorithm for maintaining a forest of in-trees (ancestor trees) and out-trees (descendant trees) around each *insertion center* (i.e., the vertex incident to the current set of edge insertions). Boolean queries are answered in $O(1)$ time using an $n \times n$ matrix *count* such that each entry $count(x, y)$ equals the number of insertion centers that lie on a path from x to y .

An out-tree (in-tree) around a vertex u maintains the so-called *blocks* with respect to u that are reachable from (reach) u . Two vertices x, y belong to the same *block with respect to u* , if x and y belong to the same SCC after the last edge insertion centered at u and after every subsequent delete operation. The insertion of a set of edges incident to u , may change the blocks with respect to u , while an edge deletion may change every block that exists so far. The main idea of this algorithm is that all in-trees and out-trees can be maintained implicitly using a single adjacency matrix M of size $O(n^2)$. Each update of the matrix requires $O(n^2)$ time. In our implementation, called **Rod**, we have used **RZ-Opt** as the decremental structure, because it has been the fastest algorithm in handling edge deletions in general digraphs.

Our experiments revealed that this algorithm spends a significant amount of time in building the adjacency matrix M (M is built from scratch at every update operation). The algorithm must maintain entries $M(v, x) = \min_w M(v, w)$, where v is a vertex, x is a block, and w is a vertex (or a block) belonging to x . The value of each entry $M(v, x)$ in **Rod** is computed using a for loop across all entries. Since these values are non-negative, we can exit the loop as soon as a zero entry is found. We have generated a variant of the algorithm, called **Rod-Opt**, based on this fact to see whether it affects performance.

The Algorithm of Roditty and Zwick in [17]. The very recent algorithm of Roditty and Zwick [17] is a combination of a new persistent dynamic algorithm for maintaining the SCCs of a graph with a new decremental algorithm for maintaining reachability trees presented in [17].

The persistent algorithm for strong connectivity works as follows. During the insertion of an edge (or of a set of edges incident to a vertex), a new version of the graph is created and the algorithm maintains all versions. Each version of the graph, once created, is not affected by any edge insertion, while, each edge deletion applies to all versions. Each SCC of version i (created by the i -th insert operation) is either a SCC or a union of SCCs of version $i - 1$. As a result, the SCCs of all versions of the graph can be maintained as a forest. The edge set of the graph is partitioned into $t + 1$ edge sets H_i ($i = 1, \dots, t + 1$). If an edge e connects two different SCCs in the current version of the graph, then $e \in H_{t+1}$. Otherwise, $e \in H_j$ where j is the version at which e became an internal edge of some SCC. When an edge insertion occurs, H_{t+1} is used to compute new SCCs of the graph. In order to achieve this, a Union-Find algorithm is used, which can efficiently merge SCCs by representing them as sets of vertices and return the SCC to which a vertex belongs. Roditty & Zwick [17] use this algorithm in a very clever way in order to maintain a reachability tree in a decremental environment (see [17] for the details) at a total cost of $O(m + n \log n)$.

The fully dynamic algorithm for maintaining the transitive closure maintains a pair of reachability trees In_u, Out_u for each vertex $u \in V$. The reachability tree Out_u (resp. In_u) maintains SCCs reachable from (resp. reaching) u . When a set of edges incident to u is inserted into the graph G , a new version G^u of G is created, and the trees In_u, Out_u are built from scratch. Trees In_u, Out_u are maintained with respect to G^u . Each version G^u undergoes only edge deletions and is replaced by a new version when another edge insertion around u occurs. When an edge deletion occurs, the forest of SCCs is updated using the persistent algorithm for strong connectivity. If a SCC C contained in a reachability tree breaks, then C is replaced by the SCCs to which it breaks, and the algorithm checks whether these SCCs can be connected to the tree. A boolean query (u, v) is answered in $O(n)$ time by checking for each vertex w whether $u \in In_w$ and $v \in Out_w$. We refer to this algorithm as RZ-P.

2.4 The Algorithm of Demetrescu and Italiano

The main idea of the algorithm of Demetrescu and Italiano [4] (see also [5]) is to reduce the transitive closure problem to the problem of maintaining polynomials over matrices subject to updates of their variables. The algorithm takes advantage of the following equivalence: If G is a directed graph and X_G is its adjacency matrix, then computing the Kleene closure X_G^* of X_G is equivalent to computing the transitive closure of G .

Let X_b^a denote a Boolean matrix. The basic data structure (we shall refer to it as Struct1) used by the algorithm maintains polynomials P over such matrices of degree 2; i.e., P is of the form $P = \sum_{i=1}^h X_1^i \cdot X_2^i$. This structure (after an initialization phase which takes $O(hn^\omega + hn^2)$ time, where ω is the exponent

of matrix multiplication) is able to maintain P in $O(n^2)$ time when a Boolean matrix X_b^a is changed. Struct1 uses $O(hn^2)$ space. Polynomials P_k of degree $k > 2$ can be maintained by using Struct1, because each P_k can be represented by a sum of $O(k^2)$ polynomials of degree 2.

Let G be a directed graph, X its adjacency matrix, X^* the Kleene closure of X , and n the number of nodes of the graph. Then X^* can be computed recursively by computing 12 polynomials and 3 closure matrices of size $\frac{n}{2} \times \frac{n}{2}$ [4,5]. Each such closure matrix of size $\frac{n}{2} \times \frac{n}{2}$ is maintained recursively by 12 polynomials and 3 closures of size $\frac{n}{4} \times \frac{n}{4}$, and so on. When an edge insertion or deletion occurs, the transitive closure information is updated by properly updating the 12 polynomials and the 3 matrix closures of size $\frac{n}{2} \times \frac{n}{2}$ (each matrix closure is updated recursively). In this way the algorithm can handle insertion of a set of edges around a vertex u and deletion of an arbitrary set of edges. Boolean queries can be answered in $O(1)$ time.

In our implementation, which we refer to as DI, we have not used matrix multiplication; however, this affects only the initialization time of the algorithm, because in update operations matrix multiplication is not used.

2.5 Simple-Minded Algorithms and Summary

Frigioni et al [6] developed three simple-minded algorithms based on the following idea. When an edge insertion or edge deletion occurs, then the particular edge is simply added or removed from G , resulting in a $O(1)$ time update operation. Queries are answered in $O(n + m)$ worst-case time by applying some graph-searching algorithm among BFS, DFS, and DBFS (vertices are visited in DFS order, but every time a vertex is visited we check whether the target vertex is any of its adjacent ones). The theoretical time and space bounds of all algorithms and their variants considered in our study are summarized in Fig. 1.

Algorithm	Reference	Amortized Update Time		Query Time	Space
Ital-Gen (†)	[6]	$O(n)$ per ins	$O(m)$ per del	$O(1)$	$O(n^2)$
RZ-Opt (†)	This paper	$O(m)$ per ins	$O(n)$ per del	$O(1)$	$O(n^2)$
RZ-1	[16]	$O(m\sqrt{n})$		$O(\sqrt{n})$	$O(n^2)$
Rod	[15]	$O(n^2)$		$O(1)$	$O(n^2)$
Rod-Opt	This paper	$O(n^2)$		$O(1)$	$O(n^2)$
RZ-P	[17]	$O(m + n \log n)$		$O(n)$	$O(nm)$
King-1	[10,12]	$O(n^2 \log n)$		$O(1)$	$O(n^2 \log n)$
King-2	This paper	$O(n^2 \log n)$		$O(1)$	$O(n^2 \log n)$
King-3	This paper	$O(n^2 D)$		$O(1)$	$O(n^2)$
DI	[4]	$O(n^2)$		$O(1)$	$O(n^2)$
Simple	[6]	$O(1)$		$O(n + m)$	$O(n + m)$

Fig. 1. Amortized bounds for $m' = \Theta(m)$ edge insertions/deletions. D denotes the diameter of the graph and $m = m_0 + m'$. (†) Pseudo fully dynamic algorithms.

3 Experimental Results

For our experimental study we used the experimental platform developed by Frigioni et al [6]. We implemented each algorithm as a C++ class using LEDA [14] and the library for dynamic graph algorithms [2]. We used the correctness checking program in [6] to verify the correctness of our implementations. The source code is available from <http://www.ceid.upatras.gr/faculty/zaro/software/>. The experiments were run on three different computing environments; namely, (i) a Sun UltraSparc II (USparc-II) with 4 processors at 300 MHz, Solaris 7 operating system, 1.2GB of main memory, and 2MB L2 cache per processor; (ii) an Intel Pentium 4 (P4) at 1.6 GHz, with linux SUSE 7.3 operating system, 512MB of main memory, and 512KB L2 cache; and (iii) an AMD Athlon at 1.9 GHz, with linux Mandrake 10 operating system, 512MB of main memory, and 256KB L2 cache. We used this variety of computing environments to investigate whether it affects the relative performance of algorithms, especially regarding memory accesses and cache effects since all algorithms require $\Omega(n^2)$ space.

In all experiments conducted we did not observe any substantial difference in the relative performance of the implementations. The same applies for the simulation of cache misses with Valgrind (valgrind.kde.org). For that reason we will mostly report experiments run on P4. RZ-P (as expected) was by far the most memory demanding algorithm, and after a certain point its performance is dominated by the swaps executed between main and secondary memory. Due to this fact, we were unable to run large input instances (e.g., graphs with more than 800 vertices) on P4 and Athlon. For the case of simple-minded algorithms, we report results only with the fastest of them in the particular class of inputs. We performed experiments on three classes of inputs.

Random Inputs. We performed our tests on random digraphs with $n \in [100, 700]$ vertices and several values on the initial number of edges m_0 . For these values of n and m_0 , we considered various lengths of operation sequences $|\sigma| \in [500, 50000]$. We generated a large collection of data sets, each consisting of 5 to 10 samples, and corresponded to a fixed value of graph parameters and $|\sigma|$. The reported values are average CPU times over the samples. The random sequence of operations consisted of update operations (insertions/deletions) and queries (Boolean). Following similar studies (e.g., [3,6]), we considered two types of patterns: uniformly mixed queries and updates (each occurring with probability 1/2, where an update can equally likely be an insertion or deletion), and uniformly mixed insertions, deletions, and queries (each such operation occurs with probability 1/3).

Our experiments revealed that DI, King-1, King-2 and King-3 were by far the slowest, followed by Rod and Rod-Opt, even for small input instances and moderate operation sequences with 50% of queries. The bad behaviour of King-1, King-2, King-3, Rod and Rod-Opt is due to the maintenance of incidence matrices. This slows down the construction and the update of the trees maintained, since they require quadratic time regardless of the edge density. Among the vari-

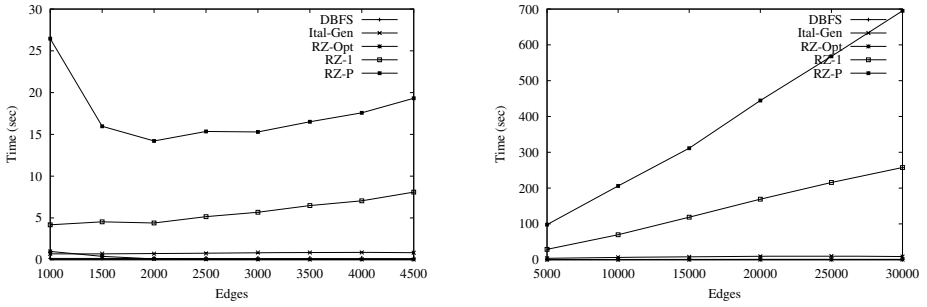


Fig. 2. Random digraphs with $n = 300$. Experiments run on P4. Left: $|\sigma| = 5000$ (33% queries). Right: $|\sigma| = 30000$ (33% queries).

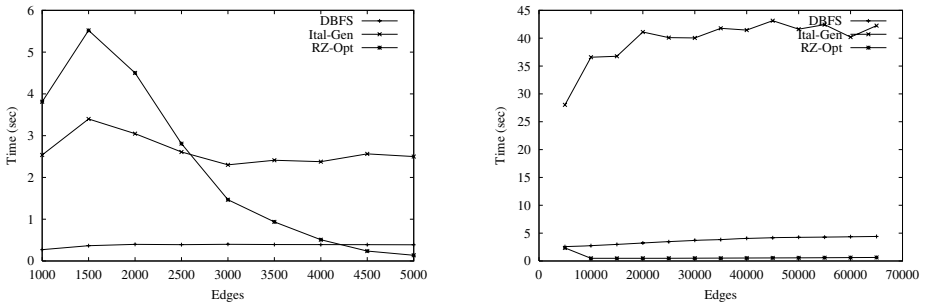


Fig. 3. Random digraphs with $n = 700$. Experiments run on P4. Left: $|\sigma| = 5000$ (33% queries). Right: $|\sigma| = 50000$ (33% queries).

ants of King’s algorithm, we observe that **King-3** is almost always the fastest due to the fact that it maintains less trees than **King-1** and **King-2**. When the graph becomes very sparse, however, **King-3** is slower due to the overhead of maintaining a BFS tree for the large (giant) component. Finally, **Rod-Opt** is from 1.5 (50% queries) to 2 (33% queries) times faster than **Rod** due to the heuristic of aborting the loop as soon as a zero $M(v, w)$ entry has been found.

We now turn to **DI**. One possible explanation for the bad performance of **DI** is that it maintains a large number of polynomials (the update of each one costs $O(n^2)$ time). Moreover, the recursive maintenance of the closure matrices turns out to be inefficient, because **DI** becomes slower as the recursion depth increases. On the other hand, **DI** (as expected) is faster than King’s algorithm and its variants, because it manages to exhibit a better locality of reference. Indeed, simulation of cache misses with Valgrind revealed that the cache behaviour of **DI** is dramatically (about 50–100 times) better. Actually, **DI** has the *smallest* ratio of cache misses w.r.t. *any* algorithm in our study.

The comparison of the rest of the algorithms is shown in Fig. 2. Simple-minded and pseudo fully dynamic algorithms clearly outperform **RZ-1** and **RZ-P**. **RZ-P** is penalized by its larger query time, by its large memory demand after a certain point (see Fig. 2(right)), and most importantly by the maintenance

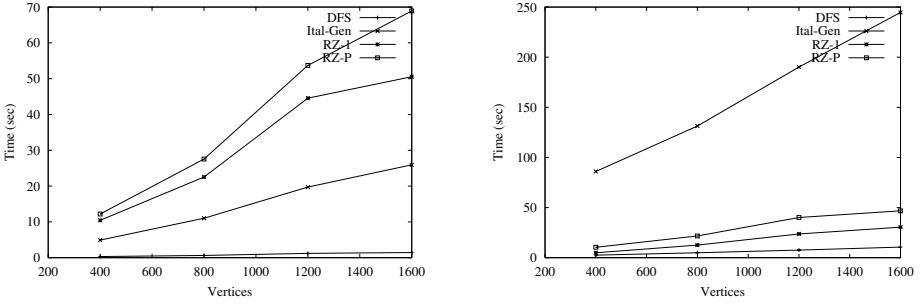


Fig. 4. Synthetic digraphs with $|\sigma| = 1920$ (33% queries). Experiments run on USparc-II. Left: clique size 10. Right: clique size 80.

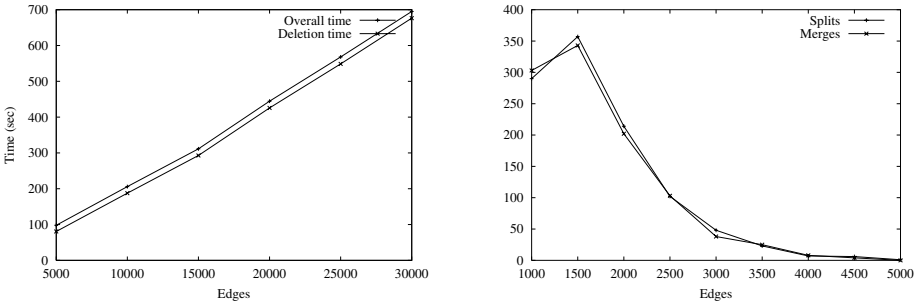


Fig. 5. Experiments on P4. Left: Deletion time vs overall time of RZ-P, for $n = 300$ and $|\sigma| = 30000$ (33% queries). Right: Splits and merges of SCCs in RZ-Opt and Itai-Gen, for $n = 700$ and $|\sigma| = 5000$ (33% queries).

of SCCs across all versions of the graph (quite expensive during deletions). Fig. 5(left) shows clearly that RZ-P spends almost all of its time in handling edge deletions. RZ-1 is faster than RZ-P due to its smaller query time and the fact that it uses RZ-Opt to handle edge deletions. Its main drawback, however, seems to be the fact that its decremental data structure (i.e., RZ-Opt) must be rebuilt every $O(\sqrt{n})$ operations.

We now turn to the three faster implementations DBFS, Itai-Gen, and RZ-Opt; see Fig. 3. When the graph is relatively sparse (less than $n \ln n$ edges), DBFS is the fastest algorithm, while RZ-Opt is considerably faster in denser graphs. The differences between the performance of Itai-Gen and RZ-Opt can be explained by how they handle SCCs. Itai-Gen is not efficient in handling edge deletions in SCCs because even if a SCC does not break, it may spend $O(n + m)$ time to determine whether the SCC has broken and to rebuild its sparse certificate. This claim is confirmed with the support of Fig. 5(right) and Fig. 3(left): although the number of SCCs that split decreases, the running time is practically unaffected. On the other hand, RZ-Opt is not efficient in handling edge insertions because if a new edge is created in a SCC, then it may spend $O(n + m)$ time to update the BFS trees. However, as the edge density increases, RZ-Opt performs

better, since the BFS trees have small depth. **RZ-Opt** is highly dominated by the merges and splits of SCCs, a fact that can be easily confirmed by an inspection of the curves of Fig. 5(right) with the overall time curve of Fig. 3(left). Consequently, in sparse graphs, where many splits and merges of SCCs occur, both algorithms have more-or-less the same performance. As soon as the strong connectivity threshold ($n \ln n$) is approached and/or surpassed **RZ-Opt** outperforms **Ital-Gen**, as it performs much less work mainly for deletions.

Synthetic Inputs. We have considered and slightly modified the synthetic inputs introduced by Frigioni et al [6], which enforce the algorithms to exhibit their worst-case behaviour. These graphs consist of a sequence of $s = \lceil n/k \rceil$ cliques C_1, \dots, C_s , each of size k , interconnected with a set of “bridges”. A bridge is a pair of directed edges connecting a node of C_i with a node of C_{i+1} , and vice versa. Insertions and deletions are only performed on bridges and in a specific order, such that the bridge inserted/deleted last would provide new reachability and SCC information from roughly $n/2$ to the other $n/2$ vertices of the graph.

As with random inputs, **DI**, King’s algorithm and its variants as well as **Rod** and **Rod-Opt** were the worst, and hence we do not report results for them. **Ital-Gen** was always faster than **RZ-Opt** (due to the inefficient insertion procedure of the latter; see below), and hence we report results only with the former. Fig. 4 illustrates the performance of the rest of the algorithms. Similar results hold for smaller or larger operation sequences and different computing environments; we report results on USparc-II (the machine with the largest memory) to include large values of n . We observed that **DFS** was always the fastest algorithm.

The performance of **Ital-Gen** deteriorates as the clique size k increases, since for large k we get large SCCs whose maintenance becomes very costly due to their splits and merges. Note that a split (resp. merge) of a SCC occurs every two edge deletions (resp. insertions), and the data structures in those SCCs must be built from scratch. On the other hand, **RZ-1** and **RZ-P** perform better than **Ital-Gen** as k increases, since they can handle better the splits and merges of large SCCs. As a side remark, the good performance of **RZ-1** indicates that **RZ-Opt** (which is used by **RZ-1** for deleting edges) is worse than **Ital-Gen** mainly due to the inefficient handling of edge insertions. For **RZ-P** a large value of k implies a small number of insertion centers (tails of edges inserted), and therefore it maintains less versions of the graph. In addition, the maintenance of the reachability trees has a very low cost, since the algorithm has to check only the external to a SCC edges, i.e., the bridges. However, **RZ-P** is slower than **RZ-1** probably due to the overhead caused by deletions, in order to maintain the forest of SCCs across all versions of the graph. In conclusion, the fully dynamic algorithms demonstrate their theoretical superiority by learning quickly the specific structure of the synthetic graphs and benefiting substantially from it.

Real-World Inputs. We have also used two inputs motivated by real-world graphs. The first graph has 1259 vertices and 5101 edges and represents a fragment of the Internet visible from RIPE (www.ripe.net) that has been used in [6]. The second graph describes a US road network (<ftp://edcftp.cr.usgs.gov>)

having 576 vertices and 1762 edges, and has been used in [3]. On these graphs, we run random sequences of operations, and we observed similar results to the experiments on random inputs.

References

1. S. Abdeddaim. Algorithms and Experiments on Transitive Closure, Path Cover and Multiple Sequence Alignment. In *Proc. 2nd Workshop on Algorithm Engineering and Experiments – ALENEX 2000*, pp. 157–169, 2000.
2. D. Alberts, G. Cattaneo, G.F. Italiano, U. Nanni, and C. Zaroliagis. A Software Library of Dynamic Graph Algorithms. In *Proc. Workshop on Algorithms and Experiments – ALEX’98*, pp. 129–136, 1998.
3. C. Demetrescu, S. Emiliozzi, and G. F. Italiano. Experimental Analysis of Dynamic All Pairs Shortest Path Algorithms. In *Proc. 15th ACM-SIAM Symp. on Discrete Algorithms – SODA 2004*, pp.362-371.
4. C. Demetrescu and G. F. Italiano. Fully Dynamic Transitive Closure: Breaking through the $O(n^2)$ Barrier. In *Proc. 41st IEEE Symp. on Foundations of Computer Science – FOCS 2000*, pp. 381–389, 2000.
5. C. Demetrescu. Fully Dynamic Algorithms for Path Problems on Directed Graphs. PhD Thesis, University of Rome “La Sapienza”, February 2001.
6. D. Frigioni, T. Miller, U. Nanni and C. Zaroliagis. An Experimental Study of Dynamic Algorithms for Transitive Closure. *ACM Journal of Experimental Algorithmics*, 6(9), 2001.
7. M.R. Henzinger and V. King. Fully Dynamic Biconnectivity and Transitive Closure. In *Proc. 36th IEEE Symposium on Foundations of Computer Science – FOCS’95*, pp. 664–672, 1995.
8. G. F. Italiano. Amortized efficiency of a path retrieval data structure. *Theoretical Computer Science*, 48:273-281, 1986.
9. G. F. Italiano. Finding paths and deleting edges in directed acyclic graphs. *Information Processing Letters*, 28:5-11, 1988.
10. V. King. Fully Dynamic Algorithms for Maintaining All-Pairs Shortest Paths and Transitive Closure in Digraphs. In *Proc. 40th IEEE Symposium on Foundations of Computer Science – FOCS’99*, pp.81-91, 1999.
11. V. King, and G. Sagert. A Fully Dynamic Algorithm for Maintaining the Transitive Closure. In *Proc. 31st ACM Symp. on Theory of Comp. – STOC’99*, pp. 492–498.
12. V. King and M. Thorup. A Space Saving Trick for Directed Dynamic Transitive Closure and Shortest Path Algorithms. In *Proc. 7th Comp. and Combinatorics Conference – COCOON 2001*, pp.268-277, 2001.
13. I. Krommidas and C. Zaroliagis. An Experimental Study of Algorithms for Fully Dynamic Transitive Closure. CTI Tech. Report TR 2005/07/01, July 2005.
14. K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.
15. L. Roditty. A faster and simpler fully dynamic transitive closure. *Proc. 14th ACM-SIAM Symp. on Discrete Algorithms – SODA 2003*, pp. 404-412.
16. L. Roditty and U. Zwick. Improved dynamic reachability algorithms for directed graphs. In *Proc. 43rd IEEE Symposium on Foundations of Computer Science – FOCS 2002*, pp.679-690, 2002.
17. L. Roditty and U. Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In *Proc. 36th ACM Symp. on Theory of Computing – STOC 2004*.

Experimental Study of Geometric t -Spanners

Mohammad Farshi^{1,*} and Joachim Gudmundsson²

¹ Department of Mathematics and Computing Science, TU Eindhoven,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
m.farshi@tue.nl

² NICTA^{**}, Sydney, Australia
joachim.gudmundsson@nicta.com.au

Abstract. The construction of t -spanners of a given point set has received a lot of attention, especially from a theoretical perspective. In this paper we perform the first extensive experimental study of the properties of t -spanners. The main aim is to examine the quality of the produced spanners in the plane. We implemented the most common t -spanner algorithms and tested them on a number of different point sets. The experiments are discussed and compared to the theoretical results and in several cases we suggest modifications that are implemented and evaluated. The quality measurements that we consider are the number of edges, the weight, the maximum degree, the diameter and the number of crossings.

1 Introduction

Consider a set V of n points in the plane. A network on V can be modeled as an undirected graph G with vertex set V of size n and an edge set E of size m where every edge $e = (u, v)$ has a weight $wt(e)$. A geometric (Euclidean) network is a network where the weight of the edge $e = (u, v)$ is the Euclidean distance $d(u, v)$ between its endpoints u and v . Let $t > 1$ be a real number. We say that G is a t -spanner for V , if for each pair of points $u, v \in V$, there exists a path in G of weight at most t times the Euclidean distance between u and v . We call this path a t -path between u and v . The minimum t such that G is a t -spanner for V is called the stretch factor, or dilation, of G . Finally, a subgraph G' of a given graph G is a t -spanner for G if for each pair of points $u, v \in V$, there exists a path in G' of weight at most t times the weight of the shortest path between u and v in G .

Complete graphs represent ideal communication networks, but they are expensive to build; sparse spanners represent low-cost alternatives. The weight of the spanner is a measure of its sparseness; other sparseness measures include the number of edges, the maximum degree, and the number of crossings. Spanners

* Supported by Ministry of Science, Research and Technology of I. R. Iran.

** National ICT Australia is funded through the Australian Government's Backing Australia's Ability initiative, in part through the Australian Research Council.

for complete Euclidean graphs as well as for arbitrary weighted graphs find applications in robotics, network topology design, distributed systems, design of parallel machines, and many other areas and have been a subject of considerable research. Recently low-weight spanners found interesting practical applications in areas such as metric space searching [15,16] and broadcasting in communication networks [8,13]. Several well-known theoretical results also use the construction of t -spanners as a building block, for example, Rao and Smith [18] made a breakthrough by showing an optimal $O(n \log n)$ -time approximation scheme for the well-known Euclidean *traveling salesperson problem*, using t -spanners (or banyans). Similarly, Czumaj and Lingas [6] showed approximation schemes for minimum-cost multi-connectivity problems in geometric networks. The problem of constructing spanners has received considerable attention from a theoretical perspective, see the survey [7], but almost no attention from a practical, or experimental perspective [15,20].

In this paper we consider the most well-known algorithms for the construction of t -spanners in the plane: greedy spanners, Θ -graphs, ordered Θ -graphs and spanners constructed from the well-separated pair decomposition (WSPD). The quality measurements used in the literature is the number of edges, the weight, the maximum degree, the diameter and the number of crossings. We study each of the algorithms independently, but also in combination with each other. To the best of the authors knowledge, this is the first time an extensive experimental study has been performed on the construction of t -spanners. Navarro and Paredes [15] presented four heuristics for point sets in high-dimensional metric space ($d = 20$) and showed by empirical methods that the running time was $\mathcal{O}(n^{2.24})$ and the number of edges in the produced graphs was $\mathcal{O}(n^{1.13})$. In [20] Sigurd and Zachariassen considered the problem of constructing a minimum weight t -spanner of a given graph, but they only considered sparse graphs of small size, i.e., graphs with at most 64 vertices and with average vertex degree 4 or 8.

The paper is organized as follows. Next we briefly go through the different properties that we considered in the experiments, and present a first algorithm that computes the t -diameter of a t -spanner. In Section 3 we give a short description of each of the algorithms that we implemented, together with their theoretical bounds. Then, in Sections 4–5 we discuss the implementations, the data we used and the experimental results, followed by Section 6 where we discuss the results and propose improvements.

2 Spanner Properties

As input we are given a set V of n points in the plane and a positive real value $t > 1$. The aim is to compute a t -spanner for V with some good properties where the quality measurements that we will consider in this paper are as follows:

Size: Defined to be the number of edges in the graph. This is the most important property of the constructed networks and all the implemented algorithms produce spanners with only $\mathcal{O}(n)$ edges. This key feature has made the con-

struction of spanners one of the fundamental tools in the development of fast approximation algorithms for geometrical problems.

Degree: The maximum number of edges incident to a vertex. This property has been shown to be useful in the development of approximation algorithms [10] and for the construction of ad hoc networks where small degree is essential in trying to develop fast localized algorithms [13].

Weight: The weight of a Euclidean network is the sum of the edge weights. The best that can be achieved is a constant times the weight of the minimum spanning tree of input point set, denoted $wt(MST)$.

Diameter: Defined as the smallest integer d such that for any pair of vertices u and v in V , there is a t -path in the graph between u and v containing at most d edges, i.e., a path of weight at most $t \cdot |uv|$. Note that there is a trade-off between the degree and the diameter and between the diameter and the size [2]. For wireless ad hoc networks it is often desirable to have small diameter since it determines the maximum number of times a message has to be transmitted in a network.

Crossings: A pair of edges with a non-empty intersection is said to *cross*. The total number of pairs of edges in a graph that cross is the number of crossings. One wants to minimize the number of crossings since it decreases the complexity and the readability of a graph [17].

2.1 Diameter

As mentioned before, the diameter of a t -spanner is the smallest integer k such that there is a t -path between each pair of points that contains at most k edges. As far as we know there is no known algorithm on how to compute the diameter of a t -spanner, below we present a dynamic programming approach for the problem which we will use in Section 4 to calculate the diameter of the constructed t -spanners.

Assume that $L[p, q, k]$ is the shortest path length between p and q with at most k edges. If there is no such path, we set $L[p, q, k]$ to ∞ . With this definition, the diameter of a t -spanner is the smallest integer k such that $L[p, q, k] \leq t \cdot |pq|$, for all points p and q .

Lemma 1. *Let $G = (V, E)$ be a graph and p and q are two vertices of G . For each integer $k \geq 2$,*

$$L[p, q, k] = \min \left\{ L[p, q, k - 1], \min_{r \in V \setminus \{p, q\}} \{L[p, r, k - 1] + L[r, q, 1]\} \right\}.$$

Note that, based on Lemma 1, $L[-, -, 1]$ and $L[-, -, k - 1]$ is sufficient for computing $L[-, -, k]$.

Now for computing the diameter, we start with $k = 1$. Obviously $L[p, q, 1] = |pq|$ for all edges (p, q) in the graph and $L[p, q, 1] = \infty$ otherwise. For $k \geq 2$, we can compute $L[-, -, k]$ using $L[-, -, k - 1]$ and $L[-, -, 1]$. We continue

construction until we received a k such that $L[p, q, k] \leq t \cdot |pq|$, for all points p and q .

Corollary 1. *The diameter of a t -spanner G can be computed in $\mathcal{O}(dn^3)$ time using $\mathcal{O}(n^2)$ space, where d is the diameter of G .*

3 The Spanner Construction Algorithms

In this section we give a short description of each of the algorithms implemented together with their theoretical bounds.

3.1 The Greedy Algorithm

The greedy algorithm was discovered independently by Bern in 1989 and Althöfer et al. [1]. The graph constructed using the greedy algorithm will be called a greedy graph.

The greedy algorithm maintains a partial t -spanner G' while processing all point pairs in order of increasing length. Processing a pair u, v entails a shortest-path query in G' between u and v . If there is a t -path between u and v in G' then the edge (u, v) is discarded, otherwise it is added to G' .

The time complexity of the greedy algorithm is $\mathcal{O}(n^3 \log n)$ and it uses $\mathcal{O}(n^2)$ space. There exists an $\mathcal{O}(n \log n)$ time greedy algorithm [9] that only uses linear space but, unfortunately, it is very complicated and therefore we decided to implement the simple version. The following theorem states the theoretical upper bounds.

Theorem 1. *The greedy graph is a t -spanner of V with $\mathcal{O}(n)$ edges, maximum degree $\mathcal{O}(1)$ and weight $\mathcal{O}(wt(MST(V)))$, and can be computed in time $\mathcal{O}(n \log n)$.*

Note also that a trivial $\Omega(n)$ lower bound on the diameter of a greedy graph is obtained by placing n points on a line.

The greedy approach can also be used to prune a given t -spanner $G = (V, E)$, that is, instead of considering all point pairs, the algorithm only considers the endpoints of the edges in E . In this paper we also perform experiments using the pruning tool in combination with the other algorithms, see Section 5. The time complexity of the implemented greedy pruning is $\mathcal{O}(mn \log(n))$, where m is the number of edges in the input graph.

Improvement. As mentioned above the running time of the implemented algorithm is $\mathcal{O}(n^3 \log n)$, which is too slow when performing experiments with up to 13.000 points. We use a speed-up strategy that turned out to decrease the running time considerably in practice. Originally the algorithm computes the shortest path for each pair of points to check if there is a t -path between the two points or not. But there are two simple observations. First, we only need to know "Is there a t -path between the points?" and second, the algorithm only

adds $\mathcal{O}(n)$ edge to the graph in total, so the graph does not change very often during the execution.

Therefore, we use a matrix to save the shortest path between each two points and update it only when we need to, thus it is not always up to date. Instead of computing a shortest path for each pair, the main change is that we only check if there is a t -path or not.

With these changes, the space complexity of the greedy algorithm increases, but the gain in running time is considerable. We counted the number of shortest path queries that this algorithm performs in the experiments and surprisingly it seems that $\mathcal{O}(n)$ shortest path queries is sufficient.

Conjecture 1. The modified greedy algorithm performs $\mathcal{O}(n)$ shortest path queries.

3.2 The Θ -Graph Construction

The Θ -graph was discovered independently by Clarkson [5] and Keil [11]. Keil only considered the graph in two dimensions while Clarkson extended his construction to also include three dimensions. Later Rupert and Seidel [19] and Althöfer et al. [1] defined the Θ -graph for higher dimensions.

The Θ -graph algorithm processes each point u as follows. Consider k non-overlapping cones with apex u and with angle $\theta = \frac{2\pi}{k}$ (k is related to t). For each non-empty cone C add an edge between u and v to the graph, where v is the point within C whose orthogonal projection onto the bisector of C is closest to u .

Theorem 2. *The Θ -graphs is a t -spanner of V for $t = \frac{1}{\cos \theta - \sin \theta}$ with $\mathcal{O}(kn)$ edges and can be computed in $\mathcal{O}(kn \log n)$ time.*

Note the even though the “out-degree” of each vertex is bounded by k the “in-degree” could be linear and the weight of the Θ -graph can be $\Omega(n \cdot wt(MST(V)))$. Finally, by placing n points on a line it follows that the diameter of the Θ -graph is $\Omega(n)$.

3.3 Ordered Θ -Graph

A simple variant of the Θ -graph that has been shown to have good theoretical performance is the *Ordered Θ -graph* by Bose et al. [3]. An ordered Θ -graph of V is obtained by inserting the points of V in some order. When a point p is inserted, we draw the cones as in the Θ -graph algorithm around p and connect p to its closest previously-inserted point in each cone.

Theorem 3. *The Ordered Θ -graphs is a t -spanner of V for $t = \frac{1}{\cos \theta - \sin \theta}$ with $\mathcal{O}(kn)$ edges and $\mathcal{O}(k \log n)$ degree, and can be computed in $\mathcal{O}(kn \log n)$ time.*

Remark 1. If the points are processed in random order then the diameter will be bounded by $\mathcal{O}(\log n)$ with high probability [3], but then the degree bound does not hold anymore.

3.4 The WSPD-Graph

The well-separated pair decomposition (WSPD) was developed by Callahan and Kosaraju [4]. Callahan and Kosaraju show that a WSPD of size $m = \mathcal{O}(s^d n)$ can be computed in $\mathcal{O}(s^d n \log n)$ time.

Constructing a t -spanner using the WSPD is surprisingly easy. It is sufficient to compute a WSPD of V w.r.t. $s = \frac{4(t+1)}{t-1}$ and then add an edge between each well-separated pair in the WSPD.

Theorem 4. *The WSPD-graph is a t -spanner for $V \subset \mathbb{R}^2$ with $\mathcal{O}((\frac{t}{t-1})^2 \cdot n)$ edges, and can be constructed in time $\mathcal{O}((\frac{t}{t-1})^2 n \log n)$.*

An $\Omega(n)$ lower bound on the degree and the diameter of a WSPD-graph can be shown by placing n points on a line with exponentially decreasing inter point distance from left to right. However we have not been able to find any non-trivial lower or upper bound on the weight of a WSPD-graph.

4 Experimental Results

In this section we discuss the results in more detail by considering each of the five properties. The experiments were done on point sets ranging from 100 to 13.000 points with five different distributions:

- uniform distribution,
- normal distribution with mean 500 and deviation 100,
- gamma distribution with shape parameter 0.75,
- $\frac{n}{100}$ uniformly distributed unit squares with 100 uniformly distributed points, and
- \sqrt{n} uniformly distributed unit squares with \sqrt{n} uniformly distributed points.

In the discussion that follows we will call the point sets produced with the two latter distributions the clustered sets, and the other point sets will be called the non-clustered sets. To avoid the effect of specific instances, we ran the algorithms on many different instances and took the average of the results.

We produced t -spanners using values of t between 1.05 and 2. For larger values of t one can use the Delaunay triangulation which is known to have dilation ≈ 2.42 [12]. The algorithms were implemented in C++ using the LEDA 5.0 library [14].

4.1 Size

The number of edges in the produced graphs were all linear with respect to the number of points and, as expected, the greedy graph had the smallest number of edges. For $t = 2$, $t = 1.1$ and $t = 1.05$ the number of edges in the greedy graph is approximately $2n$, $4n$ and $6n$ respectively, which is surprisingly small. For comparison it is interesting to note that the Delaunay triangulation has approximately $3n$ edges and dilation bounded by 2.42 [12]. A short summary

of the results is that the size of the WSPD-graph is roughly a factor 7 to 13 times greater than the size of the (ordered) Θ -graph which in turn is roughly a factor 5 to 10 times greater than the size of the greedy graph. For the uniform distribution and for $t = 2$ the results can be seen in Fig. 1a. The WSPD algorithm was expected to perform slightly better for clustered sets since it uses a clustering approach, but the improvement was greater than predicted. On clustered sets the WSPD algorithm produced graphs where the number of edges is comparable, or even smaller, than the number of edges in the (ordered) Θ -graph. Especially for small values of t the algorithm performed better.

4.2 Degree

As above, the greedy algorithm again outperformed the other approaches. It is the only algorithm that has a theoretical constant upper bound on its degree, see Theorem 1, and the experiments supports the theory. In the tests the greedy algorithm produced graphs with degree about 5, 7 and 23 for $t = 2$, $t = 1.5$ and $t = 1.05$ respectively and the bounds are roughly the same for all the test sets.

For non-clustered sets the degree of the (ordered) Θ -graphs increases very slowly with respect to the number of points, for example for the uniform distribution the ordered Θ -graph has a degree of 24 for 100 points and the degree then slowly increases to 31 for 10.000 points. The ordered Θ -graph generally performs slightly better than the Θ -graph. However, for clustered sets the results change unexpectedly. The degree of the Θ -graph deteriorate rapidly and the degree varies highly between different instances. The ordered Θ -graph on the other hand performs slightly better than for the other distributions, as shown in Fig. 1b. Again it seems that the experiments supports the theory stated in Theorem 3 since the ordered Θ -graph has $\mathcal{O}(k \log n)$ degree.

As expected the WSPD-algorithm generated the graphs with the highest degree. For small values of t it almost shows a linear behavior for sets with up to 13.000 points. Although for larger values of t it seems to converge slowly, but to be able to draw any distinct conclusions more experiments has to be performed on much larger point sets. But as observed in the previous section, the WSPD-algorithm performs much better on clustered sets and seems to converge to a constant for large point sets. For example for $t = 1.05$ and $t = 1.1$ the degree is bounded by 350 and 290 respectively, and it does not seem to increase.

Finally, we tried to improve the WSPD-graph by considering a modification of the WSPD-algorithm. Instead of adding an arbitrary edge between a well-separated pair, we add an edge between the two endpoints in the pairs with smallest degree. This does not improve the theoretical upper bound but we were hoping to see some improvements in the experimental bounds. There is a small improvement, unfortunately this improvement only shows up for graphs with $t > 1.5$, for smaller values the difference is negligible. For $t = 1.5$ the improvement is roughly a factor 1.5 and increases to about 3 for $t = 4$. Note also that the $\Omega(n)$ lower bound stated in Section 3 does not hold for the modified WSPD-algorithm.

4.3 Weight

Recall that theoretically the weight of the greedy graph is $\mathcal{O}(wt(MST))$ while the weight of the (ordered) Θ -graph and the WSPD-graph is only bounded by $\mathcal{O}(n \cdot wt(MST))$ so the fact that the weight of the greedy graphs is much less than the weight of the other graphs is hardly surprising. For $t = 2$ the weight of the greedy graph is approximately 2 times the $wt(MST)$ and for $t = 1.1$ and $t = 1.05$ the factors are 10 and 18 respectively, as can be seen in Fig. 1c. For the clustered sets the bounds are even slightly better.

For the non-clustered sets the weight of the Θ -graph was unexpectedly small and the ratio between its weight and the $wt(MST)$ increased very slowly. For example for $t = 1.1$ it went from 133 for 100 points to 330 for 13.000 points. An interesting question that we have not been able to answer neither through the experiments nor in theory is if the expected weight of the Θ -graph for uniform point sets is bounded by a constant times the $wt(MST)$. For clustered sets the weight of the Θ -graph is almost linear with respect to the number of clusters and its weight is highly dependent on the different instances.

One would expect a similar behavior from the ordered Θ -graph but the weight of the ordered Θ -graph is much higher than both the greedy graph and the Θ -graph. The ratio between the weight of the ordered Θ -graph and the $wt(MST)$ is almost a linear function with respect to the number of points up to 10.000 points before it starts to level out. Moreover the behavior of the weight of the ordered Θ -graph is unpredictable and seems to be highly dependent on the specific instances.

The weight of the WSPD-graph has the same behavior as the degree of the WSPD-graph. For a small stretch factor it shows a linear behavior compared to the $wt(MST)$ and for larger values of t it seems to converge slowly. Just as for the degree, the WSPD-algorithm performs very well on clustered sets and seems to converge to a large constant times the $wt(MST)$ for large sets. For example for $t = 1.05$ the ratio was bounded by 900 and for, $t = 2$ it was bounded by 230.

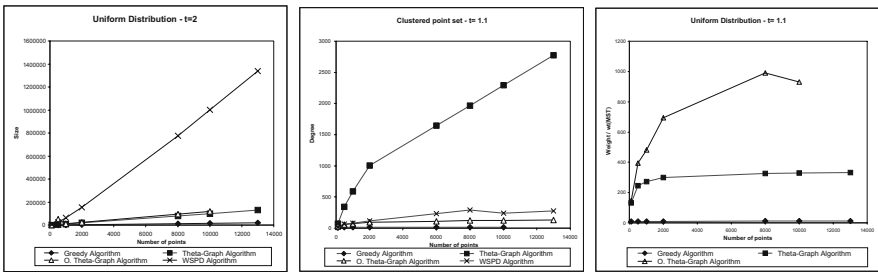


Fig. 1. (a) Size for uniform distribution sets. (b) Degree for clustered sets. (c) Weight/ $wt(MST)$ for uniform distribution sets.

4.4 Diameter

Due to the high complexity of computing the spanner diameter of a graph we could only compute the diameter for graphs with up to 2000 points.

Note that all of the algorithms that we implemented may produce graphs whose diameter is $\Theta(n)$. The only known algorithm that produces graphs with smaller diameter is the modified ordered Θ -graphs which was shown to have $\mathcal{O}(\log n)$ diameter with high probability [3].

As expected the greedy graphs had the highest diameter. This follows from the fact that the greedy graph has fewer edges than the other graphs and the greedy approach favors short edges and avoids adding long edges. The diameter of a 2-spanner generated by the greedy algorithm (normal distribution) is about 16 for a set with 100 points and it reaches 58 for a set with 2000 points. The diameter seems to increase linearly with the size of the input set. Also the diameter changes slightly depending on the different distributions, for $t = 1.1$ and $n = 1000$ the diameter varies between 22 for normal distribution to 36 for the clustered distribution.

The diameter of the (ordered) Θ -graph is much smaller than the greedy graph. A 2-spanner generated by the (ordered) Θ -graph has diameter approximately 5 for a set with 100 points and it increases to 15 for a set with 2000 points. The ordered Θ -graph has generally a slightly lower diameter compared to the Θ -graph. The diameter of the (ordered) Θ -graphs is almost the same for all the distributions.

Finally, the WSPD-graph has very low diameter and it converges fast. The diameter of a 2-spanner on a set with 100 points (normal distribution) is 3.4 and it increases to 5 for a point set with 2.000 points. In the clustered sets, the diameter is slightly higher, between 4 and 7, probably because the number of edges in these graphs are smaller compared to the graphs for the non-clustered sets. An oddity is that the modified WSPD algorithm which add an edge between the two points of well separated pair with smallest degree has slightly lower diameter compared with the WSPD-graphs.

4.5 Crossings

The last property we discuss is the number of crossings which obviously is highly dependent on the number of edges, therefore it is not surprising that the greedy graph is superior to the other graphs and it is the only graph with a reasonable number of crossings. Just as for the diameter the experiments could only be done on sets with up to 2.000 points since the number of crossings only is bounded by $\mathcal{O}(m^2)$ where m is the number of edges in the graph. For $t = 1.5$ and $n = 2.000$ the number of crossings in the greedy graph is on average 94. Also the number of crossings seems to increase linearly with respect to the number of points which is surprisingly low. For $t = 1.1$ the number of crossings for $n = 100, 500$ and 2.000 is on average 1.727, 23.851 and 50.210.

We did some initial experiments with the (ordered) Θ -graphs but the number of crossings were so high that we found the results uninteresting, for example for $t = 1.1$ and $n = 100$ the Θ -graph has 2.437 edges and 300K crossings! Thus

if the number of crossings is a priority to the user then the only option is to use the greedy algorithm.

5 Hybrid Algorithms

As you can see in Section 4, the greedy algorithm produced graphs whose size, weight, degree and number of crossings are superior to the graphs produced from the other approaches. However the running time of the greedy algorithm is $\mathcal{O}(n^3 \log n)$. A way to improve the running time while, hopefully, still obtaining the high-quality graphs is to first compute a t^α -spanner ($0 < \alpha < 1$) G of the input set which contains linear number of edges and then compute a $(t^{1-\alpha})$ -spanner of G using the greedy pruning algorithm. The dilation of the resulting graph is bounded by $t^\alpha \cdot t^{1-\alpha} = t$. The t^α -spanner can be constructed using (ordered) Θ -graphs or WSPD-graphs, ensuring that the number of edges is $\mathcal{O}(n)$, and consequently the total running time would decrease to $\mathcal{O}(n^2 \log n)$.

A second reason why we consider hybrid algorithms is the fact that the (ordered) Θ -graphs and the WSPD-graph actually have much smaller dilation than the specified t -value. For example for $t = 2$ the greedy graph has dilation close to 2 while the ordered Θ -graph and the WSPD-graph has dilation 1.4 and the Θ -graph has dilation 1.2. For $t = 1.1$ the Θ -graph has dilation 1.02, the ordered Θ -graph has dilation 1.06 and the WSPD-graph has dilation 1.04. By first producing the (ordered) Θ -graph or the WSPD-graph we use the fact that they can be constructed fast and the number of edges remaining is linear. Since their dilation in practice is very small it leaves a lot of freedom for the greedy algorithm to produce a t -spanner with good properties.

This approach has another advantage which is that the parameter α can be adjusted to fit the application. If α is chosen to be close to zero then the resulting graph is very similar to the greedy graph but the gain in running time is small. If α is chosen close to 1 then the algorithm is faster but the quality of the graph is worse.

We performed the same experiments using different algorithmic combinations and compared the properties of the generated graphs with the greedy graphs. We implemented the combination of (ordered) Θ -graph and WSPD-graph plus greedy pruning.

The experiments showed the following interesting observations. The graphs generated by the three hybrid algorithms has more or less the same properties, and even though the number of edges, degree and weight is higher their behavior is very similar to the greedy algorithm. This probably follows from the fact that the actual dilation of the (ordered) Θ -graphs and the WSPD-graph is much smaller in reality than the given parameter t^α , thus as expected the resulting graphs are mainly decided by the greedy pruning. For the uniform distribution with $\alpha = 0.9$ and $t = 1.1$ the number of edges and degree of the graphs generated by the hybrid algorithms are roughly a factor 3 greater than the number of edges and degree of the greedy graph. Also the weight is approximately six times greater and the diameter is less than a half that of the greedy graph. Finally, if

the value of α is very small, say 0.1, then the resulting graphs have more or less the same properties as the greedy graphs.

6 Conclusion and Future Work

In short the conclusions from the experiments are as follows:

- The greedy graph has surprisingly good quality when it comes to the number of edges, the weight, the degree and the number of crossings. The diameter of the greedy graph is considerably higher than the diameter of the other graphs.
- The greedy improvement worked out much better than expected and it would be very interesting if one could prove that the improved algorithm has a running time of $\mathcal{O}(n^2 \log n)$ (expected).
- The experiments shows that the weight of the Θ -graph is very small for non-clustered sets. Proving, or disproving, that the expected weight of the Θ -graph for uniform distributions is an interesting and challenging open question.
- The Θ -graph has unexpectedly high degree when the sets are clustered and its degree varies greatly between different instances. Also the degree of the ordered Θ -graph is much smaller, for clustered sets, than the Θ -graph. Surprisingly the same results can not be seen in the weight of the (ordered) Θ -graphs.
- The WSPD-algorithm produces graphs with unexpectedly poor quality for non-clustered sets. For clustered sets the results are much better; the weight and degree of the WSPD-graph are considerably smaller than the weight and degree of the (ordered) Θ -graphs. The weight and the degree of the WSPD-graphs seem to converge for very large data sets. However, to answer this conjecture we need to perform tests on much larger sets.

Future work includes more extensive tests with larger sets to be able to experimentally answer some of the remaining open questions. However the main objective will be to experimentally compare the time complexity of the algorithms.

References

1. I. Althöfer, G. Das, D. P. Dobkin, D. Joseph, and J. Soares. On sparse spanners of weighted graphs. *Discrete & Computational Geometry*, 9(1):81–100, 1993.
2. S. Arya, G. Das, D. M. Mount, J. S. Salowe, and M. Smid. Euclidean spanners: short, thin, and lanky. In *Proc. 27th ACM Symposium on Theory of Computing*, pages 489–498, 1995.
3. P. Bose, J. Gudmundsson, and P. Morin. Ordered theta graphs. *Computational Geometry: Theory and Applications*, 28:11–18, 2004.
4. P. B. Callahan and S. R. Kosaraju. A decomposition of multidimensional point sets with applications to k -nearest-neighbors and n -body potential fields. *Journal of the ACM*, 42:67–90, 1995.

5. K. L. Clarkson. Approximation algorithms for shortest path motion planning. In *Proc. 19th ACM Symposium on Computational Geometry*, pages 56–65, 1987.
6. A. Czumaj and A. Lingas. Fast approximation schemes for Euclidean multi-connectivity problems. In *Proc. 27th International Colloquium on Automata, Languages and Programming*, volume 1853 of *Lecture Notes in Computer Science*, pages 856–868. Springer-Verlag, 2000.
7. D. Eppstein. Spanning trees and spanners. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 425–461. Elsevier Science Publishers, Amsterdam, 2000.
8. A. M. Farley, A. Proskurowski, D. Zappala, and K. J. Windisch. Spanners and message distribution in networks. *Discrete Applied Mathematics*, 137(2):159–171, 2004.
9. J. Gudmundsson, C. Levcopoulos, and G. Narasimhan. Improved greedy algorithms for constructing sparse geometric spanners. *SIAM Journal of Computing*, 31(5):1479–1500, 2002.
10. J. Gudmundsson, C. Levcopoulos, G. Narasimhan, and M. Smid. Approximate distance oracles for geometric graph. In *Proc. 13th ACM-SIAM Symposium on Discrete Algorithms*, 2002.
11. J. M. Keil. Approximating the complete Euclidean graph. In *Proc. 1st Scandinavian Workshop on Algorithmic Theory*, pages 208–213, 1988.
12. J. M. Keil and C. A. Gutwin. Classes of graphs which approximate the complete Euclidean graph. *Discrete and Computational Geometry*, 7:13–28, 1992.
13. X.-Y. Li. Applications of computational geometry in wireless ad hoc networks. In X.-Z. Cheng, X. Huang, and D.-Z. Du, editors, *Ad Hoc Wireless Networking*. Kluwer, 2003.
14. K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, UK, 2000.
15. G. Navarro and R. Paredes. Practical construction of metric t -spanners. In *Proc. 5th Workshop on Algorithm Engineering and Experiments*, pages 69–81. SIAM Press, 2003.
16. G. Navarro, R. Paredes, and E. Chvez. t -spanners as a data structure for metric space searching. In *Proc. 9th International Symposium on String Processing and Information Retrieval*, volume 2476 of *Lecture Notes in Computer Science*, pages 298–309. Springer-Verlag, 2002.
17. Helen C. Purchase. Which aesthetic has the greatest effect on human understanding? In *Graph Drawing*, volume 1353 of *Lecture Notes in Computer Science*, pages 248–261. Springer-Verlag, 1997.
18. S. Rao and W. D. Smith. Approximating geometrical graphs via spanners and banyans. In *Proc. 30th ACM Symposium on the Theory of Computing*, pages 540–550. ACM, 1998.
19. J. Ruppert and R. Seidel. Approximating the d -dimensional complete Euclidean graph. In *Proc. 3rd Canadian Conference on Computational Geometry*, pages 207–210, 1991.
20. M. Sigurd and M. Zachariasen. Construction of minimum-weight spanners. In *Proc. 12th European Symposium on Algorithms*, number 3221 in *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

Highway Hierarchies Hasten Exact Shortest Path Queries

Peter Sanders^{1,*} and Dominik Schultes^{1,2}

¹ Universität Karlsruhe (TH), 76128 Karlsruhe, Germany
sanders@ira.uka.de

² Universität des Saarlandes
mail@dominik-schultes.de

Abstract. We present a new speedup technique for route planning that exploits the hierarchy inherent in real world road networks. Our algorithm preprocesses the eight digit number of nodes needed for maps of the USA or Western Europe in a few hours using linear space. Shortest (i.e. fastest) path queries then take around eight milliseconds to produce exact shortest paths. This is about 2 000 times faster than using Dijkstra’s algorithm.

1 Introduction

Computing shortest paths in graphs (networks) with nonnegative edge weights is a classical problem of computer science. From a worst case perspective, the problem has largely been solved by Dijkstra in 1959 [1] who gave an algorithm that finds all shortest paths from a starting node s using at most $m + n$ priority queue operations for a graph $G = (V, E)$ with n nodes and m edges.

However, motivated by important applications (e.g., in transportation networks), there has recently been considerable interest in the problem of accelerating *shortest path queries*, i.e., the problem to find a shortest path between a source node s and a target node t . In this case, Dijkstra’s algorithm can stop as soon as the shortest path to t is found.

A classical technique that gives a constant factor speedup is *bidirectional search* which simultaneously searches forward from s and backwards from t until the search frontiers meet. All further speedup techniques either need additional information (e.g., geometry information for *goal directed search*) or *precomputation*. There is a trade-off between the time needed for *precomputation*, the *space* needed for storing the precomputed information, and the resulting *query time*. In Section 1 we review existing precomputation approaches, which have made significant progress, but still fall short of allowing fast *exact* shortest path queries in very large graphs.

In particular, from now on we focus on shortest paths in large *road networks* where we use ‘shortest’ as a synonym for ‘fastest’. The graphs used for North

* Partially supported by DFG grant SA 933/1-2.

America or Western Europe already have around 20 000 000 nodes so that significantly superlinear preprocessing time or even slightly superlinear space is prohibitive. To our best knowledge, all commercial applications currently only compute paths heuristically that are not always shortest possible. The basic idea of these heuristics is the observation that shortest paths “usually” use small roads only locally, i.e., at the beginning and at the end of a path. Hence the heuristic algorithm only performs some kind of *local search* from s and t and then switches to search in a *highway network* that is much smaller than the complete graph. Typically, an edge is put into the highway network if the information supplied on its road type indicates that it represents an important road.

Our approach is based on the idea to compute *exact* shortest paths by defining the notion of *local search* and *highway network* appropriately. This is very simple. We define local search to be a search that visits the H closest nodes from s (or t) where H is a tuning parameter. This definition already fixes the highway network. An edge $(u, v) \in E$ should be a highway edge if there are nodes s and t such that (u, v) is on the shortest path from s to t , v is not within the H closest nodes from s , and u is not within the H closest nodes from t . Section 2 gives a more formal definition of the basic concepts used in this paper.

One might think that an expensive all-pairs shortest path computation is needed to find the highway network. However, in Section 3 we show that each highway edge is also within some local shortest path tree B rooted at some $s \in V$ such that all leaves of B are “sufficiently far away” from s .

So far, the highway network still contains all the nodes of the original network. However, we can prune it significantly: Isolated nodes are not needed. Trees attached to a biconnected component can only be traversed at the beginning and end of a path. Similarly, paths consisting of nodes with degree two can be replaced by a single edge. The result is a *contracted highway network* that only contains nodes of degree at least three. We can iterate the above approach, define local search on the highway network, find a “superhighway network”, contract it, . . . We arrive at a multi-level highway network — a *highway hierarchy*.

Section 4 develops a query algorithm that uses highway hierarchies. After several correctness preserving transformations we get a bidirectional, Dijkstra-like search in a single graph that contains all levels. The only modifications affect the selection of edges to be relaxed and how to finish the search when the search frontiers from s and t meet.

In Section 5 we summarise experiments using detailed road networks for Western Europe and the USA. Using a uniform neighbourhood size of 125 and 225 respectively, the graphs shrink geometrically from level to level. This leads to preprocessing time around four hours and average query times below 8 ms. Possible future improvements are discussed in Section 6. Proofs and additional experimental data can be found in the full version at <http://www.dominik-schultes.de/hwy/>.

Related Work

There is so much literature on shortest paths and preprocessing that we can only highlight selected results that help to put our results into perspective. In the following, speedup refers to the acceleration compared to the unidirectional

variant of Dijkstra's algorithm that stops when the target is found. For recent, more detailed overviews we refer to [2,3].

Perhaps the most interesting theoretical results on route planning are algorithms for planar graphs that might be adaptable to route networks since those are "almost planar". Using $O(n \log^3 n)$ space and preprocessing time, query time $O(\sqrt{n} \log n)$ can be achieved [4] for directed planar graphs without negative cycles. Queries accurate within a factor $(1 + \epsilon)$ can be answered in near constant time using $O((n \log n)/\epsilon)$ space and preprocessing time [5]. However, for very large graphs we need linear space consumption so that these approaches seem not directly applicable to the problem at hand.

The previous practical approach closest to ours is the *separator based multi-level method* [6]. The idea is to partition the graph into small subgraphs by removing a (hopefully small) set of separator nodes. These separator nodes together with edges representing precomputed paths between them constitute the next level of the graph. Queries then only need to search in the partitions of s and t and in the higher level graph. This process can be iterated. At least for road networks speedups so far seem to be limited to a factor around ten whereas better speedup can be observed for railway transportation problems [6]. Disadvantages compared to our method are that performance depends on very small (and thus hard to find) separators and that the higher level graphs get quite dense so that going to many levels quickly reaches a point of diminishing return. In contrast, our method has a very simple definition of what constitutes the higher level graphs and our higher level graphs remain sparse.

Reach based routing [7] excludes nodes from consideration if they do not contribute to any path long enough to be of use for the current query. Speedups up to 20 are reported for graphs with about 400 000 nodes using about 2 hours preprocessing time. Our method is an order of magnitude faster both in terms of query time and in terms of preprocessing time.

Most other preprocessing techniques are different from our approach in that they focus the search towards the target. Very high speedups (hundreds or even around 2000) are reported for *geometric containers* [8,3] and *bit vectors* [9,10]. Both methods store information with each edge. Queries use this information to decide whether this edge can possibly lead to the target. Our method achieves similar speedups but needs much less time for preprocessing: To compute k -bit vectors, $O(\sqrt{kn})$ global shortest path searches are needed (assuming planar graphs) and geometric containers even need an all-pairs computation.

An interesting alternative are *landmark* based lower bounds for strengthening goal directed search [2]. For global queries, about 16 global shortest path computations during preprocessing suffice to achieve speedup around 20. However, the landmark method needs a lot of space — one distance value for each node-landmark pair. It is also likely that for real applications each node will need to store distances to different sets of landmarks for global and local queries. Hence, landmarks have very fast preprocessing and reasonable speedups but consume too much space for very large networks.

2 Preliminaries

We expect an *undirected* graph $G = (V, E)$ with n nodes and m edges with nonnegative weights as input.¹ We assume w.l.o.g. that there are no self-loops, parallel edges, or zero weight edges in the input — they could be dealt with easily in a preprocessing step. The *length* $w(P)$ of a path P is the sum of the weights of the edges that belong to P . $P^* = \langle s, \dots, t \rangle$ is a *shortest path* if there is no path P' from s to t such that $w(P') < w(P^*)$. The *distance* $d(s, t)$ between s and t is the length of a shortest path from s to t . If $P = \langle s, \dots, s', u_1, u_2, \dots, u_k, t', \dots, t \rangle$ is a path from s to t , then $P|_{s' \rightarrow t'} = \langle s', u_1, u_2, \dots, u_k, t' \rangle$ denotes the *subpath* of P from s' to t' .

Dijkstra's Algorithm. In the context of Dijkstra's algorithm, we use the following terminology: each node is either *unreached*, *reached*, or *settled*. If a node u is reached, at least one path (not necessarily the shortest one) from the source node s to u has been found and u has been inserted into the priority queue. If a node v is *settled*, it is reached and has been removed from the priority queue by a *deleteMin* operation; a shortest path from s to v has been found.

Canonical Shortest Paths. A *selection of shortest paths* \mathcal{SP} contains for each connected pair $(s, t) \in V \times V$ exactly one shortest path from s to t . Such a selection is called *canonical* if $P = \langle s, \dots, s', \dots, t', \dots, t \rangle \in \mathcal{SP}$ implies that $P|_{s' \rightarrow t'} \in \mathcal{SP}$. The elements of a canonical selection are called *canonical shortest paths*. If Dijkstra's algorithm is started from each node $s \in V$, for each connected pair (s, t) exactly one shortest path is determined. In the full paper some modifications of Dijkstra's algorithm are described which ensure that the obtained selection of shortest paths is canonical.

Locality. Let us fix any rule that decides which element Dijkstra's algorithm removes from the priority queue when there is more than one queued element with the smallest key. Then, during a Dijkstra search from a given node s , all nodes are settled in a fixed order. The *Dijkstra rank* $r_s(v)$ of a node v is the rank of v w.r.t. this order. s has Dijkstra rank $r_s(s) = 0$, the closest neighbour v_1 of s has Dijkstra rank $r_s(v_1) = 1$, and so on. For a given node s , the distance of the H -closest node from s is denoted by $d_H(s)$, i.e., $d_H(s) = d(s, v)$, where $r_s(v) = H$. The *H -neighbourhood* $\mathcal{N}_H(s)$ (or just *neighbourhood* $\mathcal{N}(s)$) of s is $\mathcal{N}(s) := \{v \in V \mid d(s, v) \leq d_H(s)\}$.²

Highway Hierarchy. For a given parameter H , the *highway network* $G_1 = (V_1, E_1)$ of a graph G is defined as the set of edges $(u, v) \in E$ that appear in a canonical shortest path $\langle s, \dots, u, v, \dots, t \rangle$ from a node $s \in V$ to a node

¹ Unless otherwise stated, we always deal with *undirected* edges. The restriction to undirected graphs simplifies the presentation of our approach and the implementation. However, our method can be generalised to *directed* graphs.

² For directed graphs we also need an analogous value $\bar{d}_H(\cdot)$ that refers to the reverse graph $\bar{G} := (V, \{(v, u) \mid (u, v) \in E\})$. $\bar{\mathcal{N}}(\cdot)$ is defined correspondingly. From now on, whenever the target node t or the backward search from t is concerned, we have to keep in mind that \bar{G} , $\bar{d}_H(\cdot)$, and $\bar{\mathcal{N}}(\cdot)$ apply.

$t \in V$ with the property that $v \notin \mathcal{N}_H(s)$ and $u \notin \mathcal{N}_H(t)$. The set V_1 is the maximal subset of V such that G_1 contains no isolated nodes.

The *2-core* of a graph is the maximal vertex induced subgraph with minimum degree two. A graph consists of its 2-core and *attached trees*, i.e., trees whose roots belong to the 2-core, but all other nodes do not belong to it. A *line* in a graph is a path $\langle u_0, u_1, \dots, u_k \rangle$ where the inner nodes u_1, \dots, u_{k-1} have degree two. From the highway network G_1 of a graph G , the *contracted highway network* G'_1 of the graph G is obtained by taking the 2-core of G_1 and, then, removing the inner nodes of all lines $\langle u_0, u_1, \dots, u_k \rangle$ and replacing each line by an edge (u_0, u_k) . Thus, the highway network G_1 consists of the contracted highway network (or short, just *core*) G'_1 and some *components*, where ‘component’ is used as a generic term for ‘attached tree’ and ‘line’. In this paper, ‘components’ is used always in this specific sense and *not* to denote ‘connected components’ in general.

The *highway hierarchy* is obtained by applying the process that leads from G to G'_1 iteratively. The original graph $G_0 := G'_0 := G$ constitutes Level 0 of the highway hierarchy, G_1 corresponds to Level 1, the highway network G_2 of the graph G'_1 is called Level 2, and so on.

3 Construction

For each node $s_0 \in V$, we compute and store the value $d_H(s_0)$. This can be easily done by a Dijkstra search from each node s_0 that is aborted as soon as H nodes have been settled. Then, we start with an empty set of highway edges E_1 . For each node s_0 , two phases are performed: the forward construction of a partial shortest path tree B and the backward evaluation of B . The construction is done by a single source shortest path (SSSP) search from s_0 ; during the evaluation phase, paths from the leaves of B to the root s_0 are traversed and for each edge on these paths, it is decided whether to add it to E_1 or not. The crucial part is the specification of an abort criterion for the SSSP search in order to restrict it to a ‘local search’.

Phase 1: Construction of a Partial Shortest Path Tree. A Dijkstra search from s_0 is executed. During the search, a reached node is either in the state *active* or *passive*. The source node s_0 is active; each node that is reached for the first time (*insert*) and each reached node that is updated (*decreaseKey*) adopts the activation state from its (tentative) parent in the shortest path tree B . When a node p is settled using the path $\langle s_0, s_1, \dots, p \rangle$, then p ’s state is set to passive if $|\mathcal{N}(s_1) \cap \mathcal{N}(p)| \leq 1$. When no active unsettled node is left, the search is aborted and the growth of B stops.

Phase 2: Selection of the Highway Edges. During Phase 2, all edges (u, v) are added to E_1 that lie on paths $\langle s_0, \dots, u, v, \dots, t_0 \rangle$ in B with the property that $v \notin \mathcal{N}(s_0)$ and $u \notin \mathcal{N}(t_0)$, where t_0 is a leaf of B . This can be done in time $O(|B|)$.

Theorem 1. *An edge $(u, v) \in E$ is added to E_1 by the construction algorithm iff it belongs to some canonical shortest path $P = \langle s, \dots, u, v, \dots, t \rangle$ and $v \notin \mathcal{N}(s)$ and $u \notin \mathcal{N}(t)$.*

Speeding Up Construction. An active node v is declared to be a *maverick* if $d(s_0, v) > f \cdot d_H(s_0)$, where f is a parameter. When all active nodes are mavericks, the search from passive nodes is no longer continued. This way, the construction process is accelerated and E_1 becomes a superset of the highway network. Hence, queries will be slower, but still compute exact shortest paths. The *maverick factor* f enables us to adjust the trade-off between construction and query time.

Theorem 2. *The highway network can be contracted in time $O(m + n)$.*

Highway Hierarchy. The result of the contraction is the contracted highway network G'_1 , which can be used as input for the next iteration of the construction procedure in order to obtain the next level of the highway hierarchy.

4 Query

The *highway hierarchy* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ consists of the graphs $G_0, G_1, G_2, \dots, G_L$, which are arranged in $L + 1$ levels. For each node $v \in V$ and each $i \in \{j \mid v \in V_j\}$, there is one copy of v , namely v_i , that belongs to level i of \mathcal{G} . Accordingly, there are several copies of an edge (u, v) when u and v belong to more than one common level. These edges, which connect two nodes in the same level, are called *horizontal* edges. Additionally, \mathcal{G} contains a directed edge $(v_\ell, v_{\ell+1})$ for each pair $v_\ell \in V_\ell, v_{\ell+1} \in V_{\ell+1}$, where v_ℓ and $v_{\ell+1}$ are copies of the same node v . These additional edges are called *vertical* and have weight 0. For each node v , not only one value $d_H(v)$ is known, but for each level $\ell < L$, there is a distance $d_H^\ell(v)$ from v to the H -closest node in the graph G'_ℓ ; if a node v does not belong to G'_ℓ , $d_H^\ell(v)$ is defined to be $+\infty$; furthermore, $d_H^L(v) := +\infty$. Correspondingly, we use the notation $\mathcal{N}^\ell(v)$ to refer to the set $\{v' \in V'_\ell \mid d(v, v') \leq d_H^\ell(v)\}$, which is the *neighbourhood* of v in the graph G'_ℓ . Note that the neighbourhood of a node that belongs to a component is unbounded, i.e., it contains all nodes of the core of the corresponding level. The same applies to $\mathcal{N}^L(v)$, for any v .

The *multilevel query algorithm* that works on \mathcal{G} is a modification of the bidirectional version of Dijkstra’s algorithm. The source and target nodes of an s - t query are the corresponding copies of s and t in level 0. For the time being, we omit the abort-on-success criterion, i.e., we do not abort when both search scopes meet, but continue until both searches terminate; then, we consider all nodes that have been settled from both sides as meeting points and take the shortest path that has been found by this means. The modifications consist of two restrictions:

1. *In each level ℓ , no horizontal edge is relaxed that would leave the neighbourhood $\mathcal{N}^\ell(v^*)$ of the corresponding entrance point v^* .* Each node that belongs to the core and has been settled via a horizontal edge that leaves a component and each node that has been settled via a vertical edge is an *entrance point*. In addition, the source and the target nodes of the query are entrance points. The *corresponding entrance point* of a settled node v is the last entrance point on the path to v .

2. *Components are never entered using a horizontal edge.* An edge (u, v) enters a component if either u belongs to the core and v to a component or u belongs to a line and v to an attached tree. However, an edge from an attached tree to a line leaves the attached tree and does not rank among the edges that enter a component. Note that the endpoint(s) of a component do not belong to the component but to the core (or to the line in case of the root of a tree that is attached to a line).

Theorem 3. *For any given $s, t \in V$, the multilevel query algorithm finds the shortest path from s to t in G .*

Proof Idea. It is known that the bidirectional version of Dijkstra's algorithm works correctly. We have to show that the imposed restrictions do not affect the correctness. When Restriction 1 applies, it is always possible to switch to the next level using a vertical edge. Due to the definition of the highway network, it is guaranteed that the corresponding part of the shortest path which we are looking for can be found in the next level. A path from s that enters a component is not traversed due to Restriction 2. However, from the point of view of t , this path leaves the component so that the edge that has been skipped during the search from s can be relaxed in the reverse direction during the search from t . Hence, the path can be found in spite of Restriction 2. These arguments can be used in an inductive proof over the number of levels. \square

Collapse of the Vertical Dimension. So far, we allow that several copies of the same node are reached. However, it can be shown that it is sufficient if at most one copy of a node is reached via a horizontal edge, namely the copy with the smallest tentative distance or, if there are several copies with the same smallest tentative distance, the copy in the lowest level.

Due to this observation, we can let the vertical dimension collapse. We can interpret the highway hierarchy \mathcal{G} as one plain graph, i.e., there are no copies of the nodes distributed over several levels. Basically, this graph corresponds to the original graph G enhanced by some additional data: each edge (u, v) is assigned a maximum level $\ell(u, v)$, i.e., it belongs to the levels $0, 1, \dots, \ell(u, v)$; each node v is assigned to at most one component $c(v)$; a component $c(v)$ belongs to a certain level $\ell(c(v))$, which is equal to the level its inner edges belong to. Furthermore, the value $d_H^\ell(v)$ is stored only if $v \in G'_\ell$. Our implementation is based on this interpretation of \mathcal{G} .

Abort-on-Success. In the bidirectional version of Dijkstra's algorithm, we can abort as soon as both search scopes meet, i.e., there is one node v that is settled in both search scopes. Then, the shortest path P from s to t does not necessarily consist of the shortest paths from s to v and from v to t , but it is well known that it is always ensured that the right meeting point v' has already been reached from both sides. The crucial precondition for this fact is that all nodes whose distance from s is less than $d(s, v)$ have been settled in the search scope of s , and all nodes whose distance from t is less than $d(t, v)$ have been settled in the search scope of t .

Unfortunately, we cannot adopt the abort-on-success criterion as it stands because, in general, the multilevel query algorithm does not fulfil this precondition as several edges are not relaxed due to Restriction 1 and 2 so that we cannot guarantee that all nodes up to a certain distance have been settled. We have already shown that the algorithm is correct all the same because if an ‘important’ edge is not relaxed (e.g. a component is not entered), then it is relaxed from the other side (e.g. the component is left). However, we have to *wait* until the reverse search has relaxed this edge, i.e., we must not abort too early. Nevertheless, even if we have skipped an edge $e = (u, v)$ at node u , we *can* abort after both search scopes have met as soon as it is certain that e will not be relaxed from v during the final steps of the search. We can use the following approach. Let \mathcal{E}_s denote the set of all horizontal edges that have been skipped during the search from s . \mathcal{E}_t is defined accordingly. After both search scopes have met, we can abort as soon as the search from t has finished search level $\hat{\ell}_s := \max_{e \in \mathcal{E}_s} \ell(e)$ and the search from s has finished level $\hat{\ell}_t := \max_{e \in \mathcal{E}_t} \ell(e)$. A search level ℓ is *finished* when there are no reached but unsettled nodes in level ℓ or below. If the search level ℓ is finished, edges e in levels $\ell(e) \leq \ell$ cannot be relaxed any longer. Hence, when the search from t has finished search level $\hat{\ell}_s$, it is certain that no edge e that belongs to a level $\ell(e) \leq \hat{\ell}_s$ will be relaxed during the final steps of the search, in particular, no edge that has been skipped during the search from s will be traversed by the backward search. There are several possibilities to further improve this abort criterion, which are described in the full paper.

5 Experiments

Implementation. We use a binary heap priority queue. Our current implementation leaves room for reducing both running time and memory usage.

Environment. The experiments were done on a 64-bit machine with 8 GB main memory and 1 MB L2 cache, using one out of four AMD Opteron processors clocked at 2.2 GHz, running SuSE Linux (kernel 2.6.5). The program was compiled by the GNU C++ compiler 3.3.3 using optimisation level 3.

Instances. Basically, we deal with two test instances, namely, the road networks of the United States of America (minus Alaska and Hawaii) and of Western Europe. The former was obtained from the TIGER/Line Files [11] by merging the relevant data of all counties. The latter contains the 14 European countries Austria, Belgium, Denmark, France, Germany, Italy, Luxembourg, the Netherlands, Norway, Portugal, Spain, Sweden, Switzerland, and the UK. The data has been made available for scientific use by the company PTV AG. In some cases, we restrict our experiments to the German road network. In all cases, as we deal with undirected graphs, we ignored the restrictions caused by one-way streets.

The original graphs contain for each edge a length and a road category, e.g., motorway, national road, regional road, urban street. We assign average speeds to the road categories, compute for each edge the average travel time, and use it as weight. Table 1 summarises important properties of the used road networks and the key results of the experiments.

Table 1. Overview of the used road networks and key results. The parameter H is used iteratively until the construction leads to an empty highway network. We provide average values for 10 000 queries, where the source and target nodes are chosen randomly. ‘Speedup’ refers to a comparison with Dijkstra’s algorithm³. ‘Efficiency’ [2] denotes the number of nodes that belong to the computed shortest paths divided by the number of nodes that are settled by the multilevel query algorithm. For Germany, we give the memory usage on a 32-bit machine in parentheses.

		USA	Europe	Germany
input	#nodes	24 278 285	18 029 721	4 345 567
	#edges	29 106 596	22 217 686	5 446 916
	#degree 2 nodes	7 316 573	2 375 778	604 540
	#road categories	4	13	13
parameters	average speeds [km/h]	40–100	10–130	10–130
	H	225	125	100
construction	CPU time [h:min]	4:15	2:41	0:30
	#levels	7	11	11
query	CPU time [ms]	7.04	7.38	5.30
	#settled nodes	3 912	4 065	3 286
	speedup (CPU time)	2 654	2 645	680
	speedup (#settled nodes)	3 033	2 187	658
	efficiency	113%	34%	13%
	main memory usage [MB]	2 443	1 850	466 (346)

Fast vs. Precise Construction. During various experiments, we came to the conclusion that it is a good idea *not* to take a fixed maverick factor f for all levels of the construction process, but to start with a low value (i.e. fast construction) and increase it level by level (i.e. more precise construction). For the following experiments, we used the sequence 0, 2, 4, 6, . . .

Best Neighbourhood Sizes. For two levels ℓ and $\ell + 1$ of a highway hierarchy, the *shrinking factor* is the ratio between $|E'_\ell|$ and $|E'_{\ell+1}|$. In our experiments, we observed that the highway hierarchies of the USA and Europe were almost *self-similar* in the sense that the shrinking factor remained nearly unchanged from level to level when we used the same neighbourhood size H for all levels. We kept this approach and applied the same H iteratively until the construction led to an empty highway network. Figure 1 demonstrates the shrinking process for Europe. For most levels, we observe an almost constant shrinking factor (which appears as a straight line due to the logarithmic scale of the y-axis). The greater the neighbourhood size, the greater the shrinking factor. The first iteration (Level 0→1) and the last few iterations are exceptions: at the first iteration, the construction works very well due to the characteristics of the real world road network (there are many trees and lines that can be contracted); at the last iterations, the highway network collapses, i.e., it shrinks very fast, because nodes that are close to the border of the network usually do not belong

³ The averages for Dijkstra’s algorithm are based on only 1 000 queries.

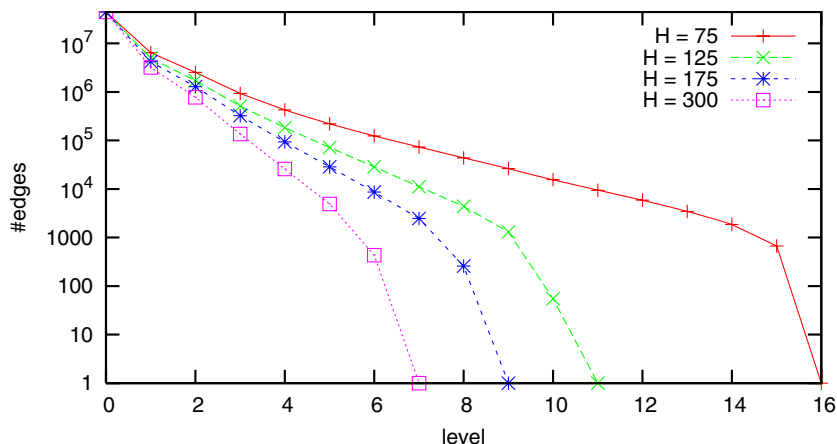


Fig. 1. Shrinking of the highway networks of Europe. For different neighbourhood sizes H and for each level ℓ , we plot $|E'_\ell|$, i.e., the number of edges that belong to the core of level ℓ .

to the next level of the highway hierarchy, and when the network gets small, almost all nodes are close to the border.

Multilevel Queries. Table 1 contains average values for queries, where the source and target nodes are chosen randomly. For the two large graphs we get a speedup of more than 2 000 compared to Dijkstra's algorithm both with respect to (query) time⁴ and with respect to the number of settled nodes.

For our largest road network (USA), the number of nodes that are settled during the search is *less* than the number of nodes that belong to the shortest paths that are found. Thus, we get an efficiency that is greater than 100%. The reason is that edges at high levels will often represent long paths containing many nodes.⁵

For use in applications it is unrealistic to assume a uniform distribution of queries in large graphs such as Europe or the USA. On the other hand, it would be hardly more realistic to arbitrarily cut the graph into smaller pieces. Therefore, we decided to measure local queries within the big graphs: For each power of two $r = 2^k$, we choose random sample points s and then use Dijkstra's algorithm to find the node t with Dijkstra rank $r_s(t) = r$. We then use our algorithm to make an s - t query. By plotting the resulting statistics for each value $r = 2^k$, we can see how the performance scales with a natural measure of difficulty of the query. Figure 2 shows the query times. Note that the median

⁴ It is likely that Dijkstra would profit more from a faster priority queue than our algorithm. Therefore, the time-speedup could decrease by a small constant factor.

⁵ The reported query times do not include the time for expanding these paths. We have made measurements with a naive recursive expansion routine which never take more than 50% of the query time. Also note that this process could be radically sped up by precomputing unpacked representations of edges.

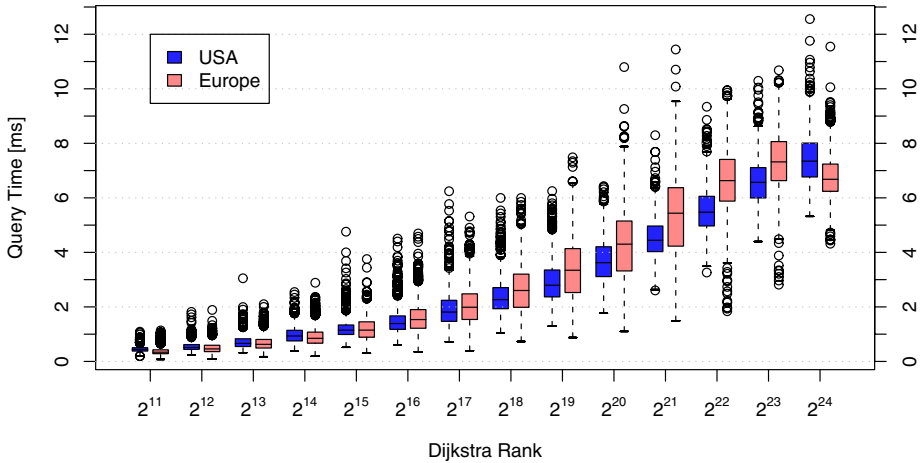


Fig. 2. Multilevel Queries. For each road network and each Dijkstra rank on the x-axis, 1 000 queries from random source nodes were performed. The results are represented as box-and-whisker plot [12]: each box spreads from the lower to the upper quartile and contains the median, the whiskers extend to the minimum and maximum value omitting outliers, which are plotted individually.

query times are scaling quite smoothly and the growth is much slower than the exponential increase we would expect in a plot with logarithmic x axis, linear y axis, and any growth rate of the form r^ρ for Dijkstra rank r and some constant power ρ . The curve is also not the straight line one would expect from a query time logarithmic in r .

6 Discussion

Starting from a simple definition of local search, we have developed nontrivial algorithms for constructing and querying highway hierarchies. We have demonstrated that highway hierarchies of the largest road networks currently used can be constructed in a few hours, i.e., fast enough to allow daily updates. The space consumption is only a small constant factor of the input size. The query times around 10 ms are more than fast enough for interactive use. The only previous speedup techniques that would achieve comparable speedup (bit vectors, geometric containers) have prohibitive preprocessing times for very large graphs.

Even faster preprocessing is a major issue for future work. We see many small (and not so small) opportunities for improvement. The local nature of preprocessing makes it likely that highway hierarchies can be quickly updated dynamically when only a few edges (e.g., for taking traffic jams into account) or a region of the network changes. We can also easily parallelise preprocessing.

Even faster queries are also interesting. For example, for some traffic simulations, millions of shortest paths queries are needed and there is no overhead for a user interface. Besides many small improvements (faster priority queues...)

a combination with other speedup techniques seems interesting. In particular, bit vectors, geometric containers, or landmarks give the search a strong sense of direction that highway hierarchies lack, i.e., these two basic approaches may complement one another. Moreover, the higher levels of the hierarchy are so small that superlinear time or space may be tolerable as long as the contributions of the lower levels can be incorporated efficiently.

Acknowledgements

We would like to thank Andrew Goldberg, Rolf Möhring, Matthias Müller-Hannemann, Heiko Schilling, Frank Schulz, Mikkel Thorup, and Dorothea Wagner for interesting discussions on various speedup techniques. Martin Holzer, Domagoj Matijevic, Frank Schulz, and Thomas Willhalm have also helped with data and tools for processing graphs.

References

1. Dijkstra, E.W.: A note on two problems in connexion with graphs. *Numerische Mathematik* **1** (1959) 269–271
2. Goldberg, A.V., Harrelson, C.: Computing the shortest path: A^* meets graph theory. In: 16th ACM-SIAM Symposium on Discrete Algorithms. (2005) 156–165
3. Willhalm, T.: Engineering Shortest Path and Layout Algorithms for Large Graphs. PhD thesis, Technische Universität Karlsruhe (2005)
4. Fakcharoenphol, J., Rao, S.: Negative weight edges, shortest paths, near linear time. In: 42nd Symposium on Foundations of Computer Science. (2001) 232–241
5. Thorup, M.: Compact oracles for reachability and approximate distances in planar digraphs. In: 42nd Symposium on Foundations of Computer Science. (2001) 242–251
6. Schulz, F., Wagner, D., Zaroliagis, C.D.: Using multi-level graphs for timetable information. In: 4th Workshop on Algorithm Engineering and Experiments. Volume 2409 of LNCS., Springer (2002) 43–59
7. Gutman, R.: Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In: 6th Workshop on Algorithm Engineering and Experiments (ALENEX). (2004)
8. Wagner, D., Willhalm, T.: Geometric speed-up techniques for finding shortest paths in large sparse graphs. In: 11th European Symposium on Algorithms. Volume 2832 of LNCS., Springer (2003) 776–787
9. Lauther, U.: An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. In: Proc. Münster GI-Days. (2004)
10. Köhler, E., Möhring, R.H., Schilling, H.: Acceleration of shortest path and constrained shortest path computation. In: 4th International Workshop on Efficient and Experimental Algorithms. (2005)
11. U.S. Census Bureau, Washington, DC: UA Census 2000 TIGER/Line Files. http://www.census.gov/geo/www/tiger/tigerua/ua_tgr2k.html (2002)
12. The R Development Core Team: R: A Language and Environment for Statistical Computing, Reference Index. <http://www.r-project.org/> (2004)

Preemptive Scheduling of Independent Jobs on Identical Parallel Machines Subject to Migration Delays*

Aleksei V. Fishkin¹, Klaus Jansen², Sergey V. Sevastyanov³, and René Sitters¹

¹ Max-Planck-Institute für Informatik, 66123 Saarbrücken, Germany
{avf, sitters}@mpi-sb.mpg.de

² Institute of Computer Science, University of Kiel, 24118 Kiel, Germany
kj@informatik.uni-kiel.de

³ Sobolev Institute of Mathematics, 630090 Novosibirsk, Russia
seva@math.nsc.ru

Abstract. We present hardness and approximation results for the problem of scheduling n independent jobs on m identical parallel machines subject to a migration delay d so as to minimize the makespan. We give a sharp threshold on the value of d for which the complexity of the problem changes from polynomial time solvable to NP-hard. We give initial results supporting a conjecture that there always exists an optimal schedule in which at most $m - 1$ jobs migrate. Further, we give a $O(n)$ time $O(1 + 1/\log_2 n)$ -approximation algorithm for $m = 2$, and show that there is a polynomial time approximation scheme for arbitrary m .

Keywords: scheduling, identical machines, preemption, migration delay.

We consider the problem of scheduling independent jobs on identical parallel machines subject to migration delays so as to minimize the makespan. Formally, there are m machines, M_1, M_2, \dots, M_m which are used to process n jobs, J_1, J_2, \dots, J_n . Each job J_j ($j = 1, \dots, n$) has a processing time p_j . Each machine can process at most one job at a time, and each job can be processed by at most one machine at a time. Preemptions with a migration delay d are allowed, that is, the processing of any job J_j on a machine M_i can be interrupted and resumed at any later time on M_i , and at least d time units later if J_j migrates to another machine M_k . The goal is to schedule the jobs so as the makespan is minimized. By extending the three-field notion [1,2], we denote this problem by $\alpha|pmtn(\text{delay} = d)|C_{\max}$, where α is either Pm (fixed number m of machines), or P (arbitrary number of machines).

In classical scheduling models the migration of a job is done without any delay constraint. However, in production planning, for example, it is natural to reserve some time for the transition of a product from one machine to another.

* Most of the work has been done within two short-term visits at MPI für Informatik in Saarbrücken, supported in part by the EU project CRESCCO and the Russian Foundation for Basic Research (grant no. 05-01-00960).

Simply the transportation of the product takes time, but also technical issues might make it necessary to wait for some time; a heated product might need to cool down first, or a product needs to dry before its next operation can start. We refine the classical model of identical machines by adding a delay constraint d , which is independent of the job and machines.

Our model generalizes two elementary identical machine scheduling problems. If the delay is zero, we obtain the preemptive problem, $P|pmtn|C_{\max}$, which can be solved efficiently by McNaughton's wrap around rule [3]. On the other hand, for large enough delay d no job will migrate in the optimal schedule. In this case we obtain the non-preemptive problem $P||C_{\max}$ which is strongly NP-hard [4]. An intriguing question is what happens for the intermediate values of d . When is the problem hard? Is there a simple rule, like McNaughton's rule, which solves the problem? Is there a good approximation algorithm?

Known Results. The two-machine non-preemptive problem, $P2||C_{\max}$, is NP-hard [4]. There is an exact pseudo-polynomial time algorithm for $Pm||C_{\max}$ [5]. Regarding approximations, there is a polynomial time approximation scheme (PTAS) for the general problem, $P||C_{\max}$ [6], that is, a family of approximation algorithms $\{A_\varepsilon\}_{\varepsilon>0}$ such that given an instance of the problem A_ε finds a solution within a factor of $(1 + \varepsilon)$ from the optimum in time polynomial in the size of the input for fixed ε .

Our problem is closely related to the problem of scheduling unit-length jobs with precedence constraints and equal communication delays, $P|prec, p_j = 1, c_{jk} = d|C_{\max}$, where a precedence relation between two jobs J_j and J_k implies that job J_k always starts after job J_j completes, and at least d time units later if the jobs are processed on different machines. If in our problem all preempted parts are integral, then we obtain an instance of the above problem with chains-like precedence constraints, $P|chains, p_j = 1, c_{jk} = d|C_{\max}$.

The version of unit communication delays, $P|prec, p_j = 1, c_{jk} = 1|C_{\max}$, is well studied. Regarding approximations, Hanen & Munier [7] presented a $(7/3 - 4/(3m))$ -approximation algorithm. Lenstra & Veltman [8] showed that there is no $(5/4 - \varepsilon)$ -approximation algorithm unless $P = NP$. Improving these bounds remains one of most challenging problems in scheduling theory [9]. Far less is known for the general version with large communication delays. Engels et al. [10] considered a version of the problem with tree-like precedence constraints and general communication delays, $P|tree, p_j = 1, c_{jk, iq}|C_{\max}$, where $c_{jk, iq}$ depends on the pair of machines (i, q) and the pair of jobs (j, k) . They showed that there is a polynomial time algorithm if all $c_{jk, ik}$ are bounded by some constant. As soon as communication delays are considered as part of the input, the problem becomes harder to solve. Afrati et al. [11] showed that the two-machine problem with tree-like precedence constraints and equal communication delays, $P2|tree, p_j = 1, c_{jk} = d|C_{\max}$, is NP-hard. Engels [12] showed that the single-machine problem with chain-like precedence constraints and just two possible delays, $P1|prec, p_j = 1, c_{jk} \in \{0, d\}|C_{\max}$, is NP-hard.

Our Results. In this paper we add a migration restriction to the classical model of scheduling on identical machines. We present hardness and approximation

results. In Section 1 we show that the sub-problem of $P|pmtn(delay = d)|C_{max}$ with delay d at most $\Delta - p_{max}$ can be solved in linear time, where $p_{max} = \max_{j=1}^n p_j$, $L = \sum_{j=1}^n p_j$, and $\Delta = \max\{p_{max}, L\}$. Further, we show that for any constant $\varepsilon > 0$ the sub-problem of $P|pmtn(delay = d)|C_{max}$ with d larger than $(1 + \varepsilon)$ times $(\Delta - p_{max})$ is NP-hard in strong sense. In Section 2 we investigate properties of the optimal schedule. We show that there always exists an optimal schedule which yields at most one preemption for the two-machine problem, $P2|pmtn(delay = d)|C_{max}$, and an optimal schedule which yields at most two migrations for the three-machine problem, $P3|pmtn(delay = d)|C_{max}$. This leads to two exact pseudo-polynomial time algorithms for the problems. We conjecture that there always exists an optimal schedule which yields at most $m - 1$ migrations for $Pm|pmtn(delay = d)|C_{max}$. Further, we show that for any instance of $P|pmtn(delay = d)|C_{max}$ there exists an optimal schedule in which no machine is idle before its completion time. In Section 3 we give an algorithm which finds near-optimal solutions for any instance of the two-machine problem. The algorithm outputs an optimal schedule if delay d is at most $(1 - 2/\log_2 n)\Delta$, and otherwise, it finds a schedule of length at most $(1 + 1/\log_2 n)$ times the optimum. The running time of the algorithm is linear in the number of jobs, n . Finally, in Section 4 we give a polynomial time approximation scheme (PTAS) for the general problem, $P|pmtn(delay = d)|C_{max}$.

As mentioned above, our model is closely related to scheduling with communication delays. Most of the results presented can be directly translated for the problem, $P|chains, p_j = 1, c_{jk} = d|C_{max}$. No results for this model with large communication delays were known before.

1 Complexity

Let $p_{max} = \max_{j=1}^n p_j$ be the maximum processing time and $L = \sum_{j=1}^n p_j/m$ be the average machine load. We define $\Delta = \max\{p_{max}, \sum p_j/m\}$. We have the following bounds

$$\Delta \leq \text{OPT} \leq \Delta + p_{max} \leq 2 \cdot \Delta, \tag{1}$$

where OPT denotes the optimum.

McNaughton's Rule. Recall that any instance with $d = 0$ can be solved by using McNaughton's wrap around rule [3]: Select the jobs one by one in some order and sequentially fill up the schedules of machines M_1, M_2, \dots, M_m within the time interval $[0, \Delta]$. If on a current machine M_i there is no enough room for the next in turn job, first complete the schedule of M_i , and only then move the rest of the job to the beginning of the schedule of the next machine M_{i+1} . The procedure outputs a schedule of length δ in $O(n)$ time. By adopting this, we can provide the following result.

Theorem 1. *The sub-problem of $P|pmtn(delay = d)|C_{max}$ with delay d at most $\Delta - p_{max}$ can be solved by using McNaughton's rule in $O(n)$ time.*

Proof. If $\Delta = p_{max}$, then $d = 0$ and McNaughton's rule gives an optimal schedule of length Δ . If $\Delta = L = \sum_{j=1}^n p_j/m$, then McNaughton's rule gives a schedule

of length L . Consider any job J_j which gets preempted and migrates from a machine M_i to machine M_{i+1} . The gap between the part of job J_j on machine M_i and the other part of J_j on machine M_{i+1} is exactly $L - p_j$, which is at least $L - p_{\max} \geq d$. Hence, the delay constraint is satisfied. The running time is linear in the number of jobs, n . \square

Hardness. Next, we provide hardness results for our problem. We reduce from the strongly NP-hard problem α -PARTITION [13]:

Instance: $k \cdot \alpha$ positive numbers $e_1, e_2, \dots, e_{\alpha \cdot k}, E \in \mathbb{Z}^+$ such that

$$\sum_{i=1}^{\alpha \cdot k} e_i = k \cdot E \text{ and } E/(\alpha + 1) < e_i < E/(\alpha - 1) \text{ for all } i = 1, \dots, \alpha \cdot k. \quad (2)$$

Question: Is there a partition of $\{1, 2, \dots, \alpha \cdot k\}$ into k disjoint subsets N_1, N_2, \dots, N_k such that

$$\sum_{i \in N_\ell} e_i = E \text{ for all } \ell = 1, \dots, k? \quad (3)$$

Theorem 2. *For any constant $\varepsilon > 0$ the sub-problem of $P|pmtn(\text{delay} = d)|C_{\max}$ with delay d at least $(1 + \varepsilon)$ times $(\Delta - p_{\max})$ is NP-hard in strong sense.*

Proof. Given an instance of α -PARTITION, we construct an instance of our problem as follows. We let the number of machines $m = \alpha \cdot k$, the number of jobs $n = m + k$, and the migration delay $d = E/(\alpha - 1)$. Next, we define m large jobs J_j ($j = 1, \dots, m$) with processing times $p_j = 2E - e_j$, and k small jobs J_j ($j = m + 1, \dots, m + k$) of equal processing length $p_{m+1} = \dots = p_{m+k} = E$. Then, the average machine load L is equal $2E$, which is at least p_{\max} . Thus, we have that $\Delta = L$.

One can see that the above construction can be completed in time polynomial in k and α . In the following we show that an instance of α -PARTITION yields the “YES”-answer if and only if the constructed instance of our problem yields a schedule of length L .

Assume that we have a schedule of length L . Since $d = E/(\alpha - 1)$ and

$$L = 2E = E + (\alpha - 1)d < E + \alpha \cdot d,$$

we can conclude that each small job migrates at most $(\alpha - 1)$ times. On the other hand, from (2) we have that $d = \frac{E}{\alpha - 1} > e_i$ ($i = 1, \dots, m$), which implies that no large job migrates. Hence, in order to have a schedule of length L , each machine should process at least one small job for some time whereas each small job can be processed by at most α machines. Since there are $m = \alpha \cdot k$ machines and k small jobs, each small job is processed by exactly α machines and no machine processes two different small jobs. This gives a feasible solution for the given instance of α -PARTITION. Similarly, one can also see that any feasible solution for an instance of α -PARTITION leads to a schedule of length L for the constructed instance of our problem.

In the instance we have that

$$d = \frac{E}{\alpha - 1} = \frac{\alpha + 1}{\alpha - 1} \cdot \frac{E}{\alpha + 1} < \frac{\alpha + 1}{\alpha - 1} \cdot e_{\min} = \left(1 + \frac{2}{\alpha - 1}\right) (\Delta - p_{\max}),$$

where $e_{\min} = \min_{j=1}^m e_j$, $\Delta = L = 2E$ and $p_{\max} = \max_{j=1}^{m+k} p_j = 2E - e_{\min}$. So, for any fixed $\varepsilon > 0$ one can find a fixed $\alpha \in \mathbb{Z}^+$ such that $\frac{2}{\alpha - 1} \leq \varepsilon$. Hence, the sub-problem with $d \geq (1 + \varepsilon)(\Delta - p_{\max})$ consists the above instance and, thereby, is NP-hard in strong sense. \square

2 Properties of the Optimal Schedule

Theorem 3. *For any instance of the two-machine problem $P2|pmtn(\text{delay} = d)|C_{\max}$ there exists an optimal schedule with at most one migration.*

Proof. Given an optimal schedule of length OPT, we construct a new optimal schedule as follows. Let L_i ($i = 1, 2$) be the total load of the jobs which completely run on machine M_i in the given optimal schedule without migrations. We reschedule the jobs of L_1 within interval $[0, L_1]$ on M_1 and the jobs of L_2 within interval $[OPT - L_2, OPT]$ on M_2 , respectively. No preemptions and migrations for these jobs are made, see Fig 1a.

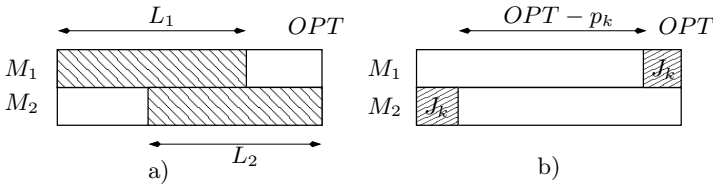


Fig. 1. Rescheduling on two machines

Next we reschedule the jobs which migrate in the given optimal schedule by using McNaughton’s wrap around rule within still available intervals $[L_1, OPT]$ and $[0, OPT - L_2]$. This completes the construction. The length of the schedule is OPT. There is at most one job which gets preempted and migrates from M_2 to M_1 .

Assume w.l.o.g. that there is one job J_k which migrates in the constructed schedule, see Fig 1b. This job J_k is scheduled in the end of $[L_1, OPT]$ on machine M_1 , and in the beginning of $[0, OPT - L_2]$ on M_2 . The time gap between the two parts of J_k is exactly $OPT - p_k$. Since J_k also migrates in OPT, we have that $p_k + d$ is at most OPT. Hence, the delay constraint is satisfied. The constructed schedule is optimal. \square

There exist instances for which the number of preemptions and the number of migrations must differ in the optimal schedule. Consider the following example. There are three *large* jobs J_1, J_2, J_3 and one *small* job J_4 . such that $\Delta = (p_1 +$

$p_2 + p_3 + p_4)/3 = 2d + p_4$ and $p_1 = p_2 = p_3$. Then, in order to get a schedule of length Δ , the small job should migrate two times, and one large job should be preempted one time. Hence, there are two migrations and three preemptions.

Theorem 4. *For any instance of the $P3|pmtn, \text{delay} = d|C_{\max}$ there exists an optimal schedule with at most two migrations.*

Proof. Given an optimal schedule of length OPT we construct a new optimal schedule in two steps. We first construct a pseudo-schedule of length OPT which may be infeasible, and then turn this into a feasible schedule.

Let L_i ($i = 1, 2, 3$) be the total load of the jobs which completely run on machine M_i in the given optimal schedule without migrations. Assume w.l.o.g that $L_1 \leq L_2 \leq L_3$. We place the jobs of L_i ($i = 1, 2, 3$) within $[0, L_i]$ on M_i , respectively. No preemptions and migrations for these jobs are made.

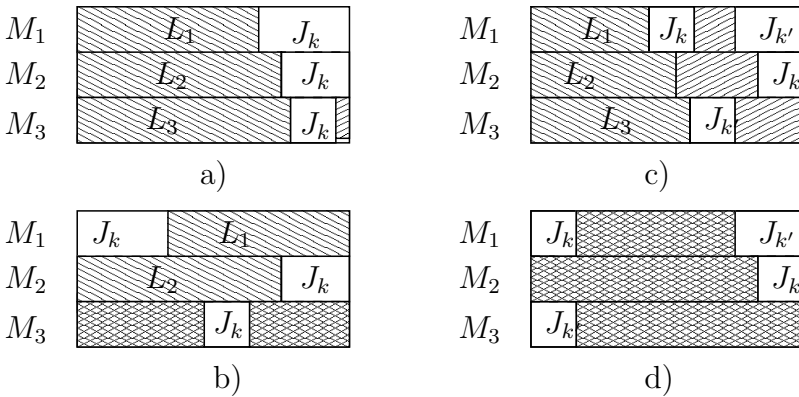


Fig. 2. Rescheduling on three machines

Next we place the jobs which migrate in the given optimal schedule. We order these jobs by non-decreasing lengths. Then we select the jobs one by one in this order and use McNaughton’s wrap-around rule. We first place the jobs on machine M_2 within interval $[L_2, \text{OPT}]$, then on machine M_1 within $[L_1, \text{OPT}]$, and finally on M_3 within $[L_3, \text{OPT}]$. This gives a pseudo-schedule of length OPT .

If there is at most one migration in the pseudo-schedule, then the proof follow directly from the proof of Lemma 3. In the following we assume that there are exactly two migrations, and modify the schedule so as to satisfy the delay constraints.

Assume that there is a job J_k which is processed on all three machines, see Fig. 2a. Due to the rescheduling procedure, we can conclude that the processing time p_k of J_k is larger than $(\text{OPT} - L_2) + (\text{OPT} - L_1)$. Since $L_1 \leq L_2 \leq L_3$, job J_k must migrate at least two times in the given optimal schedule. The delay constraint implies that $p_k + 2d$ is at most OPT . Hence, we can modify the schedule as shown in Fig 2b.

Now assume that there is one job J_k which is processed on machines M_2 and M_1 , and one job $J_{k'}$ which is processed on machines M_2 and M_3 , see Fig. 2c. Since jobs J_k and $J_{k'}$ both migrate in OPT, the delay constraint implies that $p_k + d \leq \text{OPT}$ and $p_{k'} + d \leq \text{OPT}$. Hence, we can modify the schedule as shown in Fig 2d. \square

Conjecture 1. For any instance of $Pm|pmtn, \text{delay} = d|C_{\max}$ there exists an optimal schedule with at most $m - 1$ migrations.

One can see that a simple rule, like McNaughton’s rule, cannot be used to find the optimal schedule of the preempted jobs, even if almost all information on the optimal schedule is given. Notice also that, unlike the problem with preemptions but delays, there might be more than two preempted jobs on a machine, and an optimal schedule might have to preempt the same job more than once. In fact, even more complicated structures can be constructed.

Theorem 5. *For any instance of the general problem $P|pmtn(\text{delay} = d)|C_{\max}$ there exists an optimal schedule in which no machine is idle before its completion time.*

Proof. Let σ be a schedule with minimizes the following three objectives in the given order: the makespan, the sum of all completion times, and minus the sum over all jobs of the mean completion times, where the mean completion time of a job is the average time at which it is processed. Assume w.l.o.g. that machine M_1 is idle in σ at some time t , i.e. no job is processed on M_1 in the interval $]t - \varepsilon, t[$ for some $\varepsilon > 0$, and M_1 does process some job at a later moment. We assume w.l.o.g that $d \geq \varepsilon$. Let job J_j be the first job that is processed after time t on M_1 . If J_j is entirely processed on M_1 or is not processed on any machine before time t , then we can reduce its completion time by ε without increasing the makespan or changing the schedule of any other job. If job J_j is processed before time $t - \varepsilon$, then we remove the last ε units of J_j that were processed before time $t - \varepsilon$, and process it on M_1 between $t - \varepsilon$ and t . We have increased the average completion time of J_j by a small amount without increasing the completion time of any job. Hence, σ is not optimal for the given objectives. This gives a contradiction. \square

3 Two Machines: A Near-Optimal Schedule in Linear Time

We show that for the two machine problem, $P2|pmtn(\text{delay} = d)|C_{\max}$, a near optimal solution can be found in $O(n)$ time. First we introduce some notation. We assume that jobs J_1, J_2, \dots, J_n ($n \geq 2$) are numbered in non-increasing order of their processing times $p_1 \geq \dots \geq p_n$. So, we have $\Delta = \min\{p_1, \sum_{j=1}^n p_j/2\}$ as a lower bound on the optimum OPT. We use I_k ($k = 1, \dots, n$) to denote the set of k largest jobs $\{J_1, J_2, \dots, J_k\}$. Taking the jobs of I_k as an instance of the

non-preemptive problem $P2||C_{\max}$, we use C_k to denote its optimum, and S_k to denote its optimal (non-preemptive) schedule. Then, we have that

$$C_1 \leq C_2 \leq \dots \leq C_n.$$

Lemma 1. *If $C_n = \Delta$, then $\text{OPT} = \Delta$. In the other case, let $k^* \doteq \min\{k \mid k \in \{1, 2, \dots, n\} \text{ and } C_k > \Delta\}$ and let $\gamma \doteq \max\{\Delta, p_{k^*} + d\}$. Then,*

$$\text{OPT} = \min\{\gamma, C_{k^*}\}. \tag{4}$$

Proof. The first is immediate since S_n is a feasible schedule and Δ is a lower bound on OPT . For the other case let us show that

$$C_{k^*-1} \leq \text{OPT} \leq C_{k^*}.$$

The lower bound follows from $C_{k^*-1} \leq \Delta \leq \text{OPT}$. To justify the upper bound, it suffices to take the schedule S_{k^*} and complete it to a feasible schedule for the original problem with length equal to C_{k^*} — by just assigning the remaining (“small”) jobs to the less loaded machine. — The optimal schedule should clearly have length not greater than C_{k^*} .

It is clear that OPT cannot be strictly less than C_{k^*} , if no job $J_i \in I_{k^*}$ migrates. On the other hand, if some of them migrate, OPT cannot be less than $\gamma \doteq \max\{\Delta, p_{k^*} + d\}$. Thus, we have the following lower bound on the optimum:

$$\text{OPT} \geq \min\{\gamma, C_{k^*}\}. \tag{5}$$

Let us show that, in fact, (5) holds as an equality. Indeed, if “min” in the right part of (5) is attained at C_{k^*} , then we can easily achieve it via constructing a schedule of length C_{k^*} (without preemption), as shown above. Alternatively, if $\gamma < C_{k^*}$, we construct a schedule of length γ as follows:

- take the schedule S_{k^*-1} (it has length $C_{k^*-1} \leq \Delta$ and no preemptions);
- put the job J_{k^*} to an arbitrary machine (let it be M_1) — the schedule length becomes at least C_{k^*} ;
- divide the job J_{k^*} into two parts, making the load of machine M_1 to be equal to γ ; the (positive) remainder of the job migrates to machine M_2 ;
- put the remaining jobs (J_{k^*+1}, \dots, J_n) on machine M_2 .

Clearly, the resulting schedule is feasible and its length is equal to γ . □

Theorem 6. *There exists an algorithm which for any instance of the two-machine problem $P2|pmtn(\text{delay} = d)|C_{\max}$ with n jobs (a) solves the problem to the optimum if delay d is at most $\left(1 - \frac{2}{\log_2 n}\right) \Delta$, and (b) otherwise, it provides either an optimal schedule or an approximate non-preemptive schedule S of length at most*

$$C_{\max}(S) \leq \left(1 + \frac{1}{\log_2 n}\right) \text{OPT}. \tag{6}$$

The running time of the algorithm is linear in the number of jobs, n .

Proof. Put $k' := \lceil \log_2 n \rceil$. Then

$$p_{k'} \leq \frac{2\Delta}{k'} \leq \frac{2}{\log_2 n} \Delta. \tag{7}$$

Observe that we can extract the set of jobs $I_{k'}$ (i.e., the set of the largest k' jobs) from the whole set $\{J_1, J_2, \dots, J_n\}$ in $O(n)$ time by means of the standard technique [14], after which its items can be renumbered in non-increasing order of processing times ($p_1 \geq \dots \geq p_{k'}$) in $O(\log n \log \log n) = o(n)$ time. After this we can compute all optimums $\{C_1, \dots, C_{k'}\}$ in $O(2^{k'}) = O(n)$ time by the direct enumeration of all subsets of the set $\{1, 2, \dots, k'\}$.

Next we compare $C_{k'}$ and Δ . If $C_{k'} > \Delta$, we can conclude that $k^* \leq k'$, and that k^* can be found in $O(\log n)$ comparisons of values $\{C_i \mid i = 1, \dots, k'\}$ with Δ . Next we construct the optimal schedule (of length $\text{OPT} = \min\{\gamma, C_{k^*}\}$), using the schedules S_{k^*-1} and S_{k^*} as described in the proof of the previous lemma. If $C_{k'} \leq \Delta$ (and hence, $k^* > k'$), then in the case $d + p_{k'} \leq \Delta$ we have $\text{OPT} = \Delta$, and the optimal schedule can be obtained by the completion of the schedule $S_{k'}$ up to a whole schedule using the linear-time McNaughton’s algorithm. Alternatively, if $d + p_{k'} > \Delta$, and therefore (from (7)),

$$d > \Delta - p_{k'} \geq \left(1 - \frac{2}{\log_2 n}\right) \Delta,$$

we take the schedule $S_{k'}$ and complete it to a whole non-preemptive schedule for the original instance using the simple list-scheduling algorithm (with the remaining jobs sequenced in an arbitrary order). It can be seen that the length of schedule S meets an upper bound $C_{\max}(S) \leq \Delta + \frac{1}{2} p_k$ for some $k > k'$, which implies

$$C_{\max}(S) \leq \Delta + \frac{1}{2} p_{k'}. \tag{8}$$

The desired bound (6) follows from (8) and (7). □

4 A Polynomial Time Approximation Scheme

Here we give an outline of a polynomial time approximation scheme (PTAS), that is, a family of approximation algorithms $\{A_\varepsilon\}_{\varepsilon>0}$ such that given an instance of the problem A_ε finds a solution within a factor of $(1 + \varepsilon)$ from the optimum in time polynomial in the size of the input for fixed ε . Our approach to approximation is to perform several transformations that simplify the input problem without dramatically increasing the objective value, so that the final result is amenable to a fast enumeration solution. As before, $p_{\max} = \max_{j=1}^n p_j$, $L = \sum_{j=1}^n p_j/m$, $\Delta = \max\{p_{\max}, \sum p_j/m\}$, and $\Delta \leq \text{OPT} \leq 2 \cdot \Delta$. We assume w.l.o.g. that $1/\varepsilon \geq 4$ is integral.

Cleaning Up. Our first transformation cleans up the instance. The idea is to separate the set of jobs – we say that a job J_j is *large* if $p_j > \varepsilon\Delta$, *small* otherwise. We drop all small jobs from the instance, and add them later to the schedule.

From now on we are interested in obtaining a near-optimal schedule for the large jobs. Since $\sum_{j=1}^n p_j \leq m \cdot \Delta$, we can conclude that there are at most $2m/\varepsilon$ large jobs. A job which migrates k times is split into $k + 1$ parts which we call *operations*. Our next transformation gives a lower bound on the value of d , as well as an upper bound on the maximum number of operations (migrations) of a large job.

Lemma 2. *With $1 + 2\varepsilon$ loss, we can assume that the value of d is at least $2\varepsilon\Delta$, and hence, each large job consists of at most $2/\varepsilon$ operations.*

Proof. If $d \leq 2\varepsilon\Delta$, we can construct a schedule as follows. We first use McNaughton’s wrap-around rule on the set of large jobs, ignoring all migration constraints. This gives a pseudo-schedule of length Δ in which at most $m - 1$ large jobs migrate. Next, we use the structure of the schedule. We move the parts of the jobs which run in the end of the schedule by d so that to satisfy all migration constraints. This completes a feasible schedule of length $\Delta + d \leq (1 + 2\varepsilon)\Delta$, which is at most $(1 + 2\varepsilon)\text{OPT}$.

If $d \geq 2\varepsilon\Delta$, then each job can migrate at most OPT/d times. From $\text{OPT} \leq 2\Delta$, each job can consist of at most $1/\varepsilon + 1 \leq 2/\varepsilon$ operations. \square

Rounding. By applying our main transformation we round all values in our problem to some discrete points, that is very useful for further enumeration. For simplicity, we write O_j^h to denote the h th operation of job J_j , whereas we write p_j^h and S_j^h to denote the processing time and starting time of operation O_j^h . We can bound operation sizes as follows.

Lemma 3. *With $1 + \varepsilon$ loss, we can assume that the processing time of each operation p_j^h is at least $\varepsilon^4\Delta/4$.*

Proof. Given an optimal schedule of length OPT we scale it by $1 + \varepsilon$. The time reserved for each operation O_j^h is then $(1 + \varepsilon)p_j^h$. Next, we move the start time of O_j^h so far that this creates an idle time gap of size εp_j^h before its start time.

For each large job J_j , there are at most $2/\varepsilon - 1$ operations. Hence, there is at least one big operation of J_j with processing time $p_j/(2/\varepsilon) = \varepsilon p_j/2$. We can conclude that there is an idle time gap of size at least $\varepsilon^2 p_j/2 \geq \varepsilon^3 \Delta/2$. Thus, we can take the operations of job J_j with sizes at most $\varepsilon^4 \Delta/4$, which sum up to at most $\varepsilon^4(2/\varepsilon)\Delta/4 = \varepsilon^3 \Delta/2$ total size, and then put them together into the idle gap.

We go over all jobs performing the above procedure. In the end, there is no operation with size less than $\varepsilon^4 \Delta/4$. We increased the schedule length by εOPT . \square

Next we round all values to $O(1/\varepsilon^5)$ possible discrete points.

Lemma 4. *With $1 + 2\varepsilon$ loss, we can round up all values so that the delay $d \in \{4/\varepsilon^4, 4/\varepsilon^4 + 1, \dots, 16/\varepsilon^5\}$, all operation sizes $p_j^h \in \{16/\varepsilon, 16/\varepsilon + 1, \dots, 8/\varepsilon^5\}$, and all starting times $S_j^h \in \{0, 1, \dots, 16/\varepsilon^5\}$.*

Proof. Given an optimal schedule of length OPT we scale it by $1 + \varepsilon$. The time reserved for each operation O_j^h is then $(1 + \varepsilon)p_j^h$. Next, we move the start time of O_j^h so far that this creates an idle time gap of size εp_j^h before its start time. Since

$p_j^h \geq \varepsilon^4 \Delta/4$, this gap is at least $2\varepsilon^5 \Delta/4$. Next, we use one part of this gap with size $\varepsilon^5 \Delta/4$ to round p_j^h , and the other part with the same size to round S_j^h so that their values turn to some integer multiples of $\varepsilon^5 \Delta/4$. Similar, we round the value of d to an integer multiple of $\varepsilon^5 \Delta/4$. The procedure increases the schedule length by $2\varepsilon \text{OPT}$.

Further, we divide all values by $\varepsilon^5 \Delta/4$ where $\varepsilon \in (0, 1/4]$. Since $d \in [\varepsilon \Delta, 2(1+4\varepsilon)\Delta]$, each $p_j^h \in [\varepsilon^4 \Delta/4, (1+4\varepsilon)\Delta]$, and each $S_j \in [0, 2(1+4\varepsilon)\Delta]$, we obtain the claimed bounds. \square

Enumeration. One can see that Lemmas 2 and 4 shape a schedule for the set of large jobs: (i) there are $O(1)$ unit time slots (ii) each large job J_j forms $O(1)$ operations O_j^h (iii) each operation O_j^h can eventually fill $O(1)$ time slots but it completely runs on one machine. This information is enough to find a near-optimal schedule by a brute-force enumeration. Omitting technical details, we can state the following result.

Lemma 5. *In $m^{O(1)}$ time we can find a feasible schedule for the large jobs which length is at most $(1 + O(\varepsilon))\text{OPT}$.*

Though we aim at the existence of a PTAS, we can provide some nice enumeration techniques which replace $m^{O(1)}$ by a clean bound.

Adding the Small Jobs. Now we complete the obtained schedule of the large jobs by adding the small jobs in a greedy manner. We select the small jobs one by one and preemptively schedule them on the machines as soon as possible and without migrations. We use preemptions each time when the current machine is busy processing a large job. There is no small job migrates in the schedule, and the schedule length is at most $(1 + O(\varepsilon))\text{OPT} + \varepsilon \Delta$, which is $(1 + O(\varepsilon))\text{OPT}$. In overall, we obtain the following final result.

Theorem 7. *For any instance of $P|pmtn(\text{delay} = d)|C_{\max}$ and any positive accuracy ε , one can find a schedule of length at most $(1 + \varepsilon)\text{OPT}$ in $O(m^{f(\varepsilon)} + n)$ time, where $f(\cdot)$ is some function independent of the number of jobs n and the number of machines m .*

Proof. In the algorithm we need to separate the jobs into small and large ones, round all values, and perform the enumeration procedure. The first two steps can be completed in $O(n)$ time. The enumeration procedure can be run in $m^{O(1)}$ time. \square

References

1. Graham, R.L., Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G.: Optimization and approximation in deterministic scheduling: A survey. *Annals of Discrete Mathematics* (1979) 287–326
2. Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G., Shmoys, D.B.: Sequencing and scheduling: Algorithms and complexity. In: *Logistics of Production and Inventory. Volume 4 of Handbooks in Operation Research and Management Science*. North-Holland, Amsterdam (1993) 445–522

3. McNaughton, R.: Scheduling with deadlines and loss functions. *Management Science* **6** (1959)
4. Karp, R.M.: Recucibility among combinatorial problems. In: *Complexity of Computer Computations*. Plenum Press, New York (1972)
5. Rothkopf, M.H.: Scheduling independent tasks on parallel processors. *Management Science* **12** (1966) 347–447
6. Hochbaum, D.S., Shmoys, D.: A polynomial approximation scheme for scheduling on uniform processors: using the dual approximation approach. *SIAM Journal on Computing* **17** (1988) 539–551
7. Hanen, C., Munier, A.: An approximation algorithm for scheduling dependent tasks on m processors with small communication delays. *Discrete Applied Mathematics* **108** (2001) 239–257
8. Hoogeveen, J.A., Lenstra, J.K., Veltman, B.: Three, four, five, six, or the complexity of scheduling with communication delays. *Operations Research Letters* **16** (1994) 129–137
9. Schuurman, P., Woeginger, G.J.: Polynomial time approximation algorithms for machine scheduling: Ten open problems. *Journal of Scheduling* **2** (1999) 203–213
10. Engels, D.W., Feldman, J., Karger, D.R., Ruhl, M.: Parallel processor scheduling with delay constraints. In: *SODA*. (2001) 577–585
11. Afrati, F.N., Bampis, E., Finta, L., Milis, I.: Scheduling trees with large communication delays on two identical processors. In: *Euro-Par*. (2000) 288–2295
12. Engels, D.W.: Scheduling for hardware-software partitioning in embedded system design. PhD thesis, Massachusetts Institute of Technology, Cambridge, USA (2000)
13. Garey, M.R., Johnson, D.S.: *Computers and intractability: A guide to the theory of NP-completeness*. Freeman, San Francisco, CA (1979)
14. Blum, M., Floyd, R., Pratt, V., Rivest, R., Tarjan, R.: Time bounds for selection. *J. Comp. System Sci.* (1973)

Fairness-Free Periodic Scheduling with Vacations

Jiří Sgall^{1,*}, Hadas Shachnai², and Tami Tamir³

¹ Mathematical Institute, AS CR, Žitná 25, CZ-11567 Praha 1,
Czech Republic, and Dept. of Applied Mathematics,
Faculty of Mathematics and Physics, Charles University, Praha
`sgall@math.cas.cz`

² Computer Science Department, The Technion, Haifa 32000, Israel
`hadas@cs.technion.ac.il`

³ School of Computer science, The Interdisciplinary Center, Herzliya, Israel
`tami@idc.ac.il`

Abstract. We consider a problem of *repeatedly* scheduling n jobs on m parallel machines. Each job is associated with a profit, gained each time the job is completed, and the goal is to maximize the average profit per time unit. Once the processing of a job is completed, it goes on *vacation* and returns to the system, ready to be processed again, only after its vacation is over. This problem has many applications, in production planning, machine maintenance, media-on-demand and databases query processing, among others.

We show that the problem is NP-hard already for jobs with unit processing times and unit profits, and develop approximation algorithms, as well as optimal algorithms for certain subclasses of instances. In particular, we show that a preemptive greedy algorithm achieves a ratio of 2 to the optimal for instances with arbitrary processing times and arbitrary profits. For the special case of unit processing times, we present a 1.67-approximation algorithm for instances with arbitrary profits, and a 1.39-approximation algorithm for instances where all jobs have the same (unit) profits. For the latter case, we also show that when the load generated by an instance is sufficiently large (in terms of n and m), any algorithm that uses no intended idle times yields an optimal schedule.

1 Introduction

We consider a scheduling problem in which jobs need to be scheduled repeatedly. The input is a set of m identical parallel machines and a set of n jobs, $J = \{1, \dots, n\}$, that need to be scheduled on the machines. All the jobs are ready at time $t = 0$. Each job j , is associated with a processing time $p_j \geq 1$, and a window $a_j \geq p_j$; once the processing of j is completed, it goes on *vacation* and returns after $a_j - p_j$ time units, ready to be processed again. Thus, a feasible schedule is one where each machine processes at most one job at any time, and there is a gap of at least $a_j - p_j$ time units between two consecutive executions of job j . The jobs are sequential, i.e., no job can be scheduled on more than one machine at the

* Supported by research project MSM0021620838 of MŠMT ČR.

same time. There is also a profit, b_j , associated with each execution of job j ; the goal is to find a feasible schedule that maximizes the average profit per time unit. We call this problem *scheduling with vacations (SWV)*. Our problem arises in many practical scenarios where tasks need to be accomplished periodically, but due to setup times or maintenance requirements there must be some gap between two consecutive executions of a task. The following are two of the applications that motivate our study.

Surveillance Camera Scheduling: Consider a system of robots carrying surveillance cameras, which patrol an area periodically. Each robot has a predefined path that it needs to patrol, while recording all the events along this path. Upon completion of each patrol, the robot returns to the controller, where the recorded data is downloaded/processed, and the robot is prepared for its next tour. Each patrol j is associated with a profit b_j , gained once the corresponding robot completes the tour. The controller can handle at most m robots simultaneously, and it takes p_j time units to complete the processing of robot j . The time it takes robot j to traverse its path is $t_j \geq 1$. The goal of the controller is to process the robots in a way that maximizes the profit gained from all robots, throughout the operation of the system. This yields an instance of SWV, where job j has a processing time p_j , and the window $a_j = p_j + t_j$.

Commercial Broadcast: In a broadcasting system which transmits data, e.g., commercials on a running banner, there is a profit associated with the transmission of each commercial. This profit is gained only if some predefined period of time elapsed since the previous transmission. The goal is to broadcast the commercials in a way that maximizes the overall profit of the system. Thus, we get an instance of SWV, where the window of each job corresponds to the time interval between consecutive transmissions of each commercial.

1.1 Our Results and Related Work

The problem of scheduling with vacations belongs to the class of *periodic scheduling* problems, where each job may be scheduled infinite number of times. Periodic scheduling is a well-studied problem. Problems of this type arise in many areas, including production planning, machine maintenance, telecommunication systems, media-on-demand, image and speech processing, robot control/navigation systems, and database query processing. In the paper [12], which introduced the periodic scheduling problem, the goal is to schedule each of the jobs such that the average gap between two executions of job j is a_j , using the minimum number of machines. Another early example is the *chairman assignment* problem [13], in which the number of executions of job j in any prefix of the schedule of length t must be at least $\lfloor t/a_j \rfloor$ and at most $\lceil t/a_j \rceil$. For both problems the *earliest deadline first* algorithm was shown to be optimal.

Our problem is different from previously studied variants of periodic scheduling in two major aspects. First, our goal is to maximize the total profit of the server. This may result in lack of fairness. Indeed, in a schedule which maximizes the profit, some of jobs may be waiting forever, while other (more profitable) jobs

are repeatedly processed as soon as they return from vacation. In other works (e.g., [12,7,8]), the performance of a periodic schedule is measured by some fairness criterion, or (e.g. [2,9]) the two objectives are combined; that is, each job comprises a mandatory and an optional part, with which a non-decreasing reward function is associated. In other variants of the periodic scheduling problem (see, e.g., in [1]), the processing of each job incurs a service cost, depending on the time that elapsed since the last service of this job, and the goal is to minimize the cost of the schedule, however, the cost increases with job delays – implying that an optimal solution requires some fairness.

The other major difference from well-studied variants of periodic scheduling is that in our problem, due to the vacation requirement, jobs cannot be processed too often. In the *windows scheduling* problem [4,5], the parameter associated with each job gives the *maximal* gap between any two executions of job j , however, it is feasible to schedule a job more often. In the *periodic machine scheduling problem (PMSP)* [3], this parameter specifies the *exact* gap between any two executions of j . To the best of our knowledge, there is no earlier work in which only the *minimal* gap between any two executions is given.

We first show that SWV is NP-hard already for jobs with unit length and unit profits. Since the total number of different configurations of the system is finite, an optimal solution can be found (inefficiently) by applying the known *buffer scheme* designed for the windows scheduling problem [6]. We present several approximation algorithms for various subclasses of instances. A simple greedy algorithm yields a 2-approximation for preemptive scheduling of jobs with arbitrary processing times and arbitrary profits. For the special case of unit processing times, we present a 1.67-approximation algorithm for instances with arbitrary profits, and a 1.39-approximation algorithm for instances with the same (unit) profits. For the latter case, we also show that when the load generated by an instance is sufficiently large (in terms of n and m), any algorithm that uses no intended idle times yields an optimal schedule.

Our approximation algorithms for unit length jobs apply a transformation of the instance to an *aligned instance*, where all window sizes satisfy certain properties. Our approximation technique, based on finding an optimal schedule for the resulting aligned instance, builds on a technique of [5] for the windows scheduling problem. To obtain better approximation ratio, we extend this technique: Our algorithm for unit processing times and unit profits applies the *best align* technique, which finds an aligned instance that yields the best approximation ratio for the original instance.

2 Preliminaries

2.1 Definitions and Notation

Denote by a_j the *scheduling window* (for short, *window*) of job j . The *load* incurred by job j is $\ell(j) = p_j/a_j$. The best service job j can receive is to be processed whenever it returns from vacation (i.e., no waiting time). Thus, the load of job j represents the average processing requirement of j per time unit.

The total load of the input is $L(J) = \sum_{j \in J} \ell(j)$. In the special case of unit processing times (i.e., for all j , $p_j = 1$), we get that $\ell(j) = 1/a_j$, and $L(J) = \sum_{j \in J} 1/a_j$. Each job is associated with an additional parameter, b_j , the profit gained from each execution of job j .

We say that a job j is *heavier* than j' if $b_j/p_j > b_{j'}/p_{j'}$, or $b_j/p_j = b_{j'}/p_{j'}$ and $j > j'$. A heavier job achieves larger profit per unit time of its execution; the second part of the definition only breaks ties consistently for the whole instance. Let $J_{\geq j}$ denote the set of jobs at least as heavy as j .

To define the profit of a schedule formally, let $\text{compl}_T^S(j)$ denote the number of times job j is completed in the first T time units in schedule S . The profit of schedule S is defined as

$$\text{profit}(S) = \liminf_{T \rightarrow \infty} \frac{1}{T} \sum_{j=1}^n \text{compl}_T^S(j) b_j.$$

A schedule is periodic with period p if for any time T , the jobs scheduled at time T are the same as the jobs scheduled at time $T + p$. It is easy to see that there always exists a periodic optimal schedule: Define a configuration at time T to consist of the information of how much of each job is currently executed, or how much time has elapsed since its last completion. An optimal schedule can always be chosen so that for two points of time T and T' with the same configurations, the remainder of the schedule is the same. Since the number of configurations is finite, such a schedule is periodic.

In the remainder of paper we consider only periodic schedules. For a schedule S with period p , in the definition of profit, \liminf can be replaced by \lim which always exists and it is actually equal to the value of the remaining expression for $T = p$. Similarly, it is meaningful to speak about the load (relative frequency) of a job j in S ; we denote it by $\ell^S(j)$. Finally, $L^S(J') = \sum_{j \in J'} \ell^S(j)$ denotes the load of a set of jobs $J' \subseteq J$ in a schedule S .

The next lemma is used to compare our schedules to an optimal schedule S^* .

Lemma 2.1. *Let S^* be an arbitrary schedule for an instance J and let $R \geq 1$.*

- (i) *Suppose that for a schedule S for J , and for any job j , $L^S(J_{\geq j}) \geq L^{S^*}(J_{\geq j})/R$. Then $\text{profit}(S) \geq \text{profit}(S^*)/R$.*
- (ii) *Suppose that for a distribution of schedules \mathcal{S} for J , and for any job j , $\mathbf{Exp}_{S \in \mathcal{S}}[L^S(J_{\geq j})] \geq L^{S^*}(J_{\geq j})/R$. Then one of the schedules $S \in \mathcal{S}$ satisfies $\text{profit}(S) \geq \text{profit}(S^*)/R$.*

Proof. The lemma follows from the fact that the profit of any schedule S can be computed as $\int_{\beta=0}^{\infty} L^S(J_{\geq \beta}) d\beta$, where $J_{\geq \beta} = \{j \in J \mid b_j \geq \beta\}$. ■

2.2 Hardness Results

The hardness of *SWV* with a single machine, unit profits, and unit processing times is shown by a reduction from the *periodic machine scheduling problem (PMSP)*, which is known to be NP-hard (see in [3]). In PMSP, there is a set of

n machines; the j -th machine requires maintenance every a_j time units, where $\sum_{j=1}^n 1/a_j \leq 1$. The maintenance of any machine takes one time unit, and at any time only one machine can be maintained. It is assumed that the first maintenance of machine j can be done in any of the first a_j time slots. The goal is to find a maintenance schedule that satisfies *exactly* all the requirements, that is, for all j , machine j is maintained exactly once in any window of a_j time slots.

Theorem 2.2. *SWV is NP-hard, even with a single machine, unit profits and unit processing times.*

Proof. Given an instance of PMSP with n machines in which the j -th machine requires maintenance every a_j time units, construct an instance of SWV with unit profits, unit processing times, and scheduling windows a_j for jobs $j = 1, \dots, n$. Since $\sum_{j=1}^n 1/a_j \leq 1$, there is a feasible schedule for PMSP if and only if the average profit per time-unit from job j is exactly $1/a_j$, that is, there is a schedule of the SWV instance with average profit $\sum_{j=1}^n 1/a_j$. ■

Remark 2.3. When $L(J) \geq 1$, the problem is still NP-hard for unit profits and unit processing times. The proof is similar to the hardness proof for the windows scheduling problem given in [5]. An instance with $L(J) < 1$ and unit profits can be extended into one with $L(J') = 1$ in a way that does not affect the schedule of the original jobs. Let $A = LCM(a_1, a_2, \dots, a_n)$, that is, each of the original windows divides A . Add to J dummy jobs, each having window A , such that the total load is 1. Then a schedule of the new instance with average profit 1 induces a schedule of the original instance with average profit $\sum_{j=1}^n 1/a_j$, and vice versa. For arbitrary profits, a single dummy job with $b_j = \varepsilon$ and $a_j = 1$ is sufficient.

2.3 An Optimal Super-Polynomial Algorithm

The general buffer scheme [6] is a tool designed for solving optimally the windows scheduling problem. We explain below how it can be adjusted to solve optimally the problem of scheduling with vacations. We describe the adjustment for unit-length jobs and arbitrary profits. It can be extended to handle jobs with arbitrary lengths similar to the extension for windows scheduling shown in [6]. The buffer scheme is based on representing the system as a nondeterministic finite state machine in which any directed cycle corresponds to a valid periodic schedule and vice-versa. Let $a^* = \max_j \{a_j\}$. The state of a job is represented using a set of buffers, $B_0, B_2, \dots, B_{a^*-1}$. Each job is stored in some buffer. A job is stored in B_i when i time slots remain till the end of its vacation. Initially, all jobs are ready to be processed so they are all stored in B_0 . In each iteration, the scheme schedules at most m jobs from B_0 and updates the content of the buffers:

- For all $i > 0$, all jobs stored in B_i are moved to B_{i-1} (note that none of them is scheduled).
- Each scheduled job j is moved from B_0 to B_{a_j-1} (this ensures that at least $a_j - 1$ time slots will elapse before j is scheduled again).

Note that the total number of buffer configurations is at most $\prod_j a_j$. In the state machine, each configuration is represented by a vertex, and there is an edge connecting configuration f to configuration g if it is possible to move from the state corresponding to f to the state corresponding to g in a single time slot. Each edge $e = (f, g)$ has a weight b_e whose value is the profit gained by moving from f to g , that is, the total profit of the jobs selected for execution. A periodic schedule with cycle t corresponds to a cycle of length t in the state machine. The profit of this schedule is given by the average edge weight along this cycle. Thus, an optimal schedule corresponds to a cycle with maximal mean weight. Such a cycle can be found by using the classic *Max Cycle Mean* algorithm of Karp [10]. The running time is polynomial in the graph size, which is polynomial in $\prod_j a_j$. Clearly, this algorithm is applicable only for small instances, however, since in general the problem is NP-hard, we cannot expect efficient optimal solutions.

3 A 2-Approximation Algorithm

We describe below an algorithm for preemptive scheduling of arbitrary-length jobs. Note that in the preemptive scheduling model, the execution of a preempted job can be resumed at any time. The vacation-constraint refers only to the gaps between *different* executions of a job. Recall that m is the number of machines.

Algorithm GREEDY

At any time t , schedule for one time unit the m heaviest available jobs.

It is easy to see that GREEDY always generates a periodic schedule, as the possible number of different states is finite.

Theorem 3.1. *GREEDY is a 2-approximation algorithm.*

Proof. Let S be the schedule produced by greedy and $R = 2$. Order the jobs from the heaviest one, and denote them j_1, j_2, \dots, j_n in this order.

We prove that for every i ,

$$L^S(J_{\geq j_i}) \geq \min\{m/2, L(J_{\geq j_i})/2\}.$$

Since the optimal schedule S^* satisfies $L^{S^*}(J') \leq \min\{m, L(J')\}$ for any $J' \subseteq J$, the theorem then follows from Lemma 2.1.

To prove the claim, we distinguish between two cases.

First, assume that for some i , $\ell^S(j_i) \leq \ell(j_i)/2$. If two consecutive completion times of the job are $a_{j_i} + T$ time units apart then between the two completion times there are at least T time slots in which j_i was available but not scheduled; by the definition of the algorithm, heavier jobs were scheduled in these time slots on all machines. Consequently, the case assumption implies that in half of the slots of the schedule jobs heavier than i were executed. Thus, $L^S(J_{\geq j_i}) \geq m/2$, and the same follows also for any $i' \geq i$.

Second, by induction on i we prove that until the first case occurs, $L^S(J_{\geq j_i}) \geq L(J_{\geq j_i})/2$. Since the first case is excluded, we have $\ell^S(j_i) > \ell(j_i)/2$ and the

inductive step follows by summing this inequality and the induction assumption for $i - 1$. (If $i = 1$ then the claim follows trivially.) ■

We conjecture that our analysis of GREEDY is not tight and in fact its approximation ratio is $e/(e - 1) \approx 1.58$. The following is an example of an $e/(e - 1)$ ratio for unit-length jobs: Let a be an integer and let $A = \{a + 1, a + 2, \dots, \lfloor ea \rfloor\}$. We have $\sum_{i \in A} 1/i = \ln \lfloor ea \rfloor - \ln a \leq \ln e = 1$ and for a sufficiently large, the sum is arbitrarily close to 1. Let $m = LCM\{a + 1, a + 2, \dots, \lfloor ea \rfloor\}$. That is, each of the numbers $a + 1, a + 2, \dots, ea$ divides m . The instance consists of m machines, and m jobs having window $a_j = i$ for each $i = a + 1, a + 2, \dots, \lfloor ea \rfloor$. The profits are all almost equal to 1, with a slight preference to jobs having long vacation.

The GREEDY schedule: In the first slot, the most profitable jobs are those with vacation $\lfloor ea \rfloor$, so these m jobs are scheduled first on each of the m machines. Next (on all m machines) are the jobs with vacation $\lfloor ea \rfloor - 1$, and so on. The resulting schedule has period $\lfloor ea \rfloor$ repeating on each machine $[\lfloor ea \rfloor, \lfloor ea \rfloor - 1, \dots, a, *, \dots, *]$, where stars denote a idle slots. The average profit per slot on each of the machines is therefore arbitrarily close to $(ea - a)/(ea) = (e - 1)/e$.

An optimal schedule: For each i , m/i machines schedule with no idle time all the jobs with window i . Since $\sum_{i \in A} 1/i \approx 1$, there are enough jobs to ‘saturate’ this way all the machines with no idle times. Therefore the average profit per slot is arbitrarily close to 1 and the approximation ratio tends to $e/(e - 1)$.

4 Unit Processing Times

We call an instance *aligned* if there exists an integer q such that for each j , $a_j = q2^{\alpha_j}$ for some integer $\alpha_j \geq 0$, or $a_j = 1$. (Note that this generalizes the case where all the windows are powers of 2.) The following result is due to [5].

Theorem 4.1. *An optimal solution for an aligned instance can be computed in polynomial time.*

For completeness, we describe such an optimal solution. Note that this solution works also if we start not with an empty schedule but with some schedule with period q , where some slots are already used by other jobs.

Schedule the jobs in order of increasing windows, always keeping the period of the schedule equal to the maximal window processed so far. The machines are utilized in an arbitrary order. When a new job is processed, if needed, increase the period by doubling the period and repeating the current schedule, until the period is equal to the currently processed window. Then, if possible, schedule the current job in an empty slot on some machine. Otherwise, if the current job is heavier than some of the scheduled jobs, run this job in a slot of the lightest scheduled job. (This light job can still be scheduled in some of the slots, due to a possible previous doubling of the period.)

An alternative view of this schedule S will be useful. Let j be the heaviest job such that $L(J_{\geq j}) \geq m$. Then the schedule above has the property that all jobs j' heavier than j are scheduled at their maximal rate, i.e., $\ell^S(j') = \ell(j')$,

and no jobs lighter than j occur in the schedule. The rate of the borderline job j is selected such that all the rates sum to 1.

It follows that a simple 2-approximation algorithm can be achieved by rounding the window of each job to the next higher power of 2 and using the optimal algorithm for aligned instances. Below we give two algorithms that refine this idea and achieve better approximation ratios. These algorithms are deterministic: randomization is used only for their analysis.

4.1 A 1.67-Approximation Algorithm for Arbitrary Profits

Algorithm SIMPLEALIGN

Produce 2 instances J' and J'' . The instance J' has all windows rounded up to the next power of 2; the instance J'' has all windows rounded up to the next number of the form $3 \cdot 2^\alpha$ for some integer $\alpha \geq 0$, with the exception of jobs with window 1 which remain unchanged. Produce optimal schedules S' and S'' for J' and J'' and choose the better one.

Theorem 4.2. SIMPLEALIGN is a 1.67-approximation algorithm for unit processing times and arbitrary profits.

Proof. Using Theorem 4.1, SIMPLEALIGN runs in polynomial time.

Consider a distribution which chooses S' with probability $2/5$ and S'' with probability $3/5$. We prove that the condition of Lemma 2.1 is satisfied with $R = 5/3$. We use the following claim, where by $\ell^{J'}(j)$ we denote the load of a job j rounded as in the instance J' , and similarly for $\ell^{J''}$ and J'' .

Claim 1. For any job j , $\frac{2}{5}\ell^{J'}(j) + \frac{3}{5}\ell^{J''}(j) \geq \ell(j)/R$.

Proof. We consider two cases:

(i) If for some integer α , $2^{\alpha+1} \leq a_j \leq 3 \cdot 2^\alpha$, then

$$\frac{2}{5}\ell^{J'}(j) + \frac{3}{5}\ell^{J''}(j) \geq \frac{2}{5} \cdot \frac{1}{2^{\alpha+2}} + \frac{3}{5} \cdot \frac{1}{3 \cdot 2^\alpha} = \frac{3}{5 \cdot 2^{\alpha+1}} \geq \frac{\ell(j)}{R}.$$

(ii) Otherwise, for some integer α , $3 \cdot 2^\alpha < a_j < 2^{\alpha+2}$, in which case

$$\frac{2}{5}\ell^{J'}(j) + \frac{3}{5}\ell^{J''}(j) \geq \frac{2}{5} \cdot \frac{1}{2^{\alpha+2}} + \frac{3}{5} \cdot \frac{1}{3 \cdot 2^{\alpha+1}} = \frac{1}{5 \cdot 2^\alpha} \geq \frac{\ell(j)}{R}.$$

■

Now, given an optimal schedule S^* and a job j , we prove the assumption of Lemma 2.1. We distinguish between two cases.

(a) First, assume that both $L^{S'}(J_{\geq j}) < m$ and $L^{S''}(J_{\geq j}) < m$. Then $L^{S'}(J_{\geq j}) = \sum_{j \in J_{\geq j}} \ell^{J'}(j)$, by the construction of an optimal schedule for aligned instances, and similarly for S'' . From Claim 1, we obtain $\mathbf{Exp}[L^S(J_{\geq j})] \geq L(J_{\geq j})/R \geq L^{S^*}(J_{\geq j})/R$.

- (b) Otherwise it must be the case that $L^{S'}(J_{\geq j}) = m$ or $L^{S''}(J_{\geq j}) = m$ (note that the load in a schedule cannot be larger than m). Either way, it implies that $L(J_{\geq j}) \geq m$ in the original instance. In both J' and J'' , the size of each window is at most doubled, therefore $L^{S'}(J_{\geq j}) \geq m/2$ and $L^{S''}(J_{\geq j}) \geq m/2$. Then the average load is bounded by $\mathbf{Exp}[L^S(J_{\geq j})] \geq 2/5 + 3/5 \cdot m/2 = 7m/10 \geq m/R \geq L^{S^*}(J_{\geq j})/R$.

The proof is completed by an application of Lemma 2.1. ■

4.2 A 1.39-Approximation Algorithm for Unit Profits

Now we focus on the special case where, for any job j , $b_j = 1$. Thus, the average profit is equal to the load of the schedule. In particular, if the schedule has no idle time, it is optimal.

Instead of using two schedules with periods 2 and 3 times a power of 2, the next algorithm uses one of k schedules with periods $q = k+1, k+2, \dots, 2k$ times a power of 2. For a large but constant k , the approximation ratio approaches $2 \ln 2 \approx 1.39$. The jobs with large windows are rounded similarly to the previous proof.

The jobs with small windows are handled separately. Given a constant q and a set of jobs with windows at most q , we find an optimal schedule with period q in polynomial time. If the number of machines is constant, then the number of such schedules is constant and we can find an optimal one simply by exhaustive search. (Note that we do not need to distinguish between different jobs having the same windows, as they have identical profits.)

For arbitrary m , we use Lenstra's polynomial algorithm for integer programming in fixed dimension (see [11]), similarly to its other applications in scheduling. Since q is a constant, we have a constant number of job types specified by their windows, and for each type of jobs we have a constant number of patterns specifying in which time slots it runs. The variables of the integer program correspond to the number of jobs of each type following each pattern; the number of these variables is a constant exponential in q . The constraints specify that (i) the number of scheduled jobs of each type equals to the number of such jobs in the instance, and that (ii) the number of jobs scheduled in any given time unit is at most m . Feasible integral solutions then correspond to schedules in a straightforward way. Given an instance, the integer program can be produced in linear time, and solved in time polylogarithmic in n and m (with a multiplicative constant doubly exponential in q).

Algorithm BESTALIGN

Let $\varepsilon > 0$ be given.

Let $K = \lceil 1/\varepsilon \rceil$. For all values $k \in \{K, 4K, \dots, 4^x K, \dots, 4^K K\}$ and for all values $q = \{k+1, k+2, \dots, 2k\}$ generate a schedule as follows.

Let J' be the set of jobs with windows at most $k/4$. Let J'' be an instance of jobs with windows larger than k . (Note that J' and J'' depend only on k .) Let J''' be an aligned instance obtained from J'' so that the

window of each job is rounded up to the next number of the form $q2^\alpha$, for some integer $\alpha \geq 0$.

Find the best schedule with period q for J' ; since q is a constant, this can be done as described in the previous paragraph. Then schedule J''' in the remaining slots, using the optimal algorithm for aligned instances.

Output the best schedule over all values of k and q .

Theorem 4.3. *BESTALIGN is $(2 \ln 2 + O(\varepsilon))$ -approximation algorithm for the jobs with unit processing times and unit profits.*

Thus, there exists an R -approximation algorithm for any $R > 2 \ln 2 \approx 1.386$.

Proof. By the above discussion, BESTALIGN can be implemented in polynomial time for any fixed $\varepsilon > 0$.

Let S^* be an optimal schedule for a given instance. For the proof, fix a value of k among those used in the algorithm, so that the contribution of the jobs with windows between $k/4$ and k to $L(S^*)$ is at most $\varepsilon \cdot L(S^*)$. Since there are $\lceil 1/\varepsilon \rceil$ choices of k , one of them has a sufficiently small contribution. This defines the sets of jobs J' and J'' . Let S' and S'' denote the schedule S^* restricted to J' and J'' , respectively, i.e., all the other jobs are simply removed from the schedule.

For any q , let S_q be the schedule produced by the algorithm for this choice of q and for k fixed as above. If any of S_q has load m , it is optimal for J and the theorem follows. Thus, for the remaining proof we can assume that S_q always schedules all the jobs in J''' with their maximal load, i.e., $L^{S_q}(J''') = L(J''')$.

We prove that one of the schedules S_q has load at least $3/4 \cdot L^{S'}(J') + L(J'')/(2 \ln 2 + O(\varepsilon))$. The theorem then follows since, by the choice of k , $L^{S^*}(J) \leq (1 + O(\varepsilon))(L^{S'}(J') + L^{S''}(J''))$; furthermore, $L^{S''}(J'') \leq L(J'')$, and $2 \ln 2 > 4/3$.

The schedule produced by BESTALIGN for J' has load at least $3/4$ of $L(S')$: If we take in S' a segment of length $q - k/4$ with the maximal load and append to it $k/4$ empty slots, we get a schedule with period q and load at least $3/4$ of $L(S')$. The optimal schedule for J' with period q chosen by BESTALIGN has at least the same load.

Now we complete the proof of the theorem by showing that for some distribution over the schedules S_q , $\mathbf{Exp}[L^{S_q}(J''')] \geq L(J'')/(2 \ln 2 + O(\varepsilon))$. Define the probability distribution so that the probability of choosing S_q is $\pi_q = X/(q - 1)$, where X is chosen so that the sum of the probabilities is 1, i.e.,

$$\left(\frac{1}{k} + \dots + \frac{1}{2k - 1} \right) X = 1. \tag{1}$$

We proceed with the proof job by job, as in Theorem 4.2. Since S_q schedules all the jobs with maximal rate, it is sufficient to prove for each job j that its expected load in J''' (i.e., after rounding) is at least $\ell(j)/(2 \ln 2 + O(\varepsilon))$. Assume that the integers $\alpha \geq 0$ and $t \in \{k, \dots, 2k - 1\}$ satisfy $t2^\alpha < a_j \leq (t + 1)2^\alpha$ (for each j , there exist unique values of t and α). Then the average load of j in J''' is

$$\begin{aligned} \sum_{q=k+1}^t \pi_q \cdot \frac{1}{q^{2\alpha+1}} + \sum_{q=t+1}^{2k} \pi_q \cdot \frac{1}{q^{2\alpha}} &= t \left(\sum_{q=k+1}^t \pi_q \cdot \frac{1}{2q} + \sum_{q=t+1}^{2k} \pi_q \cdot \frac{1}{q} \right) \frac{1}{t2^\alpha} \\ &\geq t \left(\sum_{q=k+1}^t \pi_q \cdot \frac{1}{2q} + \sum_{q=t+1}^{2k} \pi_q \cdot \frac{1}{q} \right) \ell(j). \end{aligned}$$

It remains to bound the coefficient on the right-hand side. By the definition of π_q , we have

$$\frac{\pi_q}{q} = \frac{X}{(q-1)q} = \left(\frac{1}{q-1} - \frac{1}{q} \right) X,$$

and substituting this and telescoping the sums we have

$$t \left(\sum_{q=k+1}^t \pi_q \cdot \frac{1}{2q} + \sum_{q=t+1}^{2k} \pi_q \cdot \frac{1}{q} \right) = tX \left(\frac{1}{2k} - \frac{1}{2t} + \frac{1}{t} - \frac{1}{2k} \right) = \frac{X}{2}.$$

Using (1), we have

$$\frac{2}{X} = 2 \left(\frac{1}{k} + \dots + \frac{1}{2k-1} \right) \leq 2 \int_k^{2k} \frac{dx}{x} + \frac{2}{k} = 2(\ln(2k) - \ln k) + \frac{2}{k} \leq 2 \ln 2 + O(\varepsilon).$$

This completes the proof of the theorem. ■

4.3 Instances with a Large Load

In the case where $p_j = b_j = 1$, the goal is to maximize the utilization of the machine. A simple observation is that the best schedule one can expect is one in which the machines are busy all the time. In this subsection we focus on two cases where such a schedule can be generated.

Our analysis of SIMPLEALIGN shows that if $L(J) \geq 5m/3$, then SIMPLEALIGN produces a schedule with load m : One of the schedules S' and S'' is guaranteed to have at least this load, and no larger load is possible.

The next theorem gives a condition to the optimality of *any* greedy algorithm in scheduling jobs with unit lengths and unit profits. In the resulting schedule, we get full utilization of all machines. In other words, no machine is ever idle, regardless of the jobs selected to be scheduled at any time slot.

Theorem 4.4. *Suppose that $n = km + r$ for some $r < m$. (i.e., $k = \lfloor n/m \rfloor$ and $r = n \bmod m$.) If $L(J) > mH_k - 1 + (r + 1)/(k + 1)$ then any algorithm with no intended idle times achieves the optimal profit. In particular, for $m = 1$, the condition is $L(J) > H_{n+1} - 1$.*

Proof. We show that if a machine is idle, $L(J)$ must be small. Assume that some machine is idle for the first time at t . Thus, none of the n jobs is available, meaning that all jobs are either processed on the other machines or on vacation.

Order the jobs by the last execution up to time t , including possible executions on the other $m - 1$ machines at time t . The last $m - 1$ jobs in this order are possibly executed at time t , we have no bound on their vacation, so each can contribute as much as 1 to $L(J)$. The previous m jobs in this order are scheduled no later than at time $t - 1$, they are on vacation at time t , thus they have $a_j \geq 2$ and contribute at most $1/2$ each to $L(J)$. Similarly, the previous m jobs are scheduled no later than at time $t - 2$ and contribute at most $1/3$ each to $L(J)$, and so on. The last $r + 1$ jobs contribute at most $1/(k + 1)$ each. Thus, the total load of the input is

$$L(J) \leq m - 1 + m \left(\frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{k} \right) + \frac{r + 1}{k + 1} = mH_k - 1 + \frac{r + 1}{k + 1}.$$

Therefore, $L(J) > mH_k - 1 + (r + 1)/(k + 1)$ implies that there is no idle time. ■

Acknowledgment. We thank Gerhard Woeginger for stimulating discussions on this paper.

References

1. S. Anily, J. Bramel, Periodic scheduling with service constraints. *Operations Research*, Vol. 48, pp. 635-645, 2000.
2. H. Aydin, R. Melhem, D. Mosse, P. Mejia-Alvarez, Optimal Reward-Based Scheduling of Periodic Real-Time Tasks. In 20th IEEE Real-Time Systems Symp., 1999.
3. A. Bar-Noy, R. Bhatia, J. Naor, B. Schieber, Minimizing Service and Operation Costs of Periodic Scheduling. In *Proc. of SODA*, 1998.
4. A. Bar-Noy and R. E. Ladner. Windows Scheduling Problems for Broadcast Systems. In *Proc. of SODA*, 2002.
5. A. Bar-Noy, R. E. Ladner, T. Tamir. Windows Scheduling as a Restricted Version of Bin Packing. In *Proc. of SODA*, 2004.
6. A. Bar-Noy, R. E. Ladner, T. Tamir. A General Buffer Scheme for the Windows Scheduling Problem. In *Proc. of WEA*, 2005.
7. S.K.Baruah, N.K.Cohen, C.G.Plaxton, D.A.Varvel, Proportionate Progress: A Notion of Fairness in Resource Allocation. *Algorithmica*, 15(6), pages 600 –625, 1996.
8. S. K. Baruah, S-S. Lin. Pfair Scheduling of Generalized Pinwheel Task Systems. *IEEE Trans. on Comp.*, Vol. 47, 812–816, 1998.
9. J. Chung, J.W.S. Liu, K. Lin. Scheduling Periodic Jobs that Allow Imprecise Results. *IEEE Trans. on Comp.*, Vol. 39 (9),pp. 1156–1174, 1990.
10. R. M. Karp. A Characterization of the Minimum Cycle Mean in a Digraph. *Discrete Math.*, 23:309-311, 1978.
11. H.W. Lenstra. Integer programming with a fixed number of variables. *Mathematics of Operations Research*, Vol. 8, 538–548, 1983.
12. C. L. Liu, J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, Vol. 20, No. 1, 46-61, 1973.
13. R. Tijdeman. The Chairman assignment Problem. *Discrete Mathematics*, Vol. 32, 323-330, 1980.

Online Bin Packing with Cardinality Constraints

Leah Epstein*

Department of Mathematics, University of Haifa, 31905 Haifa, Israel
lea@math.haifa.ac.il

Abstract. We consider a one dimensional storage system where each container can store a bounded amount of capacity as well as a bounded number of items $k \geq 2$. This defines the (standard) bin packing problem with cardinality constraints which is an important version of bin packing, introduced by Krause, Shen and Schwetman already in 1975. Following previous work on the unbounded space online problem, we establish the *exact* best competitive ratio for bounded space online algorithms for every value of k . This competitive ratio is a strictly increasing function of k which tends to $\Pi_\infty + 1 \approx 2.69103$ for large k . Lee and Lee showed in 1985 that the best possible competitive ratio for online bounded space algorithms for the classical bin packing problem is the sum of a series, and tends to Π_∞ as the allowed space (number of open bins) tends to infinity. We further design optimal online bounded space algorithms for *variable sized bin packing*, where each allowed bin size may have a distinct cardinality constraint, and for the *resource augmentation* model. All algorithms achieve the *exact* best possible competitive ratio possible for the given problem, and use constant numbers of open bins. Finally, we introduce unbounded space online algorithms with smaller competitive ratios than the previously known best algorithms for small values of k , for the standard cardinality constrained problem. These are the first algorithms with competitive ratio below 2 for $k = 4, 5, 6$.

1 Introduction

The classical bin packing problem [19,5,3] assumes no limit on the *number* of items which may be packed in a single bin. In practice, many applications require such a bound either due to overheads or additional constraints that are not modeled. For example, a disk cannot keep more than a certain number of files, even if these files are indeed very small. A processor cannot run more than a given number of tasks during a given time, even if all tasks are very short. The problem where there is a given bound $k > 1$ on the number of items which can co-exist in one bin, is called “Bin Packing with Cardinality Constraints” [11,1]. We consider several versions of this problem.

We first define the classic online bin packing problem. In this problem, we receive a sequence σ of *items* $p_1, p_2 \dots p_n$, arriving one by one. The values p_i are the *sizes* of the items. We have an infinite supply of *bins*, each of which is of

* Research supported in part by the Israel Science Foundation (grant no. 250/01).

unit size. An item must be assigned to a bin upon arrival, so that the sum of items in no bin exceeds 1. A bin is *empty* if no item is assigned to it, otherwise it is *used*. The goal is to minimize the number of bins used. In the **cardinality constrained bin packing problem**, an additional constraint is introduced. A parameter k bounds the number of items that can be assigned to a single bin.

The standard measure of algorithm quality for online bin packing is the *asymptotic competitive ratio*, which we now define. For a given input sequence σ , let $\mathcal{A}(\sigma)$ (or \mathcal{A}) be the number of bins used by algorithm \mathcal{A} on σ . Let $OPT(\sigma)$ (or OPT) be the cost of an optimal offline algorithm which knows the complete sequence of items in advance, i.e., the minimum possible number of bins used to pack items in σ . The *asymptotic performance ratio* for an algorithm \mathcal{A} is defined to be $\mathcal{R}(\mathcal{A}) = \limsup_{n \rightarrow \infty} \sup_{\sigma} \left\{ \frac{\mathcal{A}(\sigma)}{OPT(\sigma)} \mid OPT(\sigma) = n \right\}$.

In the *resource augmented* bin packing problem, the online algorithm is supplied with larger bins at its disposal than those of the offline algorithm that it is compared to. The competitive ratio then becomes a function of the bin size. All online bins are of the same size, and all the offline bins are of the same size, but these two sizes are not necessarily the same.

In the *variable-sized* bin packing problem, there is a supply of several bin sizes that can be used to pack the items. The cost of an algorithm is the sum of sizes of used bins. In this problem, the generalization into cardinality constrained packing assumes that each bin size $s_i \leq 1$ is associated a parameter k_i which bounds the number of items that can be packed into such a bin.

We stress the fact that items arrive *online*, this means that each item must be assigned in turn, without knowledge of the next items. We consider *bounded space* algorithms, which have the property that they only have a constant number of bins available to accept items at any point during processing, these bins are also called “open bins”. The bounded space assumption is a quite natural one. Essentially the bounded space restriction guarantees that output of packed bins is steady, and that the packer does not accumulate an enormous backlog of bins which are only output at the end of processing.

Previous Results. Cardinality constrained bin packing was studied in the offline environment already in 1975 by Krause, Shen and Schwetman [12,13]. They showed that the performance guarantee of the well known First Fit algorithm is at most $2.7 - \frac{12}{5k}$. Additional results were offline approximation algorithms of performance guarantee 2. These results were later improved in two ways. Kellerer and Pferschy [11] designed an improved offline approximation algorithm with performance guarantee 1.5 and finally a PTAS was designed in [2] (for a more general problem). On the other hand, Babel et al. [1], designed a simple *online* algorithm with competitive ratio 2 for any value of k . They also designed improved algorithms for $k = 2, 3$ of competitive ratios $1 + \frac{\sqrt{5}}{5} \approx 1.44721$ and 1.8 respectively. The same paper [1] also proved an almost matching lower bound of $\sqrt{2} \approx 1.41421$ for $k = 2$ and mentioned that the lower bounds of [22,20] for the classic problem hold for cardinality constrained bin packing as well. The lower bound of 1.5 given by Yao [22] holds for small values of $k > 2$ and the lower

bound of 1.5401 given by Van Vliet [20] holds for sufficiently large k . No other lower bounds are known.

For the classic bin packing problem, Lee and Lee [14] presented an algorithm called HARMONIC, which partitions items into $m > 1$ classes and uses bounded space of at most $m - 1$ open bins. For any $\varepsilon > 0$, there is a number m such that the HARMONIC algorithm that uses m classes has a performance ratio of at most $(1 + \varepsilon)\Pi_\infty$ [14], where $\Pi_\infty \approx 1.69103$ is the sum of series (see Section 2). They also showed there is no bounded space algorithm with a performance ratio below Π_∞ . Currently the best known unbounded space upper bound is 1.58889 due to Seiden [17].

The first to investigate the variable sized bin packing problem were Friesen and Langston [10]. Csirik [4] proposed the VARIABLE HARMONIC algorithm and showed that it has performance ratio at most Π_∞ . Seiden [16] showed that this algorithm is optimal among bounded space algorithms. Unbounded space variable sized bin packing was studied also in [18].

The resource augmented bin packing problem was studied by Csirik and Woeginger [6]. They showed that the optimal bounded space asymptotic performance ratio is a function $\rho(b)$ of the online bin size b . Unbounded space resource augmented bin packing was studied also in [8].

Our Results. We consider bounded space algorithms. For every value of k , we find the best competitive ratio of any online bounded space algorithm. The competitive ratio is a strictly increasing function of k and for large enough k it approaches $1 + \Pi_\infty \approx 2.69103$ where Π_∞ is the best competitive ratio shown by [14] for the classic bounded space problem. This is a surprising feature of the problem, since one would expect this value to simply tend to Π_∞ as k grows.

We further consider the resource augmented problem where the online algorithm may use larger bins compared to the optimal offline algorithm. We design optimal online algorithms for this problem as well. For large enough values of k , the competitive ratios again approach values which differ by 1 from the best competitive ratios for the classic resource augmented problem [6]. We show that the competitive ratios for our problem never drops below 1 (unlike the case studied in [6]) and identify the cases where the competitive ratio is exactly 1. For variable sized bin packing, we design algorithms of the exact optimal competitive ratios (among bounded space algorithms) for any set of bins and cardinality constraints. An interesting feature is that we prove the algorithms have optimal competitive ratios, even though we do not know what these ratios are.

A main difference between our results for bounded space algorithms and the results of [14,6,16] is that our algorithms have exactly the best possible competitive ratio achievable by bounded space online algorithms. The algorithms for variants of the classical problem have competitive ratios which tend to the best competitive ratio as the number of open bins grows without bound. Our algorithms just need a constant number of open bins to achieve the best competitive ratios. Therefore we need to be very careful in the analysis since unlike the classic problem, we may not lose any small constants, which depend on the number of open bins, in the analysis.

For small values of k we design several new unbounded space algorithms, based on combination of large and small items together in bins (see [14,15,17]), according to sizes of small items. We prove the competitive ratios of our algorithms for $k = 3, 4, 5, 6$ are $\frac{7}{4} = 1.75$, $\frac{71}{38} \approx 1.86842$, $\frac{771}{398} \approx 1.93719$, $\frac{287}{144} \approx 1.99306$ (respectively). This improves on the bounds $\frac{5}{3} = 1.8$ ($k = 3$) and 2 ($k = 4, 5, 6$) of [1].

2 Optimal Algorithms for Bounded Space Packing

In this section we define bounded space algorithms of optimal competitive ratio for each value of $k > 1$. For every $k > 1$, we define an online bounded space algorithm which packs at most k items in each bin and uses at most $k - 1$ open bins. We show that this algorithm is the best possible among bounded space algorithms. We use the well known sequence $\pi_i, i \geq 1$ which is often used for bin packing, let $\pi_1 = 2, \pi_{i+1} = \pi_i(\pi_i - 1) + 1$ and let $\Pi_\infty = \sum_{i=1}^\infty \frac{1}{\pi_i - 1} \approx 1.69103$.

This sequence was used by Lee and Lee in [14] and by Van Vliet [20]. Adaptations of this sequence were later used in several papers including [6,18].

The sequence is constructed in a way that $1 - \sum_{i=1}^j \frac{1}{\pi_i} = \frac{1}{\pi_{j+1} - 1}$ (which can be easily shown by induction using the sequence definition). This means that each time the next value π_i is picked to be an integer, such that all items $\frac{1}{\pi_j}$ for $j \leq i$ can fit together in a bin leaving some empty space. Note that Π_∞ is a lower bound on the best competitive ratio for classical bounded space bin packing, and there exists a sequence of bounded space algorithms, with an increasing sequence of open bins whose competitive ratios tend to this value [14,21]. The algorithms in this section are based on the algorithms in [14] with some differences in the construction and proof due to the cardinality constraint (which also increases the competitive ratio by 1 for large values of k). We also would like to achieve the best possible bound for every value of k separately, and not only in the limit.

Let $\mathcal{R}_k = \sum_{i=1}^k \max \left\{ \frac{1}{\pi_i - 1}, \frac{1}{k} \right\}$. We show that for every value of k , the best competitive ratio is *exactly* \mathcal{R}_k . We start with some properties of \mathcal{R}_k as a function of k .

Theorem 1. *The value of \mathcal{R}_k is a strictly increasing function of k , such that $\frac{3}{2} \leq \mathcal{R}_k < \Pi_\infty + 1$, and $\lim_{k \rightarrow \infty} \mathcal{R}_k = \Pi_\infty + 1 \approx 2.69103$.*

Proof. We first find the value of \mathcal{R}_2 . Since $\pi_1 = 2$ and $\pi_2 = 3$, we have $\mathcal{R}_2 = \frac{3}{2} = 1.5$. Note also that $\mathcal{R}_3 = \frac{11}{6} \approx 1.83333, \mathcal{R}_4 = 2, \mathcal{R}_5 = 2.1$ and $\mathcal{R}_6 = \frac{13}{6} \approx 2.16666$. Next we show the monotonicity of \mathcal{R}_k . For a given k , let $j_k = \min_{1 \leq j \leq k} \{j | \frac{1}{k} \geq \frac{1}{\pi_j - 1}\}$. The value j_k exists for all k since $\pi_k - 1 \geq k$ for all k . Then

we have $\mathcal{R}_k = \sum_{i=1}^{j_k-1} \frac{1}{\pi_i - 1} + \sum_{i=j_k}^k \frac{1}{k} = \sum_{i=1}^{j_k-1} \frac{1}{\pi_i - 1} + \frac{k - j_k + 1}{k}$. By definition of the values

j_i , clearly $j_k \leq j_{k+1}$. Therefore $\mathcal{R}_{k+1} - \mathcal{R}_k = \sum_{j_k}^{j_{k+1}-1} \frac{1}{\pi_i-1} + \frac{k-j_{k+1}+2}{k+1} - \frac{k-j_k+1}{k} > \frac{j_{k+1}-j_k}{k+1} + \frac{1-j_{k+1}}{k+1} - \frac{1-j_k}{k} = \frac{j_k-1}{k} - \frac{j_k-1}{k+1} \geq 0$. We deduce the strict inequality above by $\pi_i - 1 < k + 1$ which holds for $i < j_{k+1}$.

An upper bound on \mathcal{R}_k follows from $\mathcal{R}_k = \sum_{i=1}^k \max \left\{ \frac{1}{\pi_i-1}, \frac{1}{k} \right\} < \sum_{i=1}^k \frac{1}{\pi_i-1} + \sum_{i=1}^k \frac{1}{k} < \Pi_\infty + 1$. We next show that \mathcal{R}_k tends to this value. For a given $\varepsilon > 0$,

let ℓ be a value such that $\sum_{i=1}^{\ell} \frac{1}{\pi_i-1} \geq \Pi_\infty - \frac{\varepsilon}{2}$, and $\ell \geq \frac{2}{\varepsilon}$. Let $k = \ell^2$, then

$$\mathcal{R}_k \geq \sum_{i=1}^{\ell} \frac{1}{\pi_i-1} + \sum_{i=\ell+1}^k \frac{1}{\ell^2} \geq \Pi_\infty - \frac{\varepsilon}{2} + \frac{\ell^2-\ell}{\ell^2} \geq \Pi_\infty - \frac{\varepsilon}{2} + 1 - \frac{\varepsilon}{2} = \Pi_\infty + 1 - \varepsilon \quad \square$$

Next we define the algorithm **CARDINALITY CONSTRAINED HARMONIC_k** (CCH_k) which is an adaptation of the algorithm **HARMONIC_k** defined originally by Lee and Lee [14]. The fundamental idea of “harmonic-based” algorithms is to first classify items by size, and then pack an item according to its class (as opposed to letting the exact size influence packing decisions).

For the classification of items, we partition the interval $(0, 1]$ into sub-intervals. We use $k - 1$ sub-intervals of the form $(\frac{1}{i+1}, \frac{1}{i}]$ for $i = 1, \dots, k - 1$ and one final sub-interval $(0, \frac{1}{k}]$. Each bin will contain only items from one sub-interval (type). Items in sub-interval i are packed i to a bin for $i = 1, \dots, k - 1$, thus keeping the cardinality constraint. The items in interval k are packed k to a bin. A bin which received the full amount of items (according to its type) is closed, therefore at most $k - 1$ bins are open simultaneously (one per interval, except for $(\frac{1}{2}, 1]$).

To prove the upper bound on the competitive ratio, we use a simplified version of a theorem 9 stated in section 5. We use the technique of weighting functions. This technique was originally introduced for one-dimensional bin packing algorithms [19]. The version we use is as follows.

Theorem 2. *Consider a bin packing algorithm. Let w be a weight measure. Assume that for every output of the algorithm, the number of bins used by an algorithm ALG is bounded by $X(\sigma) + c$ for some constant c , where $X(\sigma)$ is the sum of weights of all items in the sequence according to weight measure w . Denote by W the maximum amount of weight that can be packed into a single bin of an offline algorithm according to measure w . Then the competitive ratio of the algorithm is upper bounded by W .*

We define weights as follows. The weight of item x is denoted $w(x)$. The weight of an item in interval $(\frac{1}{i+1}, \frac{1}{i}]$, for $i = 1, \dots, k - 1$, is $\frac{1}{i}$. The weight of an item in interval $(0, \frac{1}{k}]$ is $\frac{1}{k}$. Recall that except for $k - 1$ open bins that may not receive the full amount of items, each output bin receives a total weight of 1. A closed bin for items in interval $(\frac{1}{i+1}, \frac{1}{i}]$ receives i items, of weight $\frac{1}{i}$ each. A closed bin for items in interval $(0, \frac{1}{k}]$ receives k items, of weight $\frac{1}{k}$ each. Therefore we get $CCH_k(\sigma) \leq X(\sigma) + k - 1$.

Theorem 3. *For every k , the competitive ratio of CCH_k is \mathcal{R}_k , and no online algorithm which uses bounded space can have a better competitive ratio.*

Proof. We prove the upper bound first. Let $\varepsilon > 0$ a very small constant, such that $\varepsilon \ll \frac{1}{k\pi_{k+1}}$. We claim that the maximum weight of a single bin is achieved for the following set of items. $f_1 \geq \dots \geq f_k$, so that $f_i = \frac{1}{\pi_i} + \varepsilon$. This set of items fits in a single bin according to the definition of the sequence π_j . Their sum of weights is exactly $\mathcal{R}_k = \sum_{i=1}^k \max \left\{ \frac{1}{\pi_{i-1}}, \frac{1}{k} \right\}$.

To show that the maximum weight of any bin is indeed \mathcal{R}_k , consider an arbitrary set S of $\ell \leq k$ items which fits into one bin. If $\ell \neq k$, we add $k - \ell$ items of size zero and give them weight $\frac{1}{k}$. This may only increase the sum of weights. Let $g_1 \geq \dots \geq g_k$ be the sorted list of items. If $g_i \in (\frac{1}{\pi_i}, \frac{1}{\pi_{i-1}}]$ holds for all i such that $\pi_i \leq k$ and $g_i \in [0, \frac{1}{k}]$ for all i such that $\pi_i > k$, then the weight of the items of S is exactly \mathcal{R}_k . Otherwise let i be the first index of item that does not satisfy the above. If $w(g_i) = \frac{1}{k}$ we get that $\sum_{j=1}^k w(g_j) =$

$$\sum_{j=1}^{i-1} w(g_j) + \sum_{j=i}^k w(g_j) = \sum_{j=1}^{i-1} w(f_j) + \sum_{j=i}^k \frac{1}{k} \leq \sum_{j=1}^k w(f_j) = \mathcal{R}_k.$$

Otherwise, assume $w(g_i) > \frac{1}{k}$. Due to the greedy construction of the sequence π_j , and since $g_i \notin (\frac{1}{\pi_i}, \frac{1}{\pi_{i-1}}]$, we get that $g_i \leq \frac{1}{\pi_i}$ and therefore $w(g_i) < \frac{1}{\pi_{i-1}} = w(f_i)$. Let i' be the smallest index such that $w(g_{i'}) = \frac{1}{k}$ (this value exists as mentioned above since $k \leq \pi_k - 1$). If $i' = i + 1$ we get that $w(g_i) < w(f_i)$, and for $j \geq i'$, $\frac{1}{k} = w(g_j) \leq w(f_j)$. In this case we have $\sum_{j=1}^k w(g_i) < \sum_{j=1}^k w(f_i) = \mathcal{R}_k$.

Otherwise consider the values of j such that $i \leq j \leq i' - 1$. We have $g_j \leq \frac{1}{\pi_i}$, and therefore according to the weight definition for $x > \frac{1}{k}$, $\frac{w(g_j)}{g_j} \leq \frac{\pi_i + 1}{\pi_i}$ for $i \leq j \leq i' - 1$. Given that for $j < i$, $g_j \in (\frac{1}{\pi_j}, \frac{1}{\pi_{j-1}}]$, we have $\sum_{j=i}^k g_j \leq \frac{1}{\pi_{i-1}}$

and therefore $\sum_{j=i}^{i'-1} w(g_j) \leq \frac{\pi_i + 1}{\pi_i^2 - \pi_i}$. However $w(f_i) + w(f_{i+1}) = \frac{1}{\pi_{i-1}} + \frac{1}{\pi_{i+1}-1} =$

$$\frac{1}{\pi_{i-1}} + \frac{1}{\pi_i^2 - \pi_i} = \frac{\pi_i + 1}{\pi_i^2 - \pi_i}.$$

Summarizing, we get $\sum_{j=1}^k w(g_j) = \sum_{j=1}^{i-1} w(g_j) + \sum_{j=i}^{i'-1} w(g_j) +$

$$\sum_{j=i'}^k w(g_j) \leq \sum_{j=1}^{i-1} w(f_j) + w(f_i) + w(f_{i+1}) + \sum_{j=i'}^k \frac{1}{k} \leq \sum_{j=1}^k w(f_i) = \mathcal{R}_k.$$

The proof of the lower bound is similar to previously known lower bound proofs for bounded space algorithms, see [14,6]. To prove the lower bound, let N be a large constant, and $\delta > 0$ a very small constant, such that $\delta \ll \frac{1}{k\pi_{k+1}}$. We construct the following sequence. The sequence has k phases. Phase i contains N items of size $\frac{1}{\pi_i} + \delta$. Let K be the number of bins that may be open simultaneously. Except for at most K bins, all bins of each phase are closed after the phase. Such bins can be filled by a maximum amount of $\min\{\pi_i - 1, k\}$ items. Therefore phase i contributes at least $\frac{N}{\min\{\pi_i - 1, k\}} - K = N \max\{\frac{1}{\pi_i - 1}, \frac{1}{k}\} - K$ closed

bins to the output. The optimal packing of the sequence contains N identically packed bins with one item of each phase per bin. We get that the competitive ratio is at least $\mathcal{R}_k - \frac{kK}{N}$. This approaches \mathcal{R}_k for large enough N . \square

3 Extension to Resource Augmentation

Following the work of [6] which studied resource augmentation for the classic bin packing problem, we show that the algorithms defined in the previous section are optimal in a resource augmented environment as well.

We compare an online algorithm which uses bins of size 1 to an optimal offline algorithm whose bins are of size $\frac{1}{b}$. We assume that all item sizes are bounded by $\frac{1}{b}$. This problem definition is equivalent to the alternative definition where items have sizes in $(0, 1]$, the online algorithm uses bins of size b and the offline algorithm uses bins of size 1. The competitive ratio for bounded cardinality k is measured as a function of $b > 1$. The best competitive ratio for bounded space algorithms and unrestricted online algorithms are denoted $\mathcal{R}_k(b)$ and $r_k(b)$ (respectively). We note a fundamental difference between the resource augmented problem associated with the classic bin packing problem and the problem studied in this paper. As we show later in this section, the competitive ratio is never below 1 for our problem, whereas the classic problem has a competitive ratio below 1 for $b \geq 2$ [6,8].

We show that the competitive ratio (even for unbounded space algorithms) cannot actually reach 1 if $b < k$ and is exactly 1 for $b = k$ (the proof is omitted).

Theorem 4. *For all values of b, k such that $b < k$ we have $\mathcal{R}_k(b) \geq r_k(b) > 1$. For all values of b, k such that $b \geq k$, we have $r_k(b) = \mathcal{R}_k(b) = 1$.*

The algorithms are defined exactly as in the previous section. However this means that some of the defined classes do not exist if b is large enough. Note that the algorithm for the case $b \geq k$ becomes exactly Next Fit.

To define the competitive ratio, we first define sequences $\pi_i(b)$ and $\Pi_i(b)$, originally defined by [6] as follows. $\Pi_0(b) = 0$, $\pi_1(b) = \lfloor b \rfloor + 1$, $\Pi_1(b) = \frac{1}{\pi_1(b)}$, $\pi_i(b) = \left\lfloor \frac{1}{\frac{1}{b} - \Pi_{i-1}(b)} \right\rfloor + 1$ and $\Pi_i(b) = \Pi_{i-1}(b) + \frac{1}{\pi_i(b)}$. The intuition behind this function is to find a sequence of integers, such that the next integer at each point is picked greedily to be minimal, and the sum of their reciprocals is less than $\frac{1}{b}$. The values of $\pi_i(b)$ satisfy $\pi_i(b) > b$. We can show that the values are strictly increasing as a function of b . Clearly the values are non-decreasing. If two values are the same we let $\pi_i(b) = \pi_{i+1}(b) = f$ be these identical values. Then we argue that $\pi_i(b)$ should have been chosen to be at most $f - 1$. To see that note that $\frac{1}{b} - \Pi_{i-1}(b) > \frac{2}{f} \geq \frac{1}{f-1}$. This holds for all $f \geq 2$.

Csirik and Woeginger [6] introduced the function $\rho(b) = \sum_{i=1}^{\infty} \frac{1}{\pi_i(b)-1}$ and showed that this is the best possible competitive ratio with resource augmentation b for the classic bin packing problem. Note that $\rho(1) = \Pi_{\infty} \approx 1.69103$. We can prove the following theorems whose proofs are omitted.

Theorem 5. *For every k , the competitive ratio of CCH_k (defined in the previous section) is $\mathcal{R}_k(b) = \sum_{i=1}^k \max \left\{ \frac{1}{\pi_i(b)-1}, \frac{1}{k} \right\}$, and no online algorithm which uses bounded space can have a better competitive ratio.*

Theorem 6. *The value of $\mathcal{R}_k(b)$ for a fixed value of b is an increasing function of k , such that $1 \leq \mathcal{R}_k(b) < \rho(b) + 1$, and $\lim_{k \rightarrow \infty} \mathcal{R}_k = \rho(b) + 1$.*

4 Extension to Variable Sized Bins

Following the work of Seiden [16] we design optimal online bounded space algorithm for the case of variable sized bins. Similarly to that case and other work on variable sized bins [7], we design algorithms for any set of bin sizes, we prove their optimality, however we do not know their competitive ratios. Our algorithms are based on the VARIABLE HARMONIC algorithms of Csirik [4]. The optimality of these algorithms among the class of bounded space algorithms was proved in [16]. As in previous sections, the main difference between these algorithms and our algorithms is in the way that small items are packed. As in previous sections, our algorithms have the exact best possible competitive ratio for a given set of bins and cardinality constraints, this with a constant number of open bins that can be easily computed (as a function of the bins sizes and constraints). The algorithms for the classical problem get close to the best possible competitive ratio as the number of open bins grows without bound.

In order to define our general algorithm CARDINALITY CONSTRAINED VARIABLE HARMONIC (CCVH) we use some definitions. Let the bins sizes be $s_1 < \dots < s_m = 1$. Let their cardinality constraints be k_1, \dots, k_m (respectively). We define a set of critical sizes for each bin in the following way. Let $T_i = \left\{ \frac{s_i}{j} \mid 1 \leq j \leq k_i \right\}$ and $T = \bigcup_{1 \leq i \leq m} T_i$. Let $|T| = M$ and the members of T be $1 = t_1 > t_2 > \dots > t_M$. The type of a size t_r is defined to a value $i(r)$ such that $t_r \in T_{i(r)}$ (ties are broken arbitrarily). In this case the order of t_r is $\ell(r) \leq k_i$ such that $t_r = \frac{s_{i(r)}}{\ell(r)}$.

We again classify items into intervals whose right endpoint is a critical size. This associates an item with an type and order. Afterwards we pack an item according to its type and order (here as well as in the previous sections, the exact size does not influence packing decisions). Each bin will contain items of a single interval.

Since $M = |T| \leq \sum_{i=1}^m k_i$, there is a bounded number of pairs of type and order.

For the classification of items, we partition the interval $(0, 1]$ into sub-intervals. The “small” interval is $(0, t_M]$. The other intervals are $(t_{j+1}, t_j]$ for $j = 1, \dots, M - 1$. Each bin will contain only items from one pair of type and order. Items in the sub-interval whose right endpoint is t_r are packed into bins of size $s_{i(r)}$. The items in this interval are packed $\ell(r)$ to a bin, thus keeping the cardinality constraints. Note that at most $M - m$ bins are open simultaneously, since a bin which received the full amount of items (according to its type) is closed.

The differences with algorithms for the classic variable sized bin packing problem are as follows. The condition for an item to be “small” (i.e. in the “small” interval) is determined by the cardinality constraints. Items cannot be packed using Next Fit due to these constraints. Moreover, in [16] the smallest items are packed into bins of size 1. In that case it is actually possible to pack the small items into any type of bin. Here the type of bin for the small items must be $s_{i(M)}$ (if there exists another size i' such that $t_M \in T'_{i'}$, that size can be used for the small items as well).

The following theorem is used in [16] to prove upper bounds on the competitive ratio of algorithms for variable sized bins.

Theorem 7. *Consider a bin packing algorithm. Let w be a weight measure. Assume that for every output of the algorithm, the cost of all the bins used by the algorithm ALG is bounded by $X(\sigma) + c$ for some constant c , where $X(\sigma)$ is the sum of weights of all items in the sequence according to weight measure w . Denote by W_i the maximum amount of weight that can be packed into a single bin of size s_i of an offline algorithm according to measure w . Then the competitive ratio of the algorithm is upper bounded by $\max_{1 \leq i \leq m} \left\{ \frac{W_i}{s_i} \right\}$.*

We assign weights to items in the following way. A weight of an item x is again denoted by $w(x)$. An item of interval $(0, t_M]$ receives weight $\frac{s_{i(M)}}{\ell(M)}$ (note that $\ell(M) = k_{i(M)}$). An item of interval $(t_{j+1}, t_j]$ receives weight $\frac{s_{i(j)}}{\ell(j)}$. Each closed bin of interval $(0, t_M]$ is of size $s_{i(M)}$, it receives $\ell(M)$ items and thus the weight of items packed in it is equal to its size. Each closed bin of interval $(t_{j+1}, t_j]$ is of size $s_{i(j)}$. It receives $\ell(j)$ items and thus the weight of items packed in it is equal to its size. Therefore the cost of the algorithm differs from the total weight of all items by the cost of all open bins, which is clearly bounded by $M - m$.

We can now use Theorem 7 to prove the following theorem (the proof is omitted).

Theorem 8. *For a given set of bins sizes and cardinality constraints, the algorithm CVH is an optimal online algorithm.*

5 Improved Unbounded Space Algorithms for Small Values of k

We first design an algorithm for $k = 3$. Already the algorithm of [1] has a competitive ratio lower than the best bounded space algorithm ($\frac{9}{5} = 1.8$ which is smaller than $\frac{11}{6}$). We design an algorithm which uses a more careful partition into classes and has competitive ratio $\frac{7}{4} = 1.75$. The algorithm is based on the idea of the HARMONIC algorithm, and its generalizations (see [14,15,17,9]). In these generalizations, items of two intervals are combined together in the same bins. We would like to use a similar approach, however the boundaries of intervals are chosen with accord to cardinality constraints.

We use the following five intervals. $A = (\frac{2}{3}, 1]$, $B = (\frac{1}{2}, \frac{2}{3}]$, $C = (\frac{1}{3}, \frac{1}{2}]$, $D = (\frac{1}{6}, \frac{1}{3}]$, $E = (0, \frac{1}{6}]$. Items which belong to an interval I are called items of type I , type I items, or simply I items. Items of types A, C and D are packed independently of any other items, one, two and three items per bin, respectively. Note that it is always possible to combine one item of type B with two items of type E . Therefore, each item of type E receives a color upon arrival, white or red. White items are packed in separate bins (three per bin) whereas red items are packed two per bin, and combined with one type B item. If there exists such an open bin, the red type E items are added there. Otherwise once a type B item arrives later, it is added to a bin with two type E items. The colors are assigned so that an α fraction of the type E items are red. We use $\alpha = \frac{1}{4}$. Therefore every fourth type E item is red, and all others are white.

We define a bin as incomplete in the four following packings. 1. A bin with a single C item, 2. A bin with only one or two D items, 3. A bin with a single red E item, 4. A bin with one or two white E items, 5. A bin with one red E item (and possibly a B item as well).

At every time, the algorithm can have at most five incomplete bins, one for each combination. Therefore upon termination, except for at most four incomplete bins, all bins can be packed as follows. 1. A single A item, 2. Two C items, 3. Three D items, 4. One B item, 5. Two red E items, 6. Three white E items, 7. One B item and two red E items.

According to the definition of the algorithm, we never have a situation where one bin has only a B item, and another bin has two red E items. This is true since a new bin is opened for such items only if they cannot join a previously opened bin.

The algorithm is therefore at one of the following two situations. 1. There are no bins with two red E items with no B item. 2. There are no bins with one B item and no E items.

We assign two weights to each item, according to the two scenarios. The weights are assigned according to types of items. We use $w_1(I)$ and $w_2(I)$ to denote the weights of type I items according to the two weight functions. Let $w_1(A) = w_2(A) = 1$, $w_1(B) = 1$, $w_2(B) = 0$, $w_1(C) = w_2(C) = \frac{1}{2}$, $w_1(D) = w_2(D) = \frac{1}{3}$, $w_1(E) = \frac{1-\alpha}{3} = \frac{1}{4}$, $w_2(E) = \frac{\alpha+2}{6} = \frac{3}{8}$.

The weights are defined so that in the first scenario, on average all bins (but at most four) have a total amount of weight of at least 1 packed into them according to the first weight measure, and otherwise the same property holds according to the second weight measure.

We use the following theorem, see Seiden [17].

Theorem 9. *Consider a bin packing algorithm. Let w_1, w_2 be two weight measures. Assume that for every output of the algorithm, there exists i ($i = 1$ or $i = 2$) such that the number of bins used by the algorithm ALG is bounded by $X_i(\sigma) + c$ for some constant c , where $X_i(\sigma)$ is the sum of weights of all items in the sequence according to weight measure w_i . Denote by W_i the maximum amount of weight that can be packed into a single bin according to measure*

w_i ($i = 1, 2$). Then the competitive ratio of the algorithm is upper bounded by $\max(W_1, W_2)$.

To use the theorem, we need to prove that for every input $ALG \leq X_i(\sigma) + c$ for some i . We ignore the (at most five) incomplete bins, which adds at most 5 to the constant c . The weight of a bin is the sum of weights of items assigned to it. In both scenarios, bins with one A item have weight 1, bins with two C items have weight 1, and so do bins with three D items.

We remove from the sequence items of incomplete bins. Denote the amounts of B items by $n(B)$, and of E items by $n(E)$. The number of red E items is denoted $n(ER)$, and the number of white E items $n(EW)$, (i.e., $n(E) = n(EW) + n(ER)$). According to the color assignments, and since at most two white items and one red item were removed, $3n(ER) \leq n(EW) \leq 3n(ER) + 6$. In the first scenario, no bins contain red E items only. The total weight of B and E items is $n(B) + \frac{n(E)}{4}$. The number of bins used for these types is $n(B) + \frac{n(EW)}{3} \leq n(B) + \frac{n(E)+2}{4}$ (using $n(EW) \geq 3n(ER) + 6$). In this case we get $ALG < X_1 + 6$. In the second scenario, no bins contain a B item only. The total weight of B and E items is $\frac{3n(E)}{8}$. The number of bins used for these types is $\frac{n(ER)}{2} + \frac{n(EW)}{3} \leq \frac{3n(E)}{8}$ (using $3n(ER) \leq n(EW)$). In this case we get $ALG < X_2 + 5$.

Next we analyze the maximum amount of weight that a bin can contain according to the two weight measures. In both weight measures, if no item has weight 1, the total weight of three items does not exceed $\frac{3}{2}$. Using w_1 , the smallest item of weight 1 is slightly larger than $\frac{1}{2}$. If there is a C item, then there can be no D item but only a E item. We get therefore $1 + \frac{1}{2} + \frac{1}{4}$. If there is no C item, the worst case is two extra D items. This gives $1 + \frac{2}{3}$. We get therefore $W_1 = \frac{7}{4} = 1.75$. Using w_2 , the smallest item of weight 1 is slightly larger than $\frac{2}{3}$. There can be no B or C items. The worst case is two extra E items, and we get $W_2 = 1 + 2 \cdot \frac{3}{8} = 1.75$.

We proved the following theorem.

Theorem 10. *The competitive ratio of the above algorithm for $k = 3$ is at most 1.75.*

In the full version of the paper, we design the algorithms for other small values of k and prove the following theorem.

Theorem 11. *The competitive ratios of the above algorithm for $k = 4, 5, 6$ are at most $\frac{71}{38} \approx 1.86842$, $\frac{771}{398} \approx 1.93719$, $\frac{287}{144} \approx 1.99306$ (respectively).*

References

1. L. Babel, B. Chen, H. Kellerer, and V. Kotov. Algorithms for on-line bin-packing problems with cardinality constraints. *Discrete Applied Mathematics*, 143(1-3):238–251, 2004.
2. A. Caprara, H. Kellerer, and U. Pferschy. Approximation schemes for ordered vector packing problems. *Naval Research Logistics*, 92:58–69, 2003.

3. E. G. Coffman, M. R. Garey, and D. S. Johnson. Approximation algorithms for bin packing: A survey. In D. Hochbaum, editor, *Approximation algorithms*. PWS Publishing Company, 1997.
4. J. Csirik. An online algorithm for variable-sized bin packing. *Acta Informatica*, 26:697–709, 1989.
5. J. Csirik and G. J. Woeginger. On-line packing and covering problems. In A. Fiat and G. J. Woeginger, editors, *Online Algorithms: The State of the Art*, pages 147–177, 1998.
6. J. Csirik and G. J. Woeginger. Resource augmentation for online bounded space bin packing. *Journal of Algorithms*, 44(2):308–320, 2002.
7. L. Epstein and R. van Stee. On variable-sized multidimensional packing. In *Proc. of the 12th Annual European Symposium on Algorithms (ESA2004)*, pages 287–298, 2004.
8. L. Epstein and R. van Stee. Online bin packing with resource augmentation. In *Proceedings of the 2nd Workshop on Approximation and Online Algorithms (WAOA 2004)*, pages 48–60, 2004.
9. L. Epstein and R. van Stee. Optimal online bounded space multidimensional packing. In *Proc. of 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'04)*, pages 207–216, 2004.
10. D. K. Friesen and M. A. Langston. Variable sized bin packing. *SIAM Journal on Computing*, 15:222–230, 1986.
11. H. Kellerer and U. Pferschy. Cardinality constrained bin-packing problems. *Annals of Operations Research*, 92:335–348, 1999.
12. K. L. Krause, V. Y. Shen, and H. D. Schwetman. Analysis of several task-scheduling algorithms for a model of multiprogramming computer systems. *Journal of the ACM*, 22(4):522–550, 1975.
13. K. L. Krause, V. Y. Shen, and H. D. Schwetman. Errata: “Analysis of several task-scheduling algorithms for a model of multiprogramming computer systems”. *Journal of the ACM*, 24(3):527–527, 1977.
14. C. C. Lee and D. T. Lee. A simple online bin packing algorithm. *Journal of the ACM*, 32(3):562–572, 1985.
15. P. Ramanan, D. J. Brown, C. C. Lee, and D. T. Lee. Online bin packing in linear time. *Journal of Algorithms*, 10:305–326, 1989.
16. S. S. Seiden. An optimal online algorithm for bounded space variable-sized bin packing. *SIAM Journal on Discrete Mathematics*, 14(4):458–470, 2001.
17. S. S. Seiden. On the online bin packing problem. *Journal of the ACM*, 49(5):640–671, 2002.
18. S. S. Seiden, R. van Stee, and L. Epstein. New bounds for variable-sized online bin packing. *SIAM Journal on Computing*, 32(2):455–469, 2003.
19. J. D. Ullman. The performance of a memory allocation algorithm. Technical Report 100, Princeton University, Princeton, NJ, 1971.
20. A. van Vliet. An improved lower bound for online bin packing algorithms. *Information Processing Letters*, 43(5):277–284, 1992.
21. G. J. Woeginger. Improved space for bounded-space online bin packing. *SIAM Journal on Discrete Mathematics*, 6:575–581, 1993.
22. A. C. C. Yao. New algorithms for bin packing. *Journal of the ACM*, 27:207–227, 1980.

Fast Monotone 3-Approximation Algorithm for Scheduling Related Machines

Annamária Kovács

Max-Planck Institut für Informatik,
Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany
fax: +49-681-93-25-199
panni@mpi-inf.mpg.de

Abstract. We consider the problem of scheduling n jobs to m machines of different speeds s.t. the makespan is minimized ($Q||C_{\max}$). We provide a fast and simple, deterministic *monotone* 3-approximation algorithm for $Q||C_{\max}$. Monotonicity is relevant in the context of *truthful mechanisms*: when each machine speed is only known to the machine itself, we need to motivate that machines declare their true speeds to the scheduling mechanism. As shown by Archer and Tardos, such motivation is possible only if the scheduling algorithm used by the mechanism is monotone. The best previous monotone algorithm that is polynomial in m , was a 5-approximation by Andelman et al. A *randomized* 2-approximation method, satisfying a weaker definition of truthfulness, is given by Archer. As a core result, we prove the conjecture of Auletta et al., that the greedy algorithm (LPT) is monotone if machine speeds are all integer powers of 2.

1 Introduction

We consider the offline task scheduling problem on related machines ($Q||C_{\max}$). We are given a *speed vector* $\langle s_1, s_2, \dots, s_m \rangle$ representing the speeds of m machines, and a *job vector* $\langle t_1, t_2, \dots, t_n \rangle$, where t_j is the *size* of the j th job, $1 \leq j \leq n$. We assume $t_j \geq t_{j+1}$ ($1 \leq j < n$), i.e., the job sizes are ordered. The goal is to assign the jobs to the machines, so that the overall finish time is minimized: If jobs assigned to machine i are $\{t_\gamma^i\}_{\gamma=1}^\Gamma$ then the *work* assigned to i is $w_i := \sum_{\gamma=1}^\Gamma t_\gamma^i$ and the *finish time* of i is $f_i := w_i/s_i$. The *makespan* to be minimized is $\max_{i=1}^m f_i$. This problem is NP-hard even for 2 identical machines [6], but it has an approximation scheme [3]. For constant m a FPTAS exists [4].

Here we regard this problem in the context of *truthful mechanisms*: we assume that the machines are owned by selfish agents, and the speed of each machine is private information to its agent. A *mechanism* is a pair $\mathcal{M} = (A, P)$, where A is an (approximation-)algorithm to the scheduling problem, and P is a payment function. Let the job vector be fixed and public. Every machine (agent) i reports a *bid* b_i to be the inverse of her speed. With the job vector and the bid vector $\langle b_1, \dots, b_m \rangle$ as input, \mathcal{M} schedules the jobs using algorithm A , and pays each machine using the payment function $P = (P_1, P_2, \dots, P_m)$. P depends on the schedule and on the bids. The *profit* of machine i is defined by $P_i - w_i/s_i$.

The mechanism is *truthful*, if (for every job vector), for every i and every s_i , and every possible bid vector of the other machines $\langle b_1, \dots, b_{i-1}, b_{i+1}, \dots, b_m \rangle$, bidding truthfully, i.e., $b_i = 1/s_i$ maximizes the profit of i .

Applications of the *mechanism design* framework to diverse optimization problems arising in economics, game theory, and recently in computer science and networking constitute a widely studied area (see [2,7]). The above formulation concerning scheduling is one of the numerous examples. It is a natural goal to search for (truthful) mechanisms with an efficient algorithm A having a good approximation bound and an efficiently computable payment function P .

In our scheduling context, a *monotone* algorithm A is defined as follows:

Definition 1. Let A be an approximation algorithm for the $Q||C_{\max}$ problem. Suppose that on input $\langle s_1, \dots, s_{k-1}, s_k, s_{k+1}, \dots, s_m \rangle$ and $\langle t_1, t_2, \dots, t_n \rangle$, A assigns work w_k to machine k ; and on input $\langle s_1, \dots, s_{k-1}, s'_k, s_{k+1}, \dots, s_m \rangle$ and $\langle t_1, t_2, \dots, t_n \rangle$, A assigns work w'_k to machine k . The algorithm A is monotone, if $(s_k \leq s'_k) \Rightarrow (w_k \leq w'_k)$ ($1 \leq k \leq m$).

Related Work. In a seminal paper Archer and Tardos [2] show that in models when the profit function has the above form $P_i - w_i/s_i$, a truthful mechanism $\mathcal{M} = (A, P)$ exists if and only if A is a monotone algorithm. In this case they also provide an explicit formula for the payment function. Among other examples, Archer and Tardos consider the problem $Q||C_{\max}$. They show that – by some fixed order of the machines – the lexicographically minimal optimal solution is monotone. They also provide a fast *randomized* 3-approximation algorithm, that allows a mechanism, that is *truthful in expectation*, meaning that truth-telling maximizes the *expected* profit of each agent. This was later improved to a randomized 2-approximation mechanism [1].

The same problem, i.e., finding an efficient monotone approximation algorithm for $Q||C_{\max}$ is studied in [9]. The authors provide a deterministic, monotone $(4 + \epsilon)$ -approximation algorithm. They conjecture that the greedy list-processing algorithm 'Longest Processing Time first (LPT)' [10] is monotone, if machine speeds are known to be powers of 2 (*2-divisible* speeds). They apply a *variant* of LPT, which is combined with the optimal schedule of the largest jobs, in order to give a reasonable approximation bound. As a consequence, the resulting algorithm is only polynomial when m is fixed, in particular, it runs in time $\Omega(\exp(m^2/\epsilon))$. This result was considerably improved by Andelman et al. in a recent paper [7], presenting an FPTAS for the case of constant m , and a 5-approximation algorithm for arbitrary m .

Our Result. We show a fast and simple, deterministic *monotone* 3-approximation algorithm for $Q||C_{\max}$. With input jobs ordered by size, it runs in time $\mathcal{O}(m(n + \log m))$. This is an improvement over the 5-approximation bound of [7]. As compared to the algorithms in [2,1], no randomization is needed, and a stronger definition of truthfulness is fulfilled. Our approach can be sketched as follows:

We prove the conjecture of [9], that LPT is monotone on 2-divisible machines. In case of arbitrary machine speeds LPT was shown to yield a

2-approximation [10]. We show that in case of 2-divisible speeds, LPT is a 3/2-approximation algorithm. For arbitrary input speeds, the monotone 3-approximation algorithm LPT* is as simple as to run LPT with machine speeds rounded to powers of 2. In order to show monotonicity of LPT, we need to compare schedules I and II, where both are results of LPT on the same input, except that the machine speed s_k of I is increased to $s'_k = 2s_k$ in II. The proof involves a rather technical case distinction based on the number of jobs assigned to the machine k' in II, and on the ratio t_n/t_a , where t_a is the first job assigned to k' . If t_n is not much smaller than t_a , then a simple counting of the jobs assigned to each machine yields the proof. If $t_n \ll t_a$, then a comparison of the total work received by each machine in I and II is used.

For completeness, we note here that along the same lines as in [9,7], it is straightforward to show that our payment function admits *voluntary participation* (see [2]) and can be calculated in polynomial time.

The argument that LPT gives a 3/2-approximation, is relatively short and simple, while making use of the same basic idea as the much more involved proof of monotonicity. Therefore, we present this argument in the first place.

Overview. Section 2 introduces notation, and defines algorithms LPT and LPT*. In Section 3 we show that LPT* is a 3-approximation algorithm. A sketch of the proof that LPT* is monotone can be found in Section 4. The complete proof is available at [5]. We conclude with some considerations about approximation lower bounds.

2 The LPT* Algorithm

In order to distinguish arbitrary input speed vectors from 2-divisible speed vectors, in the rest of the paper $\langle \sigma_1, \dots, \sigma_i, \dots, \sigma_m \rangle$ denotes the input speed vector of arbitrary, positive speeds. Moreover we will assume that $\sigma_i \leq \sigma_{i+1}$ ($1 \leq i < m$), i.e., machine speeds are in non-decreasing order. Next we define 2-divisible speed vectors. In the definition we allow fractional speeds only for sake of simpler presentation of our proofs. Clearly, they are not essential to the result.

Definition 2. *The speed vector $\langle s_1, s_2, \dots, s_m \rangle$, or the machines are called 2-divisible if $s_i = 2^{l_i}$ ($l_i \in \mathbb{Z}$) for all i , and $s_i \leq s_{i+1}$ ($1 \leq i < m$).*

We say that machine h is to the right (left) of i if $i < h$ ($h < i$). We will refer to job j by the job size t_j . In cases when the work w_i or finish time f_i of machine i is considered at an intermediate step of the algorithm, this is emphasized by a \sim or some superscript over w_i and f_i . The *completion time* of a job t_j assigned to machine i is the finish time of i right after t_j was scheduled. Next, we define algorithms LPT and LPT*:

LPT algorithm:

Input: $\langle s_1, \dots, s_m \rangle$ and $\langle t_1, \dots, t_n \rangle$

At step j of LPT let w_i^j denote the work of machine i ($1 \leq i \leq m$). LPT assigns t_j to machine h if $(w_h^j + t_j)/s_h = \min_i (w_i^j + t_j)/s_i$, and h is the smallest machine index with this property.

LPT* algorithm: Input: $\langle \sigma_1, \dots, \sigma_m \rangle$ and $\langle t_1, \dots, t_n \rangle$

1. round the machine speeds down to $s_i := 2^{\lfloor \log \sigma_i \rfloor}$ $1 \leq i \leq m$ (the rounded speeds remain ordered);
2. run LPT on $\langle s_1, \dots, s_m \rangle$ and $\langle t_1, t_2, \dots, t_n \rangle$;
3. among machines of the same rounded speed, reorder the assigned work (i.e. the assigned sets of jobs), such that $w_i \leq w_{i+1}$ holds.

Clearly, LPT* runs in time $\mathcal{O}(m(n + \log m))$.

We close this section with a frequently used property of the LPT algorithm. Observe, that Proposition 1 (i) holds only because LPT favours slower machines in case of ties. This feature of LPT is not essential, but it facilitates simpler proofs for both theorems.

Proposition 1. *Suppose that machine speeds are 2-divisible. In LPT scheduling the following hold:*

- (i) *If $s_h = s_i/2$, then h receives its first job after the first job of i and before the second job of i .*
- (ii) *Let $s_i = s_{i+1}$. If t_j is the first job assigned to i , then t_{j+1} is the first job assigned to $i + 1$.*

3 LPT* is a 3-Approximation Algorithm

The key result of this section is Theorem 1, stating that LPT yields a 3/2-approximation on 2-divisible machines. We start by introducing some notation and making elementary observations. We fix a 2-divisible speed vector $\langle s_1, s_2, \dots, s_m \rangle$ ($s_i \leq s_{i+1}$) and a job vector $\langle t_1, t_2, \dots, t_n \rangle$ ($t_j \geq t_{j+1}$).

Let OPT be any fixed optimal schedule of this input. Opt denotes the optimum makespan and Lpt denotes the makespan resulted by LPT on the above input. We will also use the short notation $\mu := Opt$.

To get a contradiction, we assume that $Lpt > \frac{3}{2}\mu$. Let the last job $t := t_n$ be assigned to machine k in LPT. We may assume that t has completion time Lpt , i.e., t is a bottleneck job. Note that jobs following a bottleneck job neither increase Lpt , nor decrease Opt .

We denote by w_i^* , and $f_i^* = w_i^*/s_i$ the work and the finish time of machine i in OPT with respect to all the jobs. We denote by w_i , and $f_i = w_i/s_i$ the work and the finish time of machine i in LPT before scheduling t . Note that the last restriction has an influence only on f_k and w_k .

$Lpt > \frac{3}{2}\mu$ implies that for every machine i in LPT

$$f_i > \frac{3}{2}\mu - t/s_i. \tag{1}$$

We assume wlog. that the slowest nonempty machine in OPT has speed 1. It follows that

$$t \leq \mu \cdot 1 = \mu. \tag{2}$$

Definition 3. *A machine i of speed $s_i = 1$ will be called a 1-machine.*

The proof is based on the following simple technique: We strive to get a contradiction by showing that the total work in LPT is strictly more than in OPT. First we show that only 1-machines may get more work in OPT than in LPT. Then in Lemma 1 we introduce the set of \mathcal{P} -jobs. These jobs are assigned to 1-machines in OPT, but they are larger than the jobs on 1-machines in LPT. Finally, in Lemma 2 we argue that the total work difference ($\sum_{s_i > 1} (w_i - w_i^*)$) on faster machines receiving the \mathcal{P} -jobs in LPT, exceeds the potential work difference on 1-machines ($\sum_{s_i = 1} (w_i^* - w_i)$), so that in total, more work is scheduled in LPT than in OPT.

Proposition 2. *If $f_i < f_i^*$ holds for a machine i , then i is a 1-machine, and in LPT at most 1 job is assigned to i . Moreover, there exists at least one 1-machine l s.t. $f_l < f_l^*$.*

Proof. For such a machine $f_i < f_i^* \leq \mu$ holds. According to (1), $\frac{3}{2}\mu - t/s_i < f_i < \mu$, i.e., $\mu/2 < t/s_i$. By (2), $t/s_i \leq \mu/s_i$, so we obtain $s_i < 2$. If $s_i < 1$ then $f_i^* = 0$. Consequently, $s_i = 1$ and $\mu/2 < t/s_i$ implies $\mu/2 < t$, so there is at most 1 job on i . Since $t + \sum_i w_i = \sum_i w_i^*$, a machine l exists, s.t. $w_l < w_l^*$, i.e., $f_l < f_l^*$. \square

Corollary 1. $\mu/2 < t$

By Proposition 2, $f_i < f_i^*$, resp. $w_i < w_i^*$ is possible only on 1-machines. In Lemma 1 we upper bound $\sum_{s_i = 1} (w_i^* - w_i)$ (see Fig. 1). Let $f_o := \max(t, \frac{3}{4}\mu)$. It is easy to show the following Proposition:

Proposition 3. *On an arbitrary 1-machine i , there is at most 1 job in OPT, and even before scheduling t , there is at least 1 job in LPT. Moreover, $f_i \geq f_o$.*

Lemma 1. *For some $p \geq 0$, there is a set of p jobs $\mathcal{P} = \{t_{j_1}, \dots, t_{j_p}\}$ so that*

- (i) *all of the jobs in \mathcal{P} are assigned to 1-machines in OPT, and to faster machines in LPT;*

- (ii) $\frac{3}{4}\mu \leq f_o \leq t_{j_\tau} \leq \mu \quad (1 \leq \tau \leq p)$;
- (iii) $\sum_{s_i = 1} (w_i^* - w_i) \leq \sum_{\tau = 1}^p (t_{j_\tau} - f_o) \leq p \cdot \mu/4$.

Definition 4. *The jobs in \mathcal{P} will be called \mathcal{P} -jobs.*

Lemma 2. *In LPT, at most $2^r - 1$ \mathcal{P} -jobs are assigned to a machine i of speed $s_i = 2^r$ ($r \geq 1$). If $s_i \geq 4$ then $w_i - w_i^* \geq (2^r - 1) \cdot \mu/4$. If $s_i = 2$ and \hat{t} is the (only) \mathcal{P} -job assigned to i , then $w_i - w_i^* \geq \hat{t} - f_o$.*

Proof. While \mathcal{P} -jobs are being scheduled, 1-machines are still empty in LPT. Therefore, any \mathcal{P} -job t_{j_τ} has completion time less than t_{j_τ} , otherwise it would be assigned to a 1-machine. This implies that a machine of speed 2^r has at most $2^r - 1$ \mathcal{P} -jobs.

Suppose first that $s_i = 2^r \geq 4$. Then by (1) and (2), $f_i - f_i^* > \frac{3}{2}\mu - t/4 - \mu = \frac{1}{2}\mu - t/4 \geq \frac{1}{2}\mu - \mu/4 = \mu/4$. Consequently, $w_i - w_i^* > 2^r \cdot \mu/4$.

Second, if $s_i = 2$, then (1) implies $w_i - w_i^* > 2 \cdot (\frac{3}{2}\mu - t/2 - \mu) = \mu - t$, and $\mu - t \geq \hat{t} - f_o$ by Lemma 1 and Proposition 3. \square

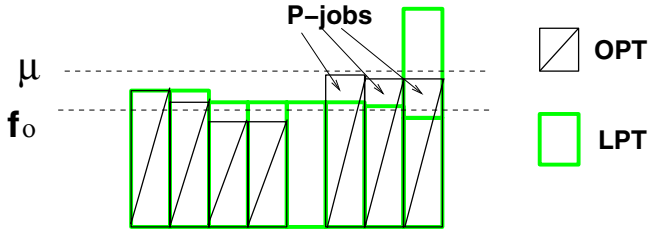


Fig. 1. Lemma 1: Jobs on 1-machines in LPT and in OPT

Theorem 1. Let $\langle s_1, s_2, \dots, s_m \rangle$ ($s_i \leq s_{i+1}$) be a 2-divisible speed vector, and $\langle t_1, t_2, \dots, t_n \rangle$ ($t_j \geq t_{j+1}$) a fixed job vector. Let Opt be the optimum makespan, and Lpt be the makespan resulted by LPT on this input. Then $Lpt \leq \frac{3}{2} \cdot Opt$.

Proof. If $s_i \neq 1$ then $w_i \geq w_i^*$ by Proposition 2. Therefore, Lemmas 1 and 2 imply

$$\sum_{s_i=1} (w_i^* - w_i) \leq \sum_{\tau=1}^p (t_{j_\tau} - f_0) \leq \sum_{s_i \neq 1} (w_i - w_i^*)$$

$$\sum_{i=1}^m w_i^* \leq \sum_{i=1}^m w_i$$

a contradiction, since $t + \sum_i w_i = \sum_i w_i^*$. □

Theorem 2. LPT* is a 3-approximation algorithm.

Proof. Suppose that on input $\langle t_1, t_2, \dots, t_n \rangle$

Opt is the optimum makespan at speed vector $\langle \sigma_1, \dots, \sigma_m \rangle$;

Opt' is the optimum makespan at speed vector $\langle s_1, \dots, s_m \rangle$;

Lpt is the makespan provided by LPT at $\langle s_1, \dots, s_m \rangle$;

Lpt^* is the makespan provided by LPT* at $\langle \sigma_1, \dots, \sigma_m \rangle$,

then $Lpt^* \leq Lpt \leq \frac{3}{2} \cdot Opt' \leq \frac{3}{2} \cdot (2 \cdot Opt)$. The first and last inequalities follow from the fact that machine speeds are increased, resp. decreased by a factor between 1 and 2. Finally, $Lpt \leq \frac{3}{2} \cdot Opt'$ holds according to Theorem 1. □

4 LPT* is Monotone

The main result of this paper is that the LPT schedule is monotone on 2-divisible machines. In particular, suppose that in LPT schedule I the 2-divisible input speed vector contains one more copies of speed 1/2 and one less copies of speed 1 than in LPT schedule II, and otherwise the inputs of I and II are the same. Let k be any machine of speed 1/2 in I, and k' be any machine of speed 1 in II. Theorem 3 claims that machine k receives not more work in schedule I than machine k' in schedule II.

Let $s_1 \leq s_2 \leq \dots \leq s_m$ denote the machine speeds in I. We will view schedule II like this: in schedule II, the speed $s_k = 1/2$ of machine k is increased to $s'_k = 1$ while the speeds of other machines remain unchanged. We will refer to machine i in schedule II by i' . Clearly, in general k' is not the k th machine, and i' is not necessarily the i th machine in II (see Fig. 3). We partition the unchanged machines into three categories:

Definition 5. Let $i \neq k$. Based on the speed s_i of machine i , we say that i is a slow machine if $s_i < 1/2$, a medium machine if $1/2 \leq s_i \leq 1$, resp. a fast machine, if $1 < s_i$. We call a slow machine tardy, if it has the speed of the slowest nonempty machines in II. Finally, a machine of speed 1 or $1/2$ will be called a 1-machine, resp. a $1/2$ -machine.

In I and in II the machines receive the same job sequence $t_1 \geq \dots \geq t_n$. If t_j is assigned to machine i , it has (time)length t_j/s_i . We denote by w_i and f_i the work and finish time of machine i in schedule I. For ease of use, w'_i and f'_i denote the respective values in schedule II (instead of, e.g., $w'_{i'}$ and $f'_{i'}$).

Theorem 3. Let the LPT schedules I and II, machines k and k' , furthermore the respective total works w_k and w'_k be as defined above, then $w_k \leq w'_k$.

Sketch of Proof. We prove Theorem 3 by contradiction: we assume $w_k > w'_k$. Let $W_{\neq k} := \sum_{i \neq k} w_i$ and $W'_{\neq k} := \sum_{i \neq k} w'_i$. In most subcases of the proof we strive to show $W_{\neq k} \geq W'_{\neq k}$, contradicting to $w_k > w'_k$. In the remaining subcases we show that the number of assigned jobs is strictly larger in schedule I than in schedule II.

Let t_a be the first job assigned to k' in II. We will call a machine *dead*, if it receives no job after t_a in II, and we call it *living* otherwise. Note that right before job t_a is scheduled, the schedules I and II are exactly the same. Therefore, on dead machines $w \geq w'$, and there are at least as many jobs on the machine in I as in II. Unless it is necessary to mention dead machines explicitly, we concentrate on living machines.

Let $t := t_n$ denote the last job. We may suppose that w_k becomes larger than w'_k only after job t ; job t is assigned to k in I, but it is not assigned to k' in II.

Let $t_a = t_{a_1} \geq t_{a_2} \geq t_{a_3} \geq \dots$ be the jobs assigned to k' . It facilitates a more handy proof if we normalize job sizes so that $t_a = 1$. We can do this without loss of generality. Consequently, the length of t_a on k' is $t_a/s'_k = 1/1 = 1$.

Because the finish time f'_k of k' plays a central role in our comparisons, we provide it with special notation: let $\lambda := f'_k$ be the finish time of k' . That is, $\lambda = w'_k/1$. Since $w_k > w'_k$, for the finish time of k in I, $f_k = w_k/1/2 > w'_k/1/2 = 2\lambda$ holds. Moreover, in I a machine $i \neq k$ of speed 2^r ($r \in \mathbb{Z}$) has finish time

$$f_i > 2\lambda - \frac{t}{2^r}, \tag{3}$$

otherwise this machine (and not k) would receive the last job t .

Let W_T and W'_T denote the total work on tardy machines in schedule I and II, respectively. We provide some intuition about the first part of our proof.

This part follows the same lines as the proof of Theorem 1. First we show, that if $w < w'$ on a slow machine, then it must be a tardy machine. After that we prove that *if $w \geq w'$ on every medium and fast machine, then $W_{\neq k} \geq W'_{\neq k}$* . Luckily, this is the case if at least 2 jobs are assigned to k' in II (CASE 1.). How do we show $W_{\neq k} \geq W'_{\neq k}$? In Lemma 3 we introduce the set of \mathcal{P} -jobs. These are very small jobs, that follow the jobs of tardy machines in II, but are still put on tardy machines in I. In the same lemma an upper bound on $W'_T - W_T$ is derived in terms of the number p of \mathcal{P} -jobs. In Lemma 4 it is shown that the difference $W'_T - W_T$ is balanced out on non-tardy machines, that receive the \mathcal{P} -jobs in II. In CASE 2., the same argument about \mathcal{P} -jobs is used parallel to other techniques, in order to show $W_{\neq k} \geq W'_{\neq k}$.

Next, we provide upper bounds on the finish time of living machines in II:

Proposition 4. *In schedule II, $f'_i \leq \max(2, 3\lambda/2)$ for any living machine i' .*

Proposition 5. *In LPT schedule II,*

- (i) *If t_j is a job on a slow machine i' , then $t \leq t_j \leq \lambda/3$, and $f'_i \leq 4\lambda/3$.*
 - (ii) *If t_j is the 2nd job on a 1/2-machine i' then $t \leq t_j \leq \lambda/3$, and $f'_i \leq 4\lambda/3$.*
 - (iii) *If t_j is the 3rd job on a 1-machine i' then $t \leq t_j \leq \lambda/2$, and $f'_i \leq 3\lambda/2$.*
- In cases (ii) and (iii) $f'_i \leq f_i$.*

Proposition 6. *Suppose that $w_i < w'_i$ holds for a slow machine i of speed $s_i = 1/2^l$ ($l \geq 2$). Then i is a tardy machine, and each tardy machine receives at most 1 job in schedule II.*

In Lemma 3 we upper bound $W'_T - W_T$. The jobs of \mathcal{P} will be called \mathcal{P} -jobs:

Lemma 3. *Suppose that $w_i < w'_i$ holds for a tardy machine i , and tardy machines have speed $1/2^d$. Let t_j be the first job on tardy machines in II. There is a set of jobs $\mathcal{P} = \{t_J, t_{J+1}, \dots, t_{J+p-1}\}$ for some $p \geq 0$, so that*

- (i) *in I all the p jobs are assigned to tardy machines, and in II all the p jobs are assigned to faster than tardy machines;*
- (ii) $\lambda/2^d \leq t_{J+\zeta} < t_j \leq \frac{4}{3}\lambda/2^d$ ($0 \leq \zeta \leq p - 1$);
- (iii) $W'_T - W_T \leq p \cdot (\frac{4}{3}\lambda/2^d - t) \leq p \cdot (\frac{1}{3}\lambda - t)$.

Lemma 4. *Machine k' does not receive \mathcal{P} -jobs in II. Suppose that at least one \mathcal{P} -job is assigned to machine $i' \neq k'$.*

- (i) *If i' is a fast machine of speed $s_i = 2^r$ ($r \geq 1$), then it receives at most 2^r \mathcal{P} -jobs and $w_i - w'_i > 2^r(\frac{1}{3}\lambda - t)$. Furthermore, $w'_i \leq 2^r \frac{4}{3}\lambda$.*
- (ii) *If i' is a medium or slow (non-tardy) machine, then it receives 1 \mathcal{P} -job, and $w_i - w'_i > \frac{4}{3}\lambda/2^d - t$, where the speed of tardy machines is $1/2^d$.*

Corollary 2. *If $w_i \geq w'_i$ for every medium and fast machine, then $W_{\neq k} \geq W'_{\neq k}$.*

Proof. By Proposition 6, $w_i \geq w'_i$ on every non-tardy machine. There are $p \geq 0$ \mathcal{P} -jobs on tardy machines in I. Let $i'_1, i'_2, \dots, i'_\xi$ be the machines with at least one \mathcal{P} -job in II. By Lemmas 3 and 4,

$$W'_T - W_T \leq p \cdot (\frac{4}{3}\lambda/2^d - t) \leq \sum_{\tau=1}^{\xi} (w_{i'_\tau} - w'_{i'_\tau}).$$

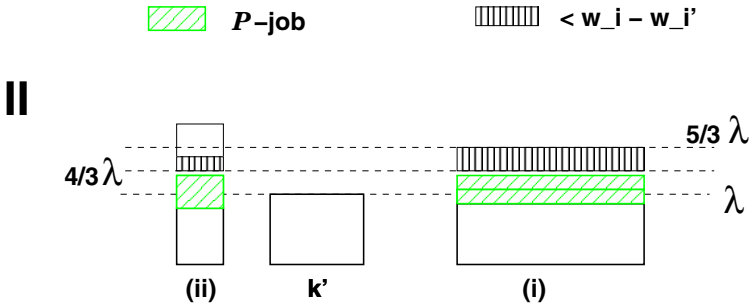


Fig. 2. Lemma 4: A fast (i) and a medium (ii) machine receiving \mathcal{P} -jobs in schedule II

To sum up, the potential total difference $W'_T - W_T$ on tardy machines is balanced out on non-tardy machines receiving \mathcal{P} -jobs in II, and $W_{\neq k} \geq W'_{\neq k}$ follows. \square

CASE 1. in schedule II at least 2 jobs are assigned to machine k'

Lemma 5. *If there are at least 2 jobs assigned to k' , then $W_{\neq k} \geq W'_{\neq k}$.*

Proof. We show that if i is a fast or medium machine then $f_i \geq f'_i$, that is, $w_i \geq w'_i$. Based on this, Corollary 2 yields the Lemma.

(i) We show that $f_i > 2$ on every machine i of speed at least $1/2$ in I. Since t is the last job, $w'_k \geq t_{a_1} + t_{a_2} \geq 1 + t$. In schedule I, let \tilde{w}_k and \tilde{f}_k be the work and finish time of k before the last step. Then $\tilde{w}_k = w_k - t > w'_k - t \geq 1$, consequently, $\tilde{f}_k > 2$. If there were a machine i of $f_i \leq 2$ and $s_i \geq 1/2$, then i would receive t instead of machine k .

(ii) We show that $f_i > 3\lambda/2$ on every machine of speed at least 1 in I. Note that $t \leq \lambda/2$ holds, because $\lambda = w'_k \geq 1 + t$. Moreover, (3) implies $f_i > 2\lambda - t/2^r \geq 2\lambda - t \geq 3\lambda/2$ if $r \geq 0$.

Now (i), (ii) and Proposition 4 imply the statement of the Lemma, unless i has speed $1/2$. Let $s_i = 1/2$. If there is only one job t_j assigned to i' in II, then $t_j \leq t_a = 1$, since $s_i < s'_k$, and t_a is the first job on k' . Consequently, $f'_i \leq 2$, which together with (i) yields $f_i \geq f'_i$. If there are at least two jobs assigned to i' , then according to Proposition 5, $f_i \geq f'_i$. \square

CASE 2. in schedule II only job t_a is assigned to machine k'

The proof of CASE 2. is more involved, and consists of further subcases. In the general part we introduce further notation and derive necessary conditions for $w < w'$ on a medium machine. As a side effect, this will prove the theorem if there are no fast machines. Recall that $t_a = 1$, so $\lambda = f'_k = w'_k/1 = t_a/1 = 1$.

In CASE 2.1. we assume $t > 1/3$. Thus, Proposition 5 (i) implies that all the slow machines are empty in II. Moreover, using the fact that before t_a is scheduled, every machine has the same finish time in I as in II, we will show that if $i \neq k$, then the number of jobs assigned to i by I is not smaller than the number of jobs assigned to i' by II; and schedule I assigns strictly more jobs to machine k , than schedule II to k' , so we get a contradiction.

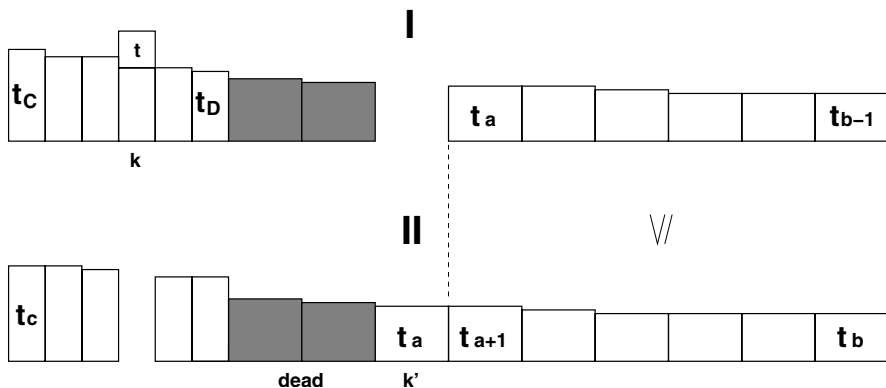


Fig. 3. Definition 6: The (first) jobs on 1/2-machines and 1-machines in I and II

The case when $t \leq 1/3$ (CASE 2.2.) is divided into three subcases, depending on the size of the smallest job assigned to any fast machine. Although this part is lengthy and intricate, it applies combinations of the two types of arguments we are presenting here, i.e., a comparison of the total work or of the number of jobs on fast and medium machines. The proof of CASE 2.2. is available at [5].

Definition 6. Let t_a, t_{a+1}, \dots, t_b ($a \leq b$) be the first jobs assigned to 1-machines to the right of k' in II. Let T_b be the time step, before t_b is scheduled.

Let t_C be the first job on the leftmost 1/2-machine, and t_D be the first job on the rightmost 1/2-machine in I. Finally, t_c denotes the first job assigned to medium machines after $T_b + 1$ in II, if such a job exists (see Figure 3).

By Proposition 1 the jobs t_{a+1}, \dots, t_b are well-defined. At time T_b schedules I and II are the same, except that jobs on 1-machines are shifted due to machine k' ; 1/2-machines and slow machines are empty. Observe that $C \leq D$, and if 1/2-machines exist in II, then t_c is the first job on the leftmost 1/2-machine. If job t_c does not exist at all, then obviously $w \geq w'$ holds for all medium machines.

Proposition 7. In schedule I, there are at least 3 jobs and total work $> 1 + t_D$ assigned to each living 1-machine.

Proposition 8. If $w_i < w'_i$ for a medium machine i , then $c < C$ and $t_c > 1 - t$.

Corollary 3. If there are no fast machines, then $W_{\neq k} \geq W'_{\neq k}$.

Proof. Recall that in II there is at least one 1-machine, and in I there is at least one 1/2-machine. If there are no fast machines, then according to Proposition 1, $b = C$ must hold (see Fig. 3). Therefore, $c > b = C$, and Proposition 8 implies $w_i \geq w'_i$ on every medium machine i . Finally, Corollary 2 yields the proof. \square

CASE 2.1. $t > 1/3$

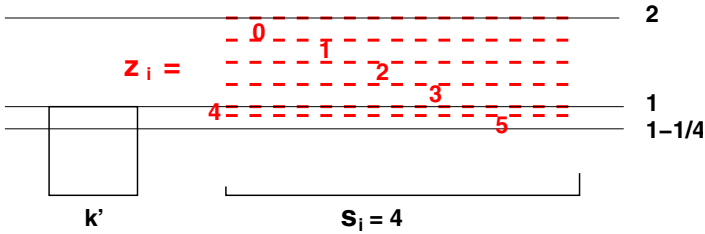


Fig. 4. Definition 8: Zones of a machine of speed 4

Definition 7. Let i be a fast machine. f_i^b denotes the common finish time of i and i' at time T_b .

Note that if $s_i = 2^r$ ($r \geq 1$), then $f_i^b \geq 1 - 1/2^r$, otherwise t_a would be assigned to i' . For the (final) finish time in II, $f_i' \leq 2$ holds by Proposition 4. Now we first partition the time interval $(1, 2]$ into 2^r equal zones of length $1/2^r$, and partition $(1 - 1/2^r, 1]$ into two further zones. (see Figure 4):

Definition 8. We say that f_i^b is in the z_i th zone if $z_i \in \mathbb{N}$ s.t.

$$f_i^b \in (2 - (z_i + 1) \cdot 1/2^r, 2 - z_i \cdot 1/2^r] \quad (0 \leq z_i \leq 2^r - 1);$$

if $f_i^b \in (1 - t/2^r, 1]$, then $z_i = 2^r$; if $f_i^b \in [1 - 1/2^r, 1 - t/2^r]$, then $z_i = 2^r + 1$.

Proposition 9. If f_i^b is in the z_i th zone, then after T_b , i gets at least z_i jobs in I, and i' gets at most z_i jobs in II.

Lemma 6. If $t > 1/3$, then i receives at least as many jobs in schedule I as i' in schedule II; k receives at least 2 jobs in I, and k' receives only 1 job in II.

Proof. (i) To machine k , schedule I assigns at least 2 jobs, otherwise $w_k > w'_k$ is impossible; on the other hand, II assigns only job t_a to k' .

(ii) To a $1/2$ -machine, II assigns at most 1 job by Proposition 5 (ii), since $t > \frac{1}{3} = \frac{\lambda}{3}$; schedule I assigns at least 1 job, otherwise k would not receive a second job.

(iii) To a 1-machine i , II assigns at most 3 jobs, because the completion time of the 3rd job is larger than $3 \cdot 1/3 = 1$, and any further job would prefer k' to i' . According to Proposition 7, schedule I assigns at least 3 jobs to a living 1-machine.

(iv) Proposition 9 proves the lemma for fast machines. □

□ Theorem 3

Theorem 4. LPT* is monotone.

Proof. Suppose that in the input of LPT*, the speed σ_k is increased to σ'_k and everything else remains unchanged. Then the index of speed σ'_k in the (re)ordered input speed vector is at least k . If $s_k = s'_k$, then step 3. of LPT* implies that k receives not less work with increased speed. If $s_k < s'_k$, then a repeated application of Theorem 3 implies Theorem 4. □

5 Conclusions

If m is constant, a monotone FPTAS exists for $Q||C_{\max}$ [7]. We don't know of approximation lower-bounds for efficient monotone algorithms for arbitrary m . We conjecture, that on 2-divisible machines LPT actually has a $4/3$ -approximation bound, implying that LPT* yields a $8/3$ -approximation. On the other hand, for any $\epsilon > 0$ an instance exists where LPT* provides an $8/3 - \epsilon$ approximation.

It doesn't seem to be worth trying to prove monotonicity and good approximation of LPT for c -divisible machines, where $c < 2$. It is shown in [8], that even for 2 machines, LPT is not monotone if $c \leq 1.78$.

A more promising approach might be to modify a better algorithm or a PTAS and apply it on 2-divisible machines achieving a close to 2 approximation. It is more of a challenge to provide a monotone PTAS or even just an efficient monotone algorithm with approximation bound below 2.

Acknowledgements. I would like to thank Martin Skutella for turning my attention to this problem. Special thanks to Katalin Friedl and Vincenzo Auletta for reading previous versions of this paper, and for their many useful comments.

References

1. A. Archer. *Mechanisms for Discrete Optimization with Rational Agents*. PhD thesis, Cornell University, 2004.
2. A. Archer and É. Tardos. Truthful mechanisms for one-parameter agents. In *Proc. 42nd IEEE Symp. on Found. of Comp. Sci. (FOCS)*, pages 482–491, 2001.
3. D.S. Hochbaum and D.B. Shmoys. A polynomial approximation scheme for scheduling on uniform processors: Using the dual approximation approach. *SIAM J. Comp.*, 17(3):539–551, 1988.
4. E. Horowitz and S. Sahni. Exact and approximate algorithms for scheduling non-identical processors. *Journal of the ACM*, 23:317–327, 1976.
5. A. Kovács. Fast monotone 3-approximation algorithm for scheduling related machines. Extended version: <http://www.mpi-sb.mpg.de/~panni/greedy.ps>.
6. D.S. Johnson M.R. Garey. *Computers and Intractability; A Guide to the Theory of NP-completeness*. Freeman, San Francisco, 1979.
7. Y. Azar N. Andelman and M. Sorani. Truthful approximation mechanisms for scheduling selfish related machines. In *Proc. 22nd Ann. Symp. on Theor. Aspects of Comp. Sci. (STACS)*, volume 3404 of *LNCS*, pages 69–82, 2005.
8. V. Auletta P. Ambrosio. Deterministic monotone algorithms for scheduling on related machines. In *Proc. 2nd Ws. on Approx. and Online Alg. (WAOA)*, 2004.
9. V. Auletta R. De Prisco P. Penna and G. Persiano. Deterministic truthful approximation mechanisms for scheduling related machines. In *Proc. of 21st STACS*, volume 2996 of *LNCS*, pages 608–619. Springer, 2004.
10. O.H. Ibarra T. Gonzalez and S. Sahni. Exact and approximate algorithms for scheduling nonidentical processors. *Journal of the ACM*, 23:317–327, 1976.

Engineering Planar Separator Algorithms^{*}

Martin Holzer¹, Grigorios Prasinos², Frank Schulz¹, Dorothea Wagner¹,
and Christos Zaroliagis²

¹ Department of Computer Science, University of Karlsruhe,
P.O. Box 6980, 76128 Karlsruhe, Germany
{mholzer, fschulz, dwagner}@ira.uka.de

² Computer Technology Institute, P.O. Box 1122, 26110 Patras, Greece,
and Department of Computer Engineering and Informatics,
University of Patras, 26500 Patras, Greece
{green, zaro}@ceid.upatras.gr

Abstract. We consider classical linear-time planar separator algorithms, determining for a given planar graph a small subset of the nodes whose removal separates the graph into two components of similar size. These algorithms are based upon *Planar Separator Theorems*, which guarantee separators of size $O(\sqrt{n})$ and remaining components of size less than $2n/3$. In this work, we present a comprehensive experimental study of the algorithms applied to a large variety of graphs, where the main goal is to find separators that do not only satisfy upper bounds but also possess other desirable qualities with respect to separator size and component balance. We propose the usage of *fundamental cycles*, whose size is at most twice the diameter of the graph, as planar separators: For graphs of small diameter the guaranteed bound is better than the $O(\sqrt{n})$ bounds, and it turns out that this simple strategy almost always outperforms the other algorithms, even for graphs with large diameter.

1 Introduction

The *Planar Separator Theorem* was introduced by Lipton and Tarjan in [1], where they give a linear-time algorithm for determining a set of nodes (separator) of size smaller than $2\sqrt{2n} \approx 2.83\sqrt{n}$ that separates a given planar graph with n nodes into two components of size smaller than $2n/3$. Djidjev [2] improved the bound on the separator size to $\sqrt{6n} \approx 2.45\sqrt{n}$, and also proved a lower bound of $1.55\sqrt{n}$, which is still the best known. The algorithms behind these two classical results share a common core algorithm, which determines an appropriate fundamental cycle in a planar graph that contributes to the sought separator.

Since then, a lot of generalizations and extensions have been made, and the upper bound on separator size has been improved by Alon, Seymour and Thomas [3] to $2.13\sqrt{n}$ and by Djidjev and Venkatesan [4] to the currently best known bound of $1.97\sqrt{n}$ (where the aforementioned core algorithm is used as a

^{*} This work was partially supported by the IST Programme of EC under contract no. IST-2002-001907 (DELIS).

subroutine, too). A recent work by Alexandrov et al. [5] considered a generalization of planar separator algorithms that computes t -separators: nodes have associated costs and weights, which are used in calculating the cost of the separator and the weights of the components resp.; the weight of each remaining component is required to be less than or equal to $t \cdot w(G)$, where t is an arbitrary constant in $(0, 1)$ and $w(G)$ the total weight of the graph. This typically requires the graph to be separated into more than two components, and with $t = 2/3$ and unit weight and cost includes the basic variant of the problem, as introduced above. The paper includes experimental study on a few synthetic and real-world families of graphs. For comparison, we consider some of the families used in [5] in our experiments.

We are not aware of a systematic and detailed experimental study regarding the classical algorithms by Lipton and Tarjan [1], and by Djidjev [2]. In this work, we do not only investigate finding separators that satisfy upper bounds, but we also consider several new algorithmic aspects regarding: (i) the optimization of separator size and balance; (ii) the consideration of fundamental-cycle separator algorithms in their own right; and (iii) the application of postprocessing techniques to improve the quality of the separators. The fundamental-cycle separator algorithms guarantee a bound on the separator size of $2d + 1$, where d denotes the diameter of a triangulation of the input graph. When the diameter is small, which is often the case for real-world graphs, this guarantees smaller separators than the $O(\sqrt{n})$ bounds of the classical algorithms.

Our main contribution in this work is the comprehensive experimental study of the above issues. It turned out that the behavior of the algorithms depends highly on the input graph: for example, on very regular graphs like grids a level of a breadth-first search tree (which is the first attempt of the classical algorithms) is already an almost optimal separator, whereas for more irregular and real-world graphs (e.g., road map graphs) the classical algorithms yield relatively bad solutions. Hence, for our experiments we used a large variety of planar graphs, both from real-world and synthetic inputs with different characteristics (e.g., size of diameter, size of minimum separator, etc). A surprising outcome of our experimental investigation is that fundamental-cycle separator algorithms always provide the best solutions. Another important issue of our experimental analysis concerns the arbitrary choices that have to be made during the course of an algorithm (e.g., the choice of a node as the root of a breadth-first search tree). It turns out that such choices influence the quality of the separators found significantly.

Due to lack of space several details concerning the algorithms and experiments had to be omitted in this extended abstract and can be found in [6].

2 Separating Planar Graphs

In this section, we consider classical linear-time planar separator algorithms implementing the Planar Separator Theorem as stated below. The node separators computed by the different algorithms fulfill different upper bounds $\beta\sqrt{n}$, for some constant β , on the separator size, while each of the remaining components contains less than two thirds of all nodes. The first theorem of this kind (for

$\beta = 2\sqrt{2}$) was introduced by Lipton and Tarjan [1]. For simplicity, we state the theorem and its related algorithms for the case of an unweighted planar graph. The algorithms and our implementations work for the weighted case, as it is introduced in [1], as well.

Theorem 1 (Planar Separator Theorem). *Given a planar graph G , the n nodes of G can be partitioned into three sets A , B , and S such that no edge joins a node in A with a node in B , neither A nor B consists of more than $2n/3$ nodes, and S contains no more than $\beta\sqrt{n}$ nodes, where β is a constant.*

An important concept used in the theorem are fundamental cycles: given a spanning tree of the input graph, a *fundamental cycle* consists of a non-tree edge e together with the path connecting the two end-nodes of e in the spanning tree.

Lemma 1 (Fundamental-Cycle Lemma). *Let G be a connected planar graph. Suppose G has a spanning tree of height h . Then, the nodes of G can be partitioned into three sets A , B , and C such that no edge joins a node in A with a node in B , neither A nor B consists of more than $2n/3$ nodes, and C is a fundamental cycle containing no more than $2h + 1$ nodes.*

2.1 Optimization Criteria

In practical applications of planar separator algorithms, requirements as to “good separations” may vary a lot. We therefore provide three optimization criteria: *separator size*, *balance*, and *separator ratio*. Let A be the smaller and B the larger of the two components, then *balance* is defined as A/B , and the *separator ratio* as S/A (cf. [7]). What is desirable are small separator size and high balance at the same time; the separator ratio, which is to be minimized, represents a trade-off between the two targets. Note that if one of the simple criteria, separator size or balance, is to be optimized and there are several optimal solutions, then the separator ratio criterion becomes relevant.

2.2 The Algorithms

We investigate two classical algorithms, by Lipton and Tarjan (LT) [1] and by Djidjev (Dj) [2]. Both work in three phases. First, a breadth-first search (BFS) tree is computed, partitioning the nodes into levels. If one of the BFS levels constitutes a separator fulfilling the size and balance requirements, then the algorithm returns that level. In case there are several feasible levels and an optimization criterion is applied, the respective algorithm selects a level that is optimal with respect to that criterion. In the second phase, separators consisting of two levels of the BFS tree are considered, yielding a separation of the tree into a lower, middle, and upper part of the graph. If there is a separator such that the biggest of these parts and the remaining two put together each meet the bound, it is returned. If not so, the third phase applies Lemma 1 to one part of a previous two-level separation. In this case, the separator consists of those two levels and the fundamental cycle found through the lemma. The algorithms differ in the selection of the levels, as described in more detail below. We also

consider fundamental-cycle separations computed by applying (the algorithmic version of) Lemma 1 directly to the graph.

Note that there are several parts in all the above algorithms, where certain arbitrary (in a sense “random”) decisions have to be made: (i) the choice of the BFS root and the search itself; (ii) the triangulation of the graph, which is needed in phase 3 of the algorithms; (iii) the choice of the fundamental cycle from among several feasible ones—the so-called choice of the *non-tree edge*. We will thoroughly discuss the influence of the choices of the BFS root and the non-tree edge in Section 4.

Lipton and Tarjan (LT). First, the middle level in the BFS tree is considered, i.e., the first level, starting from the root, that covers together with the lower levels more than half of the nodes. If this level is too large, the levels above and below are scanned until in each direction a level of size less than $2(\sqrt{n} - D)$ is found, where D is the distance to the middle level. If the part between these two levels is too large then Lemma 1 is used to separate it and in this case the separator consists of the two levels plus a fundamental cycle. We consider a textbook version [8,9] of the algorithm guaranteeing a separator of size less than $4\sqrt{n}$, i.e., $\beta = 4$.

Djidjev (Dj). Already in [1], Lipton and Tarjan give an even better bound, and in [2] Djidjev further improves the selection of levels to $\beta = \sqrt{6} \approx 2.45$. In a similar but more sophisticated way than that in LT, the algorithm tries to find a separator consisting of one or two levels of the BFS tree (which have to be smaller than in LT), and as final option also determines a fundamental cycle.

Fundamental-Cycle Separation (FCS). During the experimental phase of this study, we observed that it is very effective to omit the selection of levels and directly consider fundamental cycles as separators: We compute a simple-cycle separator by applying Lemma 1 directly to the input graph. The height of any spanning BFS tree is smaller than the *diameter* d of the graph, and thus, for BFS trees, the fundamental cycle C computed by Lemma 1 is a separator with no more than $2d + 1$ nodes. The minimum height of a spanning tree equals the *radius* r of the graph, and in this case the fundamental cycle can be guaranteed to contain no more than $2r + 1$ nodes. A spanning tree of height r can be computed in time $O(n^2)$ simply by computing the breadth first search trees originating from every node in the graph. Hence, FCS computes, in linear time, a separator of size no more than $2d + 1$, and, by investing quadratic time, even a separator of size $2r + 1$ can be guaranteed.

Simple-cycle separators are also promising from a theoretical point of view, since an upper bound on the separator size of $1.97\sqrt{n}$, which is (to our knowledge) the best bound in n that is currently known [4], is achieved by a simple cycle.¹ For graphs of small diameter and radius, the FCS approach guarantees

¹ The algorithm used in the proof of the $1.97\sqrt{n}$ bound is rather sophisticated, and since the simple-cycle separators obtained by FCS are (often by far) smaller than this bound for all graphs we are considering, we restrict our experiments to FCS concerning simple-cycle separators.

a better bound than the $\beta\sqrt{n}$ bounds, whereas in general, of course, the $\beta\sqrt{n}$ bounds are stronger since the maximum diameter and radius of planar graphs are linear in the number of nodes.

2.3 Postprocessing

To the above algorithms we provide two optional postprocessing steps, which may help to improve the separation found by the specific algorithm in terms of separator size and/or balance. The first one, called *node expulsion*, consists of moving separator nodes that are not connected to both components A and B (and hence do not actually separate two nodes from different components) to one of the components, thus decreasing the size of the separator. If a node can be moved to either component, then it is assigned to the smaller one. The idea behind the other postprocessing step, called the *Dulmage-Mendelsohn optimization* [10], is to detect a subset of the separator, $\emptyset \neq S' \subset S$, such that the subset $B' \subset B$, consisting of nodes that are adjacent to S' and belong to the larger component B , is smaller than S' . Then, the separator is modified by removing the nodes in S' and adding the nodes in B' . The size of the new separator is smaller than the original one, and the balance may be improved as well.

3 Data Sets

In the following, we give a brief description of the graph classes we used in our experiments. The first five categories consist of synthetically generated graphs, whilst the data sets in the last stem from real world.

The first category of *grid-like graphs* encompasses three classes of regular-structured graphs, namely `grid`, `rect(angular)`, `sixgrid`, and `triang(ular)`. The `grid` and `rect` graphs can be regarded as an $x \times x$ or $x \times y$ raster of nodes, respectively. A `sixgrid` graph is composed of $x \times y$ hexagons in a honeycomb-like fashion, and in a `triang` graph, starting with an initial triangle, every triangle is iteratively replaced by three triangles. In a `grid` graph with n nodes a separator with minimal size consists of approximately $\sqrt{2n/3} \approx 0.82\sqrt{n}$ nodes (recall that we consider only separators such that each component contains at most $2n/3$ nodes). If $x \ll y$, then the smallest separator of a `rectangular` graph has x nodes, and a `sixgrid` graph has an optimal separator with $x + 1$ nodes.

In [2] the currently best lower bound of $1.55\sqrt{n}$ on the separator size is proven by graphs that *approximate the sphere*. We consider two simple constructions of graphs that approximate the sphere, as *worst-case* examples concerning separator size. A `globe` graph is—simply speaking—the graph induced by (a specified number of) meridians and circles of latitude of a terrestrial globe. A `t-sphere` graph approximates the sphere by almost similar triangles, see e.g., [11]. The iterative generation process starts with an icosahedron; during an iteration each triangle is split into four smaller ones.

Given a diameter d , we construct a maximal planar graph that consists of $3d + 1$ nodes and has *diameter* d . We refer to this class as `diameter`. By construction, such a graph has always a separator of size 3.

Random maximum *planar* graphs, denoted by **del-max** and **leda-max**, are generated such that the specified number of nodes are randomly placed in the plane and the convex hull of them is triangulated, the triangulation being a Delaunay triangulation (**del-max**) or a standard LEDA-triangulation [12] (**leda-max**), respectively. In addition, we have the **del** and **leda** graphs, which are obtained from **del-max** and **leda-max**, respectively, by deleting at random a specified number of edges. We will occasionally refer to **del** and **del-max** (**leda** and **leda-max**, resp.) as the Delaunay (LEDA, resp.) graphs.

Graphs with *small separators* are generated as follows: Given a planar graph, two copies of this graph are connected via a given small number of additional nodes, which constitute a perfectly balanced separator of a so constructed graph. The challenge of the algorithms is to re-determine these small separators. We consider four of the previous graph types and get the following new kind of generated graphs: **c-grid**, **c-globe**, **c-del-max**, and **c-leda-max** graphs.

Regarding *real-world data*, we consider a graph representing a finite-element mesh [13] (**airfoil1**), and seven graphs representing the road networks of some U.S. cities and their surrounding areas (referred to as **city**), taken from the San Francisco Bay Area Regional Database (BARD) [14] and the Environmental Systems Research Institute (ESRI) info-page [15].

4 Experiments

Our experimental study is subdivided into three parts encompassing graphs of increasing size. The three algorithms LT, Dj, and FCS have been implemented in C++ using the LEDA library [12] (version 4.5). The code is compiled with GCC (version 3.3.3) and the experiments were performed on a 2.8 GHz Intel Xeon machine running a Linux kernel (version 2.6.5).

4.1 Small Graphs

For each of the generated graph types **grid**, **rect**, **sixgrid**, **globe**, **del**, **leda**, **del-max**, and **leda-max**, we considered series of 20 graphs containing between 50 and 1000 nodes. We take into account all algorithms, LT, Dj, and FCS, optimized on separator size, and each node was once chosen as BFS root. As already described above, if more than one smallest separators have been found, the one with best balance is selected.

Concerning the grid-like and **globe** graphs, the differences between the three algorithms are quite small. Due to the regular construction of these graphs, LT and Dj always succeed right after the first phase, and the smallest BFS level is almost optimum. The mean size of a fundamental-cycle separator is always slightly smaller and yields better balance. For the randomly generated graphs, the results are different: For the Delaunay graphs, LT always terminates after the first phase with a smallest valid BFS level, while Dj applies for around 15% of the BFS roots the last phase of the algorithm. Figure 1 shows clearly that FCS computes on average the best separators, while Dj is slightly better than LT. Considering the LEDA random graphs, both LT and Dj always have to pass

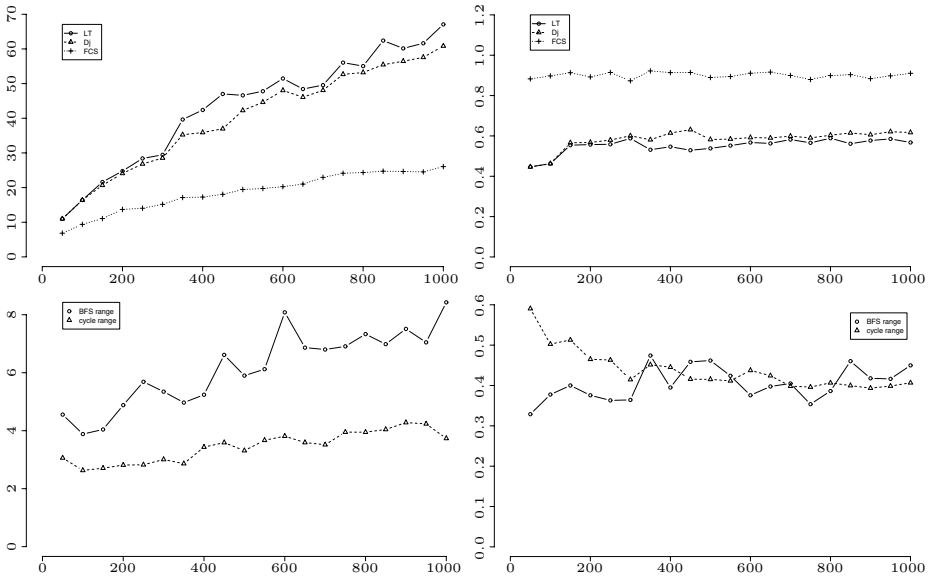


Fig. 1. Experiments with Delaunay graphs of sizes ranging from 50 to 1000 nodes: The upper diagrams show the mean separator size (left) and mean balance (right) with LT, Dj, and FCS. The lower diagrams show for FCS the ranges of the mean separator size (left) and mean balance (right), comparing BFS root and non-tree edge selection.

the third phase. The mean separator size of FCS is only slightly better than that of Dj, while LT is by far worse. The mean balance with LEDA graphs is similar for all algorithms between 0.8 and 0.9.

The lower diagrams in Figure 1 show the influence of BFS root selection and non-tree edge selection on separator size and balance. For FCS applied to the Delaunay graphs, the range of the mean of both the separator size and balance values are depicted, either over all possible BFS root nodes or over all possible non-tree edges. For example, the *range of the mean separator size over all possible BFS root nodes* is defined as follows: For every BFS root, determine the mean separator size over all possible non-tree edges. Then, the wanted *range of the mean* is the difference between the maximum and the minimum of these separator sizes among all BFS root nodes. The diagrams show that selection of the BFS root node is more decisive for the separator size than non-tree edge selection. Concerning balance, both selections are of similar importance.

4.2 Large Graphs

The second series of graphs that we experimented with, the *large graphs*, consists of 16 graphs of the categories mentioned in Section 3 of size roughly 10000 nodes (see Table 1). The **rectangular** graph represents a 20×500 raster, the **sixgrid** graph consists of 20×237 hexagons, the **globe** has 100 meridians and circles of latitude, and the **t-sphere** is constructed by 5 iterations. For **c-grid**, **c-del-max**

Table 1. Large graphs: The table depicts the number of nodes and edges, the diameter and the radius for both the graph itself (orig) and a triangulation of it (triang) as well as for each of the algorithms LT, Dj, and FCS, all of them optimized on separator size *with* postprocessing applied, the minimum and mean separator sizes over all BFS root nodes; bold-face and italic figures indicate the best result(s) for the respective graph

graph	nodes edges		diameter		radius		LT		Dj		FCS	
	orig	triang	orig	triang	min	mean	min	mean	min	mean		
grid	10000	19800	198	67	100	50	82	106	82	106	89	<i>99</i>
rectangular	10000	19480	518	20	260	10	20	27	20	27	20	<i>20</i>
sixgrid	9994	14733	513	22	257	11	21	28	21	28	21	<i>21</i>
triangular	5050	14850	99	45	66	34	58	83	58	83	58	<i>68</i>
globe	10002	20100	101	101	76	67	100	119	100	119	100	<i>106</i>
t-sphere	10242	30720	96	96	80	80	160	169	160	169	160	<i>164</i>
diameter	10000	29994	3333	3333	1667	1667	3	4	3	4	3	<i>3.3</i>
del	10000	25000	56	45	46	36	206	300	82	113	65	<i>75</i>
del-max	10000	29971	52	48	43	39	204	314	86	117	74	<i>79</i>
leda	9989	25000	18	15	11	8	76	216	7	31	5	8
leda-max	10000	29975	15	14	9	8	56	205	7	26	6	<i>10</i>
c-grid	10087	19904	212	72	106	36	38	78	38	78	5	<i>6.4</i>
c-globe	10090	20325	144	142	73	71	4	96	4	96	4	<i>12</i>
c-del-max	10005	29972	65	58	34	29	74	318	19	65	5	<i>8.3</i>
c-leda-max	10005	29984	20	16	11	8	78	209	7	32	4	<i>4.5</i>
airfoil1	4253	12289	65	31	36	21	50	89	26	85	26	<i>35</i>
city2	2948	3564	131	14	66	9	15	39	15	39	4	<i>9.5</i>
city3	15868	16690	658	13	329	9	28	53	28	53	4	<i>6.8</i>

and *c-leda-max*, the two copies of the respective graph are connected by 5 nodes, while for *c-globe* only 4 nodes are used to connect the two graphs.

Main Results. We investigated the performance in terms of separator size of LT, both unoptimized and optimized on separator size, Dj, and FCS, the latter ones optimized on separator size. We ran each of these algorithms for each graph while once making each node the root of the BFS tree.

The results of the experiments regarding the separator sizes achieved by the various algorithms are listed in Table 1, and illustrated in Figure 2 by means of box plots that represent the middle fifty per cent of the data series (note that the whiskers here span the whole range of outcomes). The data shows that—except for the *grid* graphs—the smallest minimum separator is found by FCS, and concerning the mean separator size FCS achieves the best result for all graphs under consideration. This, together with the fact that the boxes are clearly slender, and—except for *c-globe*—the ranges are minimal for FCS, suggests that FCS significantly outperforms the other algorithms in terms of separator size. In particular, this behavior is surprising for graphs with rather big diameter d and radius r (e.g., *c-globe*, *globe*, and *diameter*), since the guaranteed bound on the separator size is $2d + 1$ ($2r + 1$, respectively; cf. the description of FCS on page 631) for FCS.

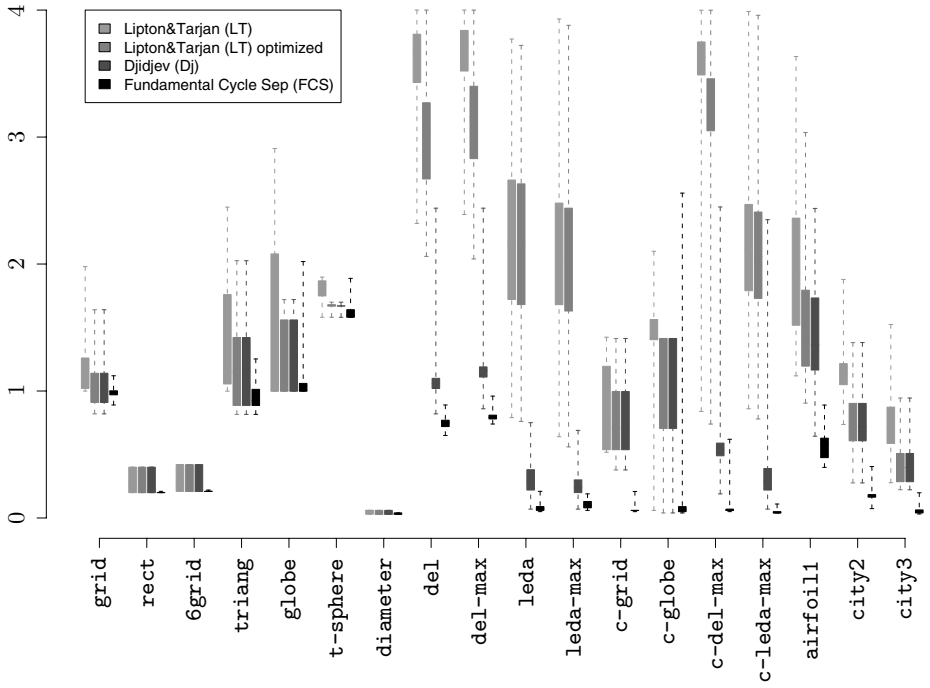


Fig. 2. Box plots depicting the separator sizes relative to \sqrt{n} obtained with unoptimized LT (light-gray) and LT (gray), Dj (dark-gray), and FCS (black), the latter three optimized on separator size. The dashed lines indicate the range of all separators *without* postprocessing applied.

For regular-structured graphs (i.e., grid-like, sphere approximation, and the *diameter* graphs) the separator sizes are similar and quite high for the three algorithms. For irregular graphs (i.e., *leda*, the graphs with small separator, and the real-world graphs), the picture looks different: The Dj algorithm always yields better results than the LT algorithm, and FCS is clearly superior to both Dj and LT. Furthermore, the minimum and mean separator sizes computed by FCS are by far below the guaranteed upper bounds.

The running time considering one BFS root node is linear for all of our algorithms. However, for the algorithms LT and Dj, the constant crucially depends on the phase in which the algorithms terminate (cf., Section 2.2). The first two phases consist basically of a breadth-first search, while the computation of the fundamental cycle requires expensive operations like embedding, triangulation, and copying. FCS, of course, computes a fundamental cycle and always needs the expensive operations. LT and Dj terminate after phase 1 with all grid-like graphs, sphere-approximating graphs, and with the *diameter*, *c-grid*, *c-globe*, and *city* graphs. In contrast, the LEDA, *c-del-max*, and *c-leda-max* graphs in the majority of cases require phase 3. For the Delaunay graphs, LT mostly terminates after phase 1, but Dj needs phase 3. The mean running time for LT

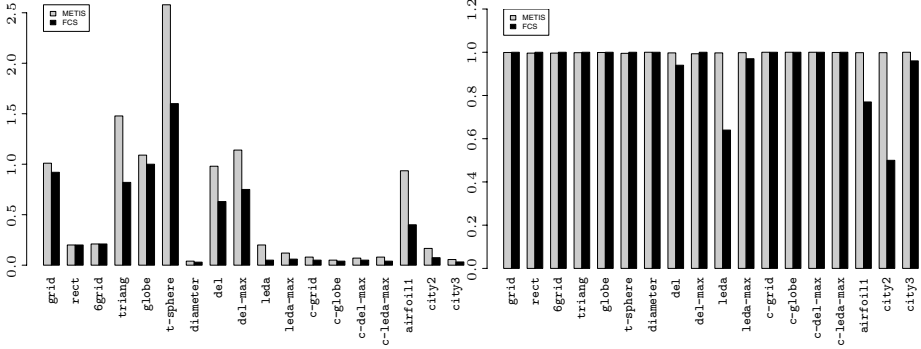


Fig. 3. Minimum separator sizes relative to \sqrt{n} (left) and balance (right) computed with MeTiS (light-gray) and with FCS (black) optimized on separator ratio

(applying only phase 1) in the *city3* graph, for example, is 0.04 seconds, while FCS involving a fundamental-cycle computation needs 0.71 seconds.

Postprocessing. The subsequent experiments deal with the effect of postprocessing on the various algorithms. We found in a pre-study that the optimization of separator size and separator ratio should be accompanied by a combination of Dulmage-Mendelsohn optimization followed by node expulsion.

Our results applying these combinations of postprocessing techniques can be summarized as follows (data not shown): On the one hand, for the grid-like and sphere-approximating graphs as well as the *diameter* graph, the separators found by the algorithms without postprocessing cannot be much improved. (Often the separators are already close to optimal solutions for these graphs.) On the other hand, for the remaining graphs, the separators computed by the algorithms Dj and LT are very large compared to an optimal solution, and in these cases the postprocessing greatly improves the separators. The separators computed by FCS can generally be improved only a little.

Benchmark. To get an idea of the quality of the separators found by the algorithms, we compare them against separators obtained with the help of *MeTiS* [16], a graph partitioning tool collection. We applied MeTiS to partition the nodes into two sets and observed very high balances (meeting the requirement that each part encompass at least one third of the graph’s nodes) with quite few cut edges. From a partitioning thus obtained we computed a node separator by choosing an appropriate subset of the end-nodes of the edges forming the cut.

Separators determined by MeTiS are a trade-off between separator size and balance, so for the sake of a meaningful comparison, we contrast the MeTiS results and our algorithms optimized on separator ratio. Since among LT, Dj, and FCS optimized on separator ratio, the solutions computed by FCS were the best with respect to both separator size and balance, we compare MeTiS with FCS only. Figure 3 shows the best separator size and balance values. One may state that with FCS, the separator size is always at least as good as with

MeTiS and balance is almost always comparable. Those graphs whose balance is considerably worse with FCS than with MeTiS (`leda` and `city2`) exhibit by far smaller separators with FCS, which suggests that the weighting between the two criteria, separator size and balance, seems to be more in favor of separator size with FCS, while MeTiS tends to prefer balance. Indeed, almost perfect balance can always be achieved with FCS optimized only on balance.

4.3 City Graphs

We consider a series of city graphs with numbers of nodes up to about 45,000. For these graphs we computed separations by the following linear-time procedure: run FCS on ten BFS trees of a given graph, determined by a random node as root, and from among these separations take the one with best separator ratio.

The results of the experiments with the city graph series are depicted in the table aside. Obviously, all city graphs have extremely small separators, which are also found by our algorithm. The separators for the `city2` and `city3` graphs, which had already been included in the experiments of the previous section, are somewhat bigger than those of the preceding experiment (8 and 7 instead of 4, resp.; see Table 1). This is due to the fact that: (i) the separator ratio is now optimized instead of the separator size; and (ii) we do not longer take into account every node as a BFS root.

graph	nodes	edges	size	balance
<code>city1</code>	1429	3034	5	0.871
<code>city2</code>	2948	3564	8	0.996
<code>city3</code>	15868	16690	7	0.869
<code>city4</code>	20036	41476	10	0.789
<code>city5</code>	24106	53826	5	0.740
<code>city6</code>	38823	79988	8	0.704
<code>city7</code>	44878	90930	7	0.547

5 Conclusions and Outlook

Our experiments have shown that, especially for graphs with small separators, there is a high potential for optimizing the separators computed by the algorithms. Both the postprocessing and in particular the Fundamental-Cycle Separation yielded almost-optimal separators with respect to separator size and balance. Applied to graphs whose triangulations have small diameter (which is true for many graphs, especially from real world), FCS is empirically and theoretically superior to the classical algorithms guaranteeing separators of size $O(\sqrt{n})$. Selection of the non-tree edge in the fundamental-cycle computation has a considerable influence on both criteria, and we are able to select the best during the respective algorithm. The choice of the BFS root also exhibits a great impact on separator quality, mainly on its size. The experiments on city graphs confirmed that FCS, applied to a small random sample of BFS root nodes and separator ratio as optimization criterion yields excellent separators in linear time.

An issue for further investigation would be to explore whether more sophisticated strategies for selecting an appropriate BFS root can be developed. In addition, we would like to investigate other parts of the algorithms that are also subject to arbitrary choices, namely triangulation and breadth-first search.

Acknowledgments

The authors would like to thank Imen Borgi and Jürgen Graf for their assistance with parts of the implementation work and the anonymous referees for their detailed comments and very helpful hints for further research.

References

1. Lipton, R.J., Tarjan, R.E.: A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics* **36** (1979) 177–189
2. Djidjev, H.N.: On the problem of partitioning planar graphs. *SIAM Journal on Algebraic and Discrete Methods* **3** (1982) 229–240
3. Alon, N., Seymour, P., Thomas, R.: Planar separators. *SIAM Journal on Discrete Mathematics* **7** (2004) 184–193
4. Djidjev, H.N., Venkatesan, S.M.: Reduced constants for simple cycle graph separation. *Acta Informatica* **34** (1997) 231–243
5. Aleksandrov, L., Djidjev, H.N., Guo, H., Maheshwari, A.: Partitioning planar graphs with costs and weights. In: *ALENEX 2002*. Volume 2409 of LNCS., Springer (2002) 98–110
6. Holzer, M., Prasinos, G., Schulz, F., Wagner, D., Zaroliagis, C.: Engineering planar separator algorithms. Technical Report 2005-20, Fakultät Informatik, Universität Karlsruhe (TH) (2005)
<http://www.ubka.uni-karlsruhe.de/vvv/ira/2005/20/20.pdf>.
7. Leighton, T., Rao, S.: Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *Journal of the ACM* **46** (1999) 787–832
8. Mehlhorn, K.: *Data Structures and Algorithms 1, 2, and 3*. Springer (1984)
9. Kozen, D.: *The Design and Analysis of Algorithms*. Springer (1992)
10. Ashcraft, C., Liu, J.W.H.: Applications of the Dulmage-Mendelsohn decomposition and network flow to graph bisection improvement. Technical Report CS-96-05, Dept. of Computer Science, York University, North York, Ontario, Canada (1996)
<http://www.cs.yorku.ca/techreports/1996/CS-96-05.html>.
11. Bourke, P.: Sphere generation (1992)
<http://astronomy.swin.edu.au/~pbourke/modelling/sphere/>.
12. Näher, S., Mehlhorn, K.: *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press (1999) <http://www.algorithmic-solutions.com>.
13. Diekmann, R.: (Graph Partitioning Graph Collection)
<http://wwwcs.upb.de/fachbereich/AG/monien/RESEARCH/PART/graphs.html>.
14. BARD: (Bay Area Regional Database) <http://bard.wr.usgs.gov>.
15. ESRI: (Environmental Systems Research Institute) <http://www.esri.com>.
16. Karypis, G.: (MeTiS) <http://www-users.cs.umn.edu/~karypis/metis>.

STXXL : Standard Template Library for XXL Data Sets

Roman Dementiev¹, Lutz Kettner², and Peter Sanders^{1,*}

¹ Fakultät für Informatik, Universität Karlsruhe,
Karlsruhe, Germany

{dementiev, sanders}@ira.uka.de

² Max Planck Institut für Informatik,
Saarbrücken, Germany

kettner@mpi-sb.mpg.de

Abstract. We present a software library STXXL, that enables practice-oriented experimentation with huge data sets. STXXL is an implementation of the C++ standard template library STL for external memory computations. It supports parallel disks, overlapping between I/O and computation, and *pipelining* technique that can save more than *half* of the I/Os. STXXL has already been used for computing minimum spanning trees, connected components, breadth-first search decompositions, constructing suffix arrays, and computing social network analysis metrics.

1 Introduction

Massive data sets arise naturally in many domains: geographic information systems, computer graphics, database systems, telecommunication billing systems, network analysis, and scientific computing. Applications working in those domains have to process terabytes of data. However, the internal memories of computers can keep only a small fraction of these huge data sets. During the processing the applications need to access the external storage (e.g. hard disks). One such access can be about 10^6 times slower than a main memory access. For any such access to the hard disk, accesses to the next elements in the external memory are much cheaper. In order to amortize the high cost of a random access one can read or write *contiguous* chunks of size B . One minimizes the number of I/Os performed, and to increase I/O bandwidth, applications use *multiple disks*, in parallel. In each I/O step the algorithms try to transfer D blocks between the main memory of size M and D disks (one block from each disk). This model has been formalized by Vitter and Shriver as Parallel Disk Model (PDM) [1] and is the standard theoretical model for designing and analyzing I/O-efficient algorithms. In this model, N is the input size and B is the block size measured in bytes.

Theoretically I/O-efficient algorithms and data structures have been developed for many problem domains: graph algorithms, string processing, computational geometry, etc. (for a survey see [2]). Some of them have been implemented:

* Partially supported by DFG grant SA 933/1-2.

sorting, matrix multiplication [3], (geometric) search trees [3], priority queues [4], suffix array construction [4]. However there is an increasing gap between theoretical achievements of external memory (EM) algorithms and their practical usage. Several EM software library projects (LEDA-SM [4] and TPIE [5]) have been started to reduce this gap. They offer frameworks which aim to speed up the process of implementing I/O-efficient algorithms, abstracting away the details of how I/O is performed.

We have started to develop an external memory library STXXL making more emphasis on performance, trying to avoid the drawbacks of the previous libraries impeding their practical usage. The following are some key features of STXXL:

- Transparent support of parallel disks.
- The library is able to handle problems of size up to dozens of terabytes.
- Explicit *overlapping* between I/O and computation.
- A library feature “*pipelining*” can save more than *half* the number of I/Os performed by many algorithms, directly feeding the output from an EM algorithm into another EM algorithm, without needing to store it on the disk in between.
- The library avoids superfluous copying of data blocks, e.g. in I/O subsystem.
- Short *development times* due to well known STL-compatible interfaces for EM algorithms and data structures. STL – Standard Template Library is the library of algorithms and data structures that is a part of the C++ standard. STL algorithms can be directly applied to STXXL containers; moreover the I/O complexity of the algorithms remains optimal in most of the cases.

STXXL library is open source and available under the Boost Software License 1.0 (http://www.boost.org/LICENSE_1_0.txt). The latest version of the library, a user tutorial and a programmer documentation can be downloaded at <http://stxxl.sourceforge.net>. Currently the size of the library is about 15 000 lines of code.

The remaining part of this paper is organized as follows. Section 2 discusses the design of STXXL. In Section 3 we implement a short benchmark and use it to study the performance of STXXL. Section 4 gives a short overview of the projects using STXXL. We make some concluding remarks and point out the directions of future work in Section 5.

Related Work. TPIE [3] was the first large software project implementing I/O-efficient algorithms and data structures. The library provides implementation of I/O-efficient sorting, merging, matrix operations, many (geometric) search data structures (B⁺-tree, persistent B⁺-tree, R-tree, K-D-B-tree, KD-tree, Bkd-tree), and the logarithmic method. The work on the TPIE project is in progress.

LEDA-SM [4] EM library was designed as an extension to the LEDA library for handling large data sets. The library offers implementations of I/O-efficient sorting, EM stack, queue, radix heap, array heap, buffer tree, array, B⁺-tree, string, suffix array, matrices, static graph, and some simple graph algorithms. However, the data structures and algorithms can not handle more than 2³¹ bytes. The development of LEDA-SM has been stopped.

LEDA-SM and TPIE libraries currently offer only single disk EM algorithms and data structures. They are not designed to explicitly support overlapping between I/O and computation. The overlapping relies largely on the operating system that caches and prefetches data according to a general purpose policy, which can not be as efficient as the *explicit* approach. Furthermore, overlapping based on system cache on most of the operating systems requires additional copies of the data, which leads to CPU and internal memory overhead.

The idea of pipelined execution of the algorithms that process large data sets not fitting into main memory is very well known in relational database management systems. The pipelined execution strategy allows to execute a database query with minimum number of EM accesses, to save memory space to store intermediate results, and to obtain the first result as soon as possible.

FG [6] is a design framework for parallel programs running on clusters, where parallel programs are split into series of asynchronous stages, which are executed in the pipelined fashion with the help of multithreading. This allows to mitigate disk access latency, communication network latency, and overlap I/O and communication.

2 STXXL Design

STXXL consists of three layers (see Figure 1). The lowest layer, the Asynchronous I/O primitives layer (AIO layer) abstracts away the details of how asynchronous I/O is performed on a particular operating system. Other existing EM algorithm libraries rely only on synchronous I/O APIs [4] or allow reading ahead sequences stored in a file using the POSIX asynchronous I/O API [5]. Unfortunately, asynchronous I/O APIs are very different on different operating systems (e.g. POSIX AIO and Win32 overlapped I/O). Therefore, we have introduced the AIO layer to make porting STXXL easy. Porting the whole library to a different platform (for example Windows) requires only reimplementing the AIO layer using native file access methods and/or native multithreading mechanisms. STXXL has already several implementations of the layer which use synchronous file access methods under POSIX/UNIX systems. The `read/write` calls using direct access (`O_DIRECT` option) have shown the best performance under Linux. To provide asynchrony we use POSIX threads or Boost threads.

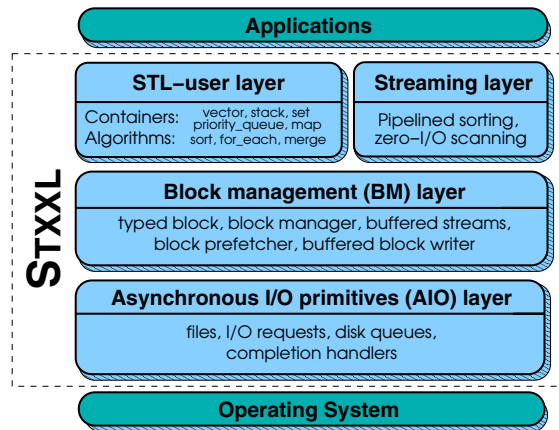


Fig. 1. Structure of STXXL

The Block Management layer (BM layer) provides a programming interface simulating the *parallel* disk model. The block manager implements block allocation/deallocation allowing several block-to-disk assignment strategies: striping, randomized striping, randomized cycling, etc. The BM layer provides implementation of parallel disk buffered writing [7], optimal prefetching [7], and block caching. The implementations are fully *asynchronous* and designed to explicitly support *overlapping* between I/O and computation.

The top of STXXL consists of two modules. The STL-user layer provides EM data structures which have (almost) the same interfaces (including syntax and semantics) as their STL counterparts. The Streaming layer provides efficient support for *pipelining* EM algorithms. The algorithms for external memory suffix array construction implemented with this module [8] require only 1/3 of I/Os which must be performed by implementations that use conventional data structures and algorithms (either from STXXL STL-user layer, or LEDA-SM, or TPIE).

The rest of this section discusses the STL-user and Streaming layers in more detail. The detailed description of the BM and AIO layers can be found in the extended version of the paper [9].

2.1 STL-User Layer

Containers

Vector is an array whose size can vary dynamically. Similar to LEDA-SM arrays [4], the user has the choice over the block striping strategy of `vector`, the size of the vector cache, the cache replacement strategy (LRU, random, user-defined). STXXL `vector` has STL compatible Random Access Iterators. One random access costs $\mathcal{O}(1)$ I/Os in the worst case. Sequential scanning of the vector costs $\mathcal{O}(1/DB)$ amortized I/Os per vector element.

EM priority queues are used for time-forward processing technique in external graph algorithms [10,2] and online sorting. The STXXL implementation of `priority_queue` is based on [11]. This queue needs less than a third of I/Os used by other similar cache (I/O) efficient priority queues. The implementation supports parallel disks and overlaps I/O and computation.

The current version of STXXL also has an implementation of EM `map` (based on B⁺-tree), `FIFO queue`, and several efficient implementations of `stack`.

STXXL allows to store the references to objects located in EM using EM iterators (e.g. `stxxl::vector::iterator`). The iterators remain valid while storing to and loading from EM. When dereferencing an EM iterator, the pointed object is loaded from EM by the library on demand.

STXXL containers differ from the STL containers in their treatment of memory and distinction of uninitialized and initialized memory. STXXL containers assume that the data types they store are plain old data types (POD). The constructors and destructors of the contained data types are not called when a container changes its size. The support of constructors and destructors would imply significant I/O cost penalty, e.g. on the deallocation of a non-empty container, one has to load all contained objects and call their destructors. This

restriction sounds more severe than it is, since EM data structures can not cope with custom dynamic memory management anyway, the common use of custom constructors/destructors. However, we plan to implement special versions of STXXL containers which will support not only PODs and handle construction/destruction appropriately.

Algorithms

The algorithms of STL can be divided into two groups by their memory access pattern: *scanning* algorithms and *random access* algorithms.

Scanning algorithms. These are the algorithms that work with Input, Output, Forward, and Bidirectional iterators only. Since random access operations are not allowed with these kinds of iterators, the algorithms inherently exhibit strong spatial locality of reference. STXXL containers and their iterators are STL-compatible, therefore one can directly apply STL scanning algorithms to them, and they will run I/O-efficiently (see the use of `std::generate` and `std::unique` algorithms in the Listing 1.1). Scanning algorithms are the majority of the STL algorithms (62 out of 71). STXXL also offers specialized implementations of some scanning algorithms (`stxxl::for_each`, `stxxl::generate`, etc.), which perform better in terms of constant factors in the I/O volume and internal CPU work. Being aware of the sequential access pattern of the applied algorithm, the STXXL implementations can do prefetching and use queued writing, thereby enabling overlapping of I/O with computation.

Random access algorithms. These algorithms require RandomAccess iterators, hence may perform many random I/Os¹. For such algorithms, STXXL provides specialized I/O-efficient implementations that work with STL-user layer external memory containers. Currently the library provides two implementations of sorting: an `std::sort`-like sorting routine – `stxxl::sort`, and a sorter that exploits integer keys – `stxxl::ksort`. Both sorters are highly efficient parallel disk implementations. The algorithm they implement guarantees close to optimal I/O volume and almost perfect overlapping between I/O and computation [7]. The performance of the sorter scales well. With eight disks which have peak bandwidth of 380 MB/s it sorts 128 byte elements with 32 bit keys achieving I/O bandwidth of 315 MB/s.

Listing 1.1 shows how to program using the STL-user layer and how STXXL containers can be used together with both STXXL algorithms and STL algorithms. This example generates a huge random directed graph in sorted edge array representation. The edges must be sorted lexicographically. A straightforward procedure to do this is to: 1) generate a sequence of random edges, 2) sort the sequence, 3) remove duplicate edges from it. The STL/STXXL code for it is only five lines long: Line 1 creates an STXXL EM vector with 10 billion edges. Line 2 fills the vector with random edges (`generate` from STL is used, `random_edge` functor returns random edge objects). In the next line the STXXL sorter sorts randomly generated edges using 512 megabytes of internal memory. The lexicographical order is defined by functor `my_cmp`. Line 6 deletes duplicate

¹ The `std::nth_element` algorithm is an exception. It needs $\mathcal{O}(N/B)$ I/Os on average.

edges in the EM vector with the help of the STL `unique` algorithm. The `NewEnd` vector iterator points to the right boundary of the range without duplicates. Finally (Line 7), we chop the vector at the `NewEnd` boundary.

Listing 1.1. Generating a random graph using the STL-user layer

```

1 | stxxl::vector<edge> Edges(1000000000ULL);
2 | std::generate(Edges.begin(), Edges.end(), random_edge());
3 | stxxl::sort(Edges.begin(), Edges.end(), edge_cmp(),
4 |           512*1024*1024);
5 | stxxl::vector<edge>::iterator NewEnd =
6 |           std::unique(Edges.begin(), Edges.end());
7 | Edges.resize(NewEnd - Edges.begin());

```

2.2 Streaming Layer

The streaming layer provides a framework for *pipelined* processing of large sequences. The pipelined processing technique is well known in the database world. To the best of our knowledge we are the first to apply this method systematically in the domain of EM algorithms. We introduce it in the context of an EM software library.

Usually the interface of an EM algorithm assumes that it reads the input from EM container(s) and writes output to EM container(s). The idea of pipelining is to equip the EM algorithms with a new interface that allows them to feed the output as a data stream directly to the algorithm that consumes the output, rather than writing it to EM. Logically, the input of an EM algorithm does not have to reside in EM, it could be rather a data stream produced by another EM algorithm.

Many EM algorithms can be viewed as a data flow through a directed acyclic graph $G = (V = F \cup S \cup R, E)$. The *file nodes* F represent physical data sources and data sinks, which are stored on disks (e.g. in the EM containers of STL-user layer). A file node outputs or/and reads one stream of elements. Streaming nodes S are equivalent to scan operations in non-pipelined EM algorithms, but do not perform any I/O, unless a node needs to access EM data structures. Sorting nodes R read a stream and output it in a sorted order. Edges E in the graph G denote the directions of data flow between nodes. A pipelined execution of the computations in a data flow is possible in an I/O-efficient way [8].

In STXXL, all data flow node implementations have an STXXL stream interface which is similar to STL Input iterators². As an input iterator, an STXXL stream object may be dereferenced to refer to some object and may be incremented to proceed to the next object in the stream. The reference obtained by dereferencing is read-only and must be convertible to the `value_type` of the STXXL stream. STXXL stream has a boolean member function `empty()` which returns `true` iff the end of the stream is reached. The binding of a STXXL stream object to its input streams (incoming edges in a data flow graph G) happens at compile time using templates, such that we benefit from function inlining in

² Do not confuse with the stream interface of the C++ `iostream` library.

C++. After constructing all node objects, the computation starts in a “lazy” fashion, first trying to evaluate the result of the topologically latest node. The node reads its intermediate input nodes, element by element, using dereference and increment operator of the STXXL stream interface. The input nodes proceed in the same way, invoking the inputs needed to produce an output element. This process terminates when the result of the topologically latest node is computed. This style of pipelined execution scheduling is I/O-efficient, it allows to keep the intermediate results in-memory without needing to store them in EM.

In the extended version of the paper [9] we show how to “pipeline” the random graph generation example from the previous chapter, such that the number of I/Os is more than halved.

3 Performance

We demonstrate some performance characteristics of STXXL using the EM maximal independent set (MIS) algorithm from [10] as an example. This algorithm is based on the time-forward processing technique. As the input for the MIS algorithm, we use the random graph computed by the examples in the previous Section (Listings 1.1 and its pipelined version [9]). Our benchmark includes the running time of the input generation.

The MIS algorithm given in Listing 1.2 is only nine lines long not including declarations. The algorithm visits the graph nodes scanning lexicographically sorted input edges. When a node is visited, we add it to the maximal independent set if none of its visited neighbours is already in the MIS. The neighbour nodes of the MIS nodes are stored as events in a priority queue. In Lines 6–7, the template metaprogram [12] `PRIORITY_QUEUE_GENERATOR` computes the type of priority queue that will store events. The metaprogram finds the optimal values for numerous tuning parameters (the number and the maximum arity of external/internal mergers, the size of merge buffers, EM block size, etc.) under the constraint that the total size of the priority queue internal buffers must be limited by `PQ_MEM` bytes. The `node_greater` comparison functor defines the order of nodes of type `node_type` and minimum value that a node object can have, such that the `top()` method will return the smallest contained element. The last template parameter tells that the priority queue can not contain more than `INPUT_SIZE` elements (in 1024 units). Line 8 creates the priority queue `depend` having prefetch buffer pool of size `PQ_PPOOL_MEM` bytes and buffered write memory pool of size `PQ_WPOOL_MEM` bytes. The external vector `MIS` stores the nodes belonging to the maximal independent set. Ordered input edges come in the form of an STXXL stream called `edges`. If the current node `edges->src` is not a neighbour of a MIS node (the comparison with the current event `depend.top()`, Line 13), then it is included in MIS (if it was not there before, Line 15). All neighbour nodes `edges->dst` of a node in MIS `edges->src` are inserted in the event priority queue `depend` (Line 16). Lines 11–12 remove the events already passed through from the priority queue.

Listing 1.2. Computing a Maximal Independent Set using STXXL

```

1  struct node_greater : public std::greater<node_type> {
2      node_type min_value() const {
3          return std::numeric_limits<node_type>::max();
4      }
5  };
6  typedef stxxl::PRIORITY_QUEUE_GENERATOR<node_type,
7      node_greater, PQ_MEM, INPUT_SIZE/1024>::result pq_type;
8  pq_type depend(PQ_PPPOOL_MEM, PQ_WPOOL_MEM);
9  stxxl::vector<node_type> MIS; // output
10 for (;!edges.empty();++edges) {
11     while(!depend.empty() && edges->src > depend.top())
12         depend.pop(); // delete old events
13     if(depend.empty() || edges->src != depend.top()) {
14         if(MIS.empty() || MIS.back() != edges->src)
15             MIS.push_back(edges->src);
16         depend.push(edges->dst);
17     }
18 }

```

To make a comparison with other EM libraries, we have implemented the graph generation algorithm using TPIE and LEDA-SM. The MIS algorithm was implemented in LEDA-SM using its array heap data structure as a priority queue. The I/O-efficient implementation of the MIS algorithm was not possible in TPIE, since it does not have an I/O-efficient priority queue implementation. For TPIE, we report only the running time of the graph generation. The source code of all our implementations is available under <http://i10www.ira.uka.de/dementiev/stxxl/paper/index.shtml>.

To make the benchmark closer to real applications, the `edge` data structure has two 32-bit integer fields, which can store some additional information associated with the edge. The priority queues of LEDA-SM always store a pair `<key,info>`. The `info` field takes at least four bytes. Therefore, to make a fair comparison with STXXL, we have changed the event data type stored in the priority queue, such that it also has a 4-byte dummy `info` field.

The experiments were run on a 2-processor Xeon (2 GHz) workstation (only one processor was used) and 1 GB of main memory (swapping was switched off). The OS was Debian Linux with kernel 2.4.20. The computer had four 80 GB IDE (IBM/Hitachi 120 GXP series) hard disks formatted with the XFS file system and dedicated solely for the experiments. We used LEDA-SM version 1.3 with LEDA version 4.2.1³ and TPIE of January 21, 2005. For compilation of STXXL and TPIE sources, the `g++` version 3.3 was used. LEDA-SM and LEDA were compiled with `g++` version 2.95, because they could not be compiled by later `g++` versions. The optimization level was set to `-O3`. We used library sorters that use C++ comparison operators to compare elements. All programs have been tuned to achieve their maximum performance. We have tried all available

³ Later versions of the LEDA are not supported by the last LEDA-SM version 1.3.

file access methods and disk block sizes. In order to tune the TPIE benchmark implementation, we followed the performance tuning Section of [5]. The input size (the length of the random edge sequence, see Listing 1.1) for all tests was 2000 MB⁴. The benchmark programs were limited to use only 512 MB of main memory. The remaining 512 MB are given to operating system kernel, daemons, shared libraries and file system buffer cache, from which TPIE and LEDA-SM might benefit. The STXXL implementations do not use the file system cache.

Table 1. Running time (in seconds)/I/O bandwidth (in MB/s) of the MIS benchmark running on single disk. For TPIE only graph generation is shown (marked with *).

		LEDA-SM	STXXL-STL	STXXL-Pipel.	TPIE
Input graph generation	Filling	51/41	89/24	100/20	40/52
	Sorting	371/23	188/45		307/28
	Dup. removal	160/26	104/40	128/26	109/39
MIS computation		513/6	153/21		-N/A-
Total		1095/16	534/33	228/24	456*/32*

Table 1 compares the MIS benchmark performance of the LEDA-SM implementation, the STXXL implementation based on the STL-user level, a pipelined STXXL implementation, and a TPIE implementation (only input graph generation). The running times, averaged over three runs, and average I/O bandwidths are given for each stage of the benchmark. The running time of the different stages of the pipelined implementation cannot be measured separately. However, we show the values of time and I/O counters from the beginning of the execution till the time when the sorted runs are written to the disk(s) and from this point to the end of the MIS computation. The total time numbers show that the pipelined STXXL implementation is significantly faster than the other implementations. It is 2.4 times faster than the second leading implementation (STXXL-STL). The win is due to reduced I/O volume: the STXXL-STL implementation transfers 17 GB, the pipelined implementation needs only 5.2 GB. However the 3.25 fold I/O volume reduction does not imply equal reduction of the running time because the run formation fused with filling/generating phase becomes compute bound. This is indicated by the almost zero value of the STXXL I/O wait counter, which measures the time the processing thread waited for the completion of an I/O. The second reason is that the fusion of merging, duplicate removal and CPU intensive priority queue operations in the MIS computation is almost compute bound. Comparing the running times of the total input graph generation we conclude that STXXL-STL implementation is about 20 % faster than TPIE and 53 % faster than LEDA-SM. This could be due to better (explicit) overlapping between I/O and computation. Another possible reason could be that TPIE uses a more expensive way of reporting run-time

⁴ Algorithms and data structures of LEDA-SM are limited to inputs of size 2 GB.

Table 2. Running time (in seconds)/I/O bandwidth (in MB/s) of the MIS benchmark running on multiple disk

Disks		STXXL-STL		STXXL-Pipelined	
		2	4	2	4
Input graph generation	Filling	72/28	64/31	98/20	98/20
	Sorting	104/77	80/100		
	Dup. removal	58/69	34/118	112/30	110/31
MIS computation		127/25	114/28		
Total		360/50	291/61	210/26	208/27

errors, such as I/O errors⁵. The running time of the filling stage of STXXL-STL implementation is much higher than of TPIE and LEDA-SM because they rely on operating system cache. The filled blocks do not go immediately to the disk(s) but remain in the main memory until other data needs to be cached by the system. The indication of this is the very high bandwidth of 52 MB/s for TPIE implementation, which is even higher than the maximum physical disk bandwidth (48 MB/s) at its outermost zone. However, the cached blocks need to be flushed in the sorting stage and then the TPIE implementation pays the remaining due. The unsatisfactory bandwidth of 24 MB/s of the STXXL-STL filling phase could be improved to 33 MB/s by replacing the call `std::generate` by the native `stxxl::generate` call that efficiently overlaps I/O and computation. STXXL STL-user sorter sustains an I/O bandwidth of about 45 MB/s which is 95 % of the disk's peak bandwidth. The high CPU load in the priority queue and not very perfect overlapping between I/O and computation explain the low bandwidth of the MIS computation stage in all three implementations. We also run the graph generation test on 16 GByte inputs. All implementations scale almost linearly with the input size: the TPIE implementation finishes in 1h 3min, STXXL-STL in 49min, and STXXL-Pipelined in 28min.

The MIS computation of STXXL, which is dominated by PQ operations, is 3.35 times faster than LEDA-SM. The main reason for this big speedup is likely to be the more efficient priority queue algorithm from [11].

Table 2 shows the parallel disk performance of the STXXL implementations. The STXXL-STL implementation achieves speedup of about 1.5 using two disks and 1.8 using four disks. The reason for this low speedup is that many parts of the code become compute bound: priority queue operations in the MIS computation, run formation in the sorting, and generating random edges in the filling stage. The STXXL-Pipelined implementation was almost compute bound in the single disk case, and as expected, with two disks the first phase shows no speedup. However the second phase has a small improvement in speed due to faster I/O.

⁵ TPIE uses function return types for error codes and diagnostics, which can become quite expensive at the level of the single-item interfaces (e.g. `read_item` and `write_item`) that is predominantly used in TPIEs algorithms. Instead, STXXL checks (I/O) errors on the per-block basis. We will use C++ exceptions to propagate errors to the user layer without any disadvantage for the library users. First experiments indicate that this will have negligible impact on runtime.

Close to zero I/O wait time indicates that the STXXL-Pipelined implementation is fully compute bound when running with two or four disks. The longest MIS computation, requiring the entire space of four disks (360 GBytes), for the graph with $4.3 \cdot 10^9$ nodes and $13.4 \cdot 10^9$ edges took 2h 44min on an Opteron system.

4 Applications

STXXL has been successfully applied in implementation projects that studied various I/O efficient algorithms from the practical point of view. The fast algorithmic components of STXXL library gave the implementations an opportunity to solve problems of very large size on a low-cost hardware in a record time.

The performance of EM *suffix array construction* algorithms was investigated in [8]. The experimentation with pipelined STXXL implementations of the algorithms has shown that computing suffix arrays in EM is feasible even on a low-cost machine. Suffix arrays for long strings up to 4 billion characters could be computed in hours.

The project [13] has compared experimentally two EM *breadth-first search* (BFS) algorithms. The pipelining technique of STXXL has helped to save a factor of 2–3 in I/O volume. Using STXXL, it became possible to compute BFS decomposition of large grid graphs with 128 million edges in less than a day, and for random sparse graphs within an hour.

Simple algorithms for computing *minimum spanning trees* (MST), *connected components*, and *spanning forests* were developed in [14]. Their implementations were built using STL-user-level algorithms and data structures of STXXL. The largest solved MST problem had 2^{32} nodes, the input graph edges occupied 96 GBytes. The computation on a PC took 8h 40min.

The number of triangles in a graph is a very important metric in social network analysis. We have designed and implemented an external memory algorithm that counts and lists all triangles in a graph. Using our implementation we have counted the number of triangles of a web crawl graph from the WebBase project⁶. In this graph the nodes are web pages and edges are hyperlinks between them. For the computation we ignored the direction of the links. Our crawl graph had 135 million nodes and 1.2 billion edges. During computation on an Opteron SMP which took only 4h 46min we have detected 10.6 billion triangles. Total volume of 851 GB was transferred between 1GB of main memory and seven hard disks. The details about the algorithm and the source code are available under <http://i10www.ira.uka.de/dementiev/tria/algorithm.shtml>.

5 Conclusions

We have described STXXL: a library for external memory computation that aims for high performance and ease-of-use. The library supports parallel disks and explicitly overlaps I/O and computation. The library is easy to use for people who know the C++ Standard Template Library. STXXL supports algorithm pipelining, which saves many I/Os for many EM algorithms. Several projects using

⁶ <http://www-diglib.stanford.edu/~testbed/doc2/WebBase/>

STXXL have been finished already. With help of STXXL, they have solved very large problem instances externally using a low cost hardware in a record time. The work on the project is in progress. Future directions of STXXL development cover the implementation of the remaining STL containers, improving the pipelined sorter with respect to better overlapping of I/O and computation, implementations of graph and text processing EM algorithms. We plan to submit STXXL to the collection of the Boost C++ libraries (www.boost.org) which includes a Windows port.

References

1. Vitter, J.S., Shriver, E.A.M.: Algorithms for parallel memory, I/II. *Algorithmica* **12** (1994) 110–169
2. Meyer, U., Sanders, P., Sibeyn, J., eds.: Algorithms for Memory Hierarchies. Volume 2625 of LNCS Tutorial. Springer (2003)
3. Arge, L., Procopiuc, O., Vitter, J.S.: Implementing I/O-efficient Data Structures Using TPIE. In: 10th European Symposium on Algorithms (ESA). Volume 2461 of LNCS., Springer (2002) 88–100
4. Crauser, A.: LEDA-SM: External Memory Algorithms and Data Structures in Theory and Practice. PhD thesis, Universität des Saarlandes, Saarbrücken (2001) <http://www.mpi-sb.mpg.de/~crauser/diss.pdf>.
5. L. Arge, R. Barve, D. Hutchinson, O. Procopiuc, L. Toma, D. E. Vengroff, R. Wickeremesinghe: TPIE: User manual and reference. (2003)
6. Davidson, E.R., Cormen, T.H.: Building on a Framework: Using FG for More Flexibility and Improved Performance in Parallel Programs. (In: 19th International Parallel and Distributed Processing Symposium (IPDPS 2005)) to appear.
7. Dementiev, R., Sanders, P.: Asynchronous parallel disk sorting. In: 15th ACM Symposium on Parallelism in Algorithms and Architectures, San Diego (2003) 138–148
8. Dementiev, R., Mehnert, J., Kärkkäinen, J., Sanders, P.: Better External Memory Suffix Array Construction. In: Workshop on Algorithm Engineering & Experiments, Vancouver (2005) <http://i10www.ira.uka.de/dementiev/files/DKMS05.pdf> see also <http://i10www.ira.uka.de/dementiev/esuffix/docu/data/diplom.pdf>.
9. Dementiev, R., Kettner, L., Sanders, P.: Stxxl: Standard Template Library for XXL Data Sets. Technical Report 18, Fakultät für Informatik, University of Karlsruhe (2005)
10. Zeh, N.R.: I/O Efficient Algorithms for Shortest Path Related Problems. PhD thesis, Carleton University, Ottawa (2002)
11. Sanders, P.: Fast priority queues for cached memory. *ACM Journal of Experimental Algorithmics* **5** (2000)
12. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison Wesley Professional (2000)
13. Ajwani, D.: Design, Implementation and Experimental Study of External Memory BFS Algorithms. Master's thesis, Max-Planck-Institut für Informatik, Saarbrücken, Germany (2005)
14. Dementiev, R., Sanders, P., Schultes, D., Sibeyn, J.: Engineering an External Memory Minimum Spanning Tree Algorithm. In: IFIP TCS, Toulouse (2004) 195–208

Negative Cycle Detection Problem

Chi-Him Wong and Yiu-Cheong Tam

The Chinese University of Hong Kong
{02658924, 02654543}@alumni.cse.cuhk.edu.hk

Abstract. In this paper, we will describe some heuristics that can be used to improve the runtime of a wide range of commonly used algorithms for the negative cycle detection problem significantly, such as Bellman-Ford-Tarjan (BFCT) algorithm, Goldberg-Radzik (GORC) algorithm and Bellman-Ford-Moore algorithm with Predecessor Array (BFCF). The heuristics are very easy to be implemented and only require modifications of several lines of code of the original algorithms. We observed that the modified algorithms outperformed the original ones, particularly in random graphs and no cycle graphs. We discovered that 69% of test cases have improved. Also, the improvements are sometimes dramatic, which have an improvement of a factor of 23, excluding the infinity case, while the worst case has only decreased by 85% only, which is comparably small when compared to the improvement.

1 The Negative Cycle Detection Problem

1.1 Introduction

The Negative Cycle Detection problem has numerous applications in model verification, compiler construction, software engineering, VLSI design, scheduling, circuit production, constraint programming and image processing. For example, Constraint-based program analysis requires feasibility checking of constraint sets. Constraint graphs are often used to represent systems of difference constraints; an application of Farkas' Lemma shows that a system of difference constraints is feasible if and only if there are no negative cost cycles in the corresponding constraint graph. That is, a difference constraint problem is feasible if and only if there are no negative cycles in the graph.

In the design of VLSI circuits, it is required to isolate negative feedback loops. These negative feedback loops correspond to negative cost cycles in the amplifier-gain graph of the circuit. The problem of checking whether a zero-clairvoyant scheduling system has a valid schedule can also be reduced to the problem of identifying negative cost cycles in the appropriate graph. Recent approaches to the image segmentation problem are also based on negative cycle detection. Most of the approaches are based on the famous Bellman-Ford (BF) algorithm. All

⁰ The work described in this paper was partially supported by a direct allocation grant from the Research Grant Council of the Hong Kong Special Administrative Region, China (Project No. 2050321).

these algorithms have their worst case bound of $O(|V||E|)$ in runtime. Most of the previous algorithms run relatively slow on no cycle graphs. However, no cycle graph is common in practice. So we try to develop some heuristics that are good in detecting no cycle graphs. Our heuristics can be used to improve the runtime of a wide range of Bellman-Ford based algorithms. We discovered that 69% of test cases have improved. Also, the improvements are sometimes dramatic, which have an improvement of a factor of 23, excluding the infinity case, while the worst case has only decreased by 85% only, which is comparably small when compared to the improvement. There are also significant improvement in no cycle graphs and random graphs.

2 Related Works

2.1 Definitions

The Negative Cycle Detection problem can be defined as the problem of deciding whether a negative cost cycle exists in a directed graph. This does not require finding a path from a particular source to a particular destination, as it is only a decision problem and require only a yes or no answer. Formally, the Negative Cycle Detection (NCD) problem is defined as follows:

Given a directed graph $G = \langle V, E, c \rangle$, where $V = \{ v_0, v_1, \dots, v_{n-1} \}$, $|V| = n$, $E = \{ e_{ij} : v_i \rightarrow v_j \}$, $|E| = m$, and a cost function $c : E \rightarrow \mathbb{Z}$, is there a negative cost cycle in G ?

There are no restrictions on the edge costs, i.e., they can be arbitrary integers as opposed to small integers, as required by some scaling algorithms. We can even extend the weights to floating point numbers.

Our heuristics can be applied to a wide range of commonly used algorithms for the Negative Cycle Detection problem that is a *single source algorithm*. In the following, we will define the meaning of single source algorithm.

An algorithm for the negative cycle detection problem is defined as a single source algorithm if and only if the algorithm is relaxation-based and all the relaxations (or calculations of labels) originate from a single vertex (the source).

We will give some examples of single source algorithms in the following.

2.2 The Bellman-Ford Moore Algorithm with Predecessor Array (BFCA)

The Bellman-Ford-Moore (BFMO) algorithm attempts to reduce the number of vertices that must be examined in each stage of the “standard” Bellman-Ford (BF) algorithm by using a First-In-First-Out (FIFO) queue to store the vertices whose distance labels were changed in the previous stage.

Predecessor Array is a strategy that uses parent pointer to store the parent of each vertex v_i , where the parent of v_i is the vertex that caused the most recent label change to v_i . This strategy is widely used in many Bellman-Ford based algorithms.

The Bellman-Ford-Moore algorithm with Predecessor Array (BFCF) is an algorithm that combines the above two techniques. As the algorithm starts with a vertex with label 0 and all the other vertices of label infinity, it is a single source algorithm. Notice that BFCF still has the worst case bound of $O(|V||E|)$ in runtime.

2.3 The Bellman-Ford-Tarjan Algorithm (BFCT)

Another variation of the BF algorithm is BFCT, which combines the “standard” BF algorithm with the FIFO queue of BFFI and the sub-tree disassembly cycle detection strategy due to Tarjan. The sub-tree disassembly strategy is implemented by describing the tree structure by, besides the usual predecessor function, the first son and the adjacent brother function (next and previous). This allows traversals of sub-trees in linear time, and tree modifications in constant time. When no cycle is found in the traversal of a sub-tree T_v , all vertices in T_v will be discarded from the queue and the tree. [Tar81]. A negative cycle will be detected when a tree path from a vertex goes back to one of its ancestors. As the algorithm is started with one source of label 0 and all the other vertices of labels infinity, it is a single source algorithm.

2.4 The Goldberg-Radzik Algorithm (GORC)

The Goldberg-Radzik algorithm improved the Bellman-Ford-Moore algorithm. It achieves the same worst-case bound of $O(|V||E|)$, but can usually outperform BFFI in practice. The algorithm maintains the set of labelled vertices in two queues, queue A and queue B . Vertices in queue A will undergo a Bellman-Ford-Moore pass while vertices in queue B will undergo a depth first search with no update. At the beginning of the algorithm, the source is put in queue B . In pass B , if there is an outgoing arc with reduced cost including zero, the path is traversed and all vertices visited will be put in queue A . Notice that there are no updates on the vertices’ labels in this step. Also, if a reduced path from a vertex goes back to one of its ancestors, a negative cycle is detected. On the other hand, pass A is a one step Bellman-Ford-Moore pass that relaxes the vertices in queue A following the queue order. Notice that topological sort will be done after the depth first searches in pass B , so we will relax the vertices in pass A topologically and the label will be updated.

The algorithm starts with one source of label 0 and all the other vertices of labels infinity, it is a single source algorithm.

3 Our New Approach to This Problem

Our approach is to incorporate two heuristics to the above single source algorithms.

3.1 Heuristic One: Pumping Negative Strategy

The first heuristic is based on an observation that in any negative cost cycle, we can find a negative weighted edge e such that when we travel through the

negative cost cycle starting from e , the accumulated costs are always negative. The formal definition of this lemma is as follows.

Lemma 1: Given a directed graph $G = \langle V, E, c \rangle$, where $V = \{ v_0, v_1, \dots, v_{n-1} \}$, $|V| = n$, $E = \{ e_{ij} : v_i \rightarrow v_j \}$, $|E| = m$, and c is cost function $E \rightarrow Z$, with a negative cost cycle $\{ v_{\pi(1)}, v_{\pi(2)}, v_{\pi(3)}, \dots, v_{\pi(k)} \}$, where $v_{\pi(1)} = v_{\pi(k)}$, there exists at least one node $v_{\pi(i)}$ where $1 \leq i \leq k$ such that when we travel through the negative cost cycle starting from $v_{\pi(i)}$, the accumulated costs are always negative.

In this heuristic, we will stop the relaxation once the label becomes positive. But since we do not know which vertex in the negative cycle is the starting vertex with the property as stated in Lemma 1, we need to treat each vertex as source once. The proof of Lemma 1 is shown in Appendix II.

3.2 Heuristic Two: Reduced Cost Elimination

This heuristic is simply not deleting the labels on the vertices when moving from one pass to another with different vertices as the source. Consider a case where a particular vertex relaxes another vertex with label that is not marked 0. If the label is smaller than the accumulated cost, we will have the following two conclusions:

1. That particular vertex has been travelled previously as it has a label that is not 0.
2. The accumulated cost of the previous travel must be smaller than current accumulated cost as the label is smaller than the current accumulated cost.

Therefore, as the current accumulated cost is larger, using the current accumulated cost will reduce the number of vertices visited. Therefore, we can keep the cost label.

4 Modifications on Various Algorithms

In this section, we will briefly describe how we implement our heuristics into some single source algorithms.

4.1 Modification of BFCF

For heuristic one, we first change the algorithm such that the original BFCF algorithm will run V times with a different vertex as the source each time. We also add a condition for the relaxation process which is, a relaxation can only be done and continued if the accumulated cost is negative. When there are no more relaxations, a new source will be chosen and the whole process is repeated. For heuristic two, we can implement it simply by keeping the cost label on each vertex unchanged when we start a new pass with a new source. The pseudo-code is as follows:

Initialize all cost labels to 0. /*So, all accumulated cost will be negative (Heuristic 1)*/


```

For all v /*(Heuristic 1)*/
{
  BFCF with v as the source /*(Heuristic 1)*/
  /*Keep all the labels unchanged (Heuristic 2)*/
}

```

4.2 Modifications of BFCT

The modifications to BFCT are similar to that in BFCF. However, each vertex in BFCT will be involved in the first son and adjacent brother functions (next and previous). During our modification, we can leave the values of the function (next and previous) unchanged when we choose a new starting point. This will improve the runtime without affecting the correctness.

4.3 Modifications of GORC

The modifications for GORC are a bit different from that of BFCF and BFCT. For heuristic one, we need to modify the condition in pass *B* to ensure that the depth first search traversal will only traverse an edge when it leads to a negative reduced cost. GORC has already implemented the reduced cost elimination heuristic, so we do not need to do any modification for heuristic two.

5 Experimental Setups

5.1 Results

The problem generator was developed by the authors of [Gol95]. It can be downloaded from the website <http://www.avglab.com/andrew/index.html>. There are several sets of data.

1. The Rand-5 families have a fixed network size $n=2000000$ and $m=10000000$. The maximum arc length U is fixed at 32000 and the minimum arc length L varies from 0 to -64000 [Gol95].
2. The SQNC families represent the square grid families. Vertices of these networks correspond to points on the $x - y$ plane with integer coordinates $[x, y]$, $0 \leq x \leq X$, $0 \leq y \leq Y$ and $X = Y$. The SQNC01 family has no cycles. The SQNC02 family has sparse small negative cycles. The SQNC03 family has dense small negative cycles. The SQNC04 family has several long cycles. The SQNC05 family has Hamilton cycle.
3. The LNC families are the grid networks mentioned above but with $Y=16$. The LNC01, LNC02, LNC03, LNC04 and LNC05 are similar to those in the SQNC families.
4. The PNC families are the layered networks. A layered network consists of layers $0 \dots X-1$. Each layer is a simple cycle plus a collection of arcs connecting randomly selected pairs of vertices on the cycle. In the PNC families each layer contains 32 vertices and $X = n/32$ [Gol95]. The PNC01, PNC02, PNC03, PNC04 and PNC05 are similar to those in the SQNC families.

There are 5 test cases for each row shown in the following tables. The number of scan operations shown is the average of the 5 test cases. The percentage improvement is calculated according to the number of operation of relaxing. Results with more than 10% faster or slower will be highlighted in different colors. Finally, the 'M' in front of the name of the algorithm implies the modified algorithm with our heuristics implemented. The results are shown in Appendix I.

6 Conclusion

In this paper, we described two heuristics that can be incorporated into a wide range of commonly used single source algorithms for the Negative Cycle Detection problem. The modifications are very simple, involving only adding several lines of code.

After modification, all algorithms have increased the speed generally. There are significant results in the set of random graphs and no cycle graphs. We discovered that 69% of test cases have improved. Also, the improvements are sometimes dramatic, which have an improvement of a factor of 23, excluding the infinity case, while the worst case has only decreased by 85% only, which is comparably small when compared to the improvement.

References

- [CG96] Boris V. Cherkassky and Andrew V. Goldberg. Negative-cycle detection algorithms. In Josep D'iaz and Maria Serna, editors, Algorithms—ESA '96, Fourth Annual European Symposium, volume 1136 of Lecture Notes in Computer Science, pages 349–363, Barcelona, Spain, 25–27 September 1996. Springer.
- [Gol95] Andrew V. Goldberg. Scaling algorithms for the shortest paths problem. *SIAM Journal on Computing*, 24(3):494–504, June 1995.
- [Tar81] R. E. Tarjan. Shortest Paths. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, 1981.

Appendix I

Table 1. Results for no cycle graphs LNC01

	#Vertices	MGORC	GORC	Improve	MBFCT	BFCT	Improve
LNC01	8193	29571	51894	75.5%	14891	27338	83.6%
	16385	59399	102501	72.6%	29864	54115	81.2%
	32769	117508	204668	74.2%	59016	107378	81.9%
	65537	236006	412421	74.8%	118558	217019	83.0%
	131073	471187	824874	75.1%	236706	433352	83.1%
	262145	940478	1643797	74.8%	472542	864805	83.0%
	524289	1881286	3289308	74.8%	945099	1733480	83.4%
	1048577	3764090	6590393	75.1%	1890963	3455877	82.8%
	2097153	7523534	13186716	75.3%	3779505	6917921	83.0%
	4194305	15052684	26342784	75.0%	7561213	13826436	82.9%
	8388609	30111238	52697650	75.0%	15127398	27660704	82.9%
	16777217	60193979	58728907	-2.4%	30237773	30997195	2.5%
Average				68.32%			76.11%

Table 2. Results for no cycle graphs PNC01

	#Vertices	MGORC	GORC	Improve	MBFCT	BFCT	Improve
PNC01	8193	0	141440	INF	0	101024	INF
	16385	0	282502	INF	0	206452	INF
	32769	0	577308	INF	0	421004	INF
	65537	0	1175657	INF	0	840400	INF
	131073	0	2326792	INF	0	1697588	INF
	262145	0	4662634	INF	0	3378583	INF
	524289	0	9360087	INF	0	6776352	INF
	1048577	0	18799475	INF	0	13545258	INF
	2097153	0	37359269	INF	0	27079159	INF
	4194305	0	74824647	INF	0	54191807	INF
	8388609	0	149660040	INF	0	108381976	INF
Average				INF			INF

Table 3. Results for no cycle graphs SQNC01

	#Vertices	MGORC	GORC	Improve	MBFCT	BFCT	Improve
SQNC01	4097	13503	29080	115.4%	6933	12259	76.8%
	16385	51932	114828	121.1%	26775	48554	81.3%
	65537	203158	479398	136.0%	104971	191840	82.8%
	262145	807586	1961943	142.9%	418223	733359	75.4%
	1048577	3211258	7930611	147.0%	1663635	2919808	75.5%
	4194305	12817258	32338613	152.3%	6643025	11873024	78.7%
	16777217	51213266	140101492	173.6%	26550354	48035413	80.9%
Average				141.19%			78.77%

Table 4. Results for a few short cycle graphs LNC02

	#Vertices	MGORC	GORC	Improve	MBFCT	BFCT	Improve
LNC02	8193	6442	11480	78.2%	3240	5825	79.8%
	16385	19433	34692	78.5%	9716	17916	84.4%
	32769	42381	46765	10.3%	21241	24144	13.7%
	65537	90512	114548	26.6%	45277	61044	34.8%
	131073	151996	267506	76.0%	75979	141937	86.8%
	262145	247744	438869	77.1%	123912	231623	86.9%
	524289	696032	1163851	67.2%	348365	616440	77.0%
	1048577	1563907	1669857	6.8%	782080	883473	13.0%
	2097153	1845286	3252535	76.3%	923584	1729789	87.3%
	4194305	6011875	6697666	11.4%	3007849	3560722	18.4%
	8388609	13820476	24466542	77.0%	6911521	12969000	87.6%
	16777217	22339356	37514660	67.9%	11172724	19902492	78.1%
Average				54.44%			62.32%

Table 5. Results for a few short cycle graphs PNC02

	#Vertices	MGORC	GORC	Improve	MBFCT	BFCT	Improve
PNC02	8193	47185	59183	25.4%	29484	30705	4.1%
	16385	86450	64450	-25.4%	53720	54659	1.7%
	32769	225235	205400	-8.8%	140052	142698	1.9%
	65537	287805	203174	-29.4%	178108	180636	1.4%
	131073	350055	294782	-15.8%	218465	212419	-2.8%
	262145	503878	386424	-23.3%	314458	330199	5.0%
	524289	1316216	1095546	-16.8%	820598	874808	6.6%
	1048577	4790290	3915894	-18.3%	2983431	3175931	6.5%
	2097153	10428337	8593237	-17.6%	6495202	6953796	7.1%
	4194305	23124100	19013193	-17.8%	14408767	15390086	6.8%
	8388609	67316841	55413705	-17.7%	41913396	44775974	6.8%
Average				-15.05%			4.10%

Table 6. Results for a few short cycle graphs SQNC02

	#Vertices	MGORC	GORC	Improve	MBFCT	BFCT	Improve
SQNC02	4097	13503	29080	115.4%	6933	12259	76.8%
	16385	51932	114828	121.1%	26775	48554	81.3%
	65537	203158	479398	136.0%	104971	191840	82.8%
	262145	807586	1961943	142.9%	418223	733359	75.4%
	1048577	3211258	7930611	147.0%	1663635	2919808	75.5%
	4194305	12817258	32338613	152.3%	6643025	11873024	78.7%
	16777217	51213266	140101492	173.6%	26550354	48035413	80.9%
Average				141.19%			78.77%

Table 7. Results for many short cycles graphs LNC03

	#Vertices	MGORC	GORC	Improve	MBFCT	BFCT	Improve
LNC03	8193	343	396	15.3%	173	196	13.2%
	16385	71	136	93.2%	31	47	48.4%
	32769	112	275	145.9%	54	72	33.1%
	65537	208	262	26.2%	106	97	-8.5%
	131073	180	221	22.3%	92	91	-0.7%
	262145	138	297	115.5%	72	82	14.2%
	524289	240	349	45.4%	120	110	-8.0%
	1048577	198	224	13.4%	97	69	-28.7%
	2097153	149	313	110.3%	74	73	-1.1%
	4194305	172	344	99.7%	86	86	-0.2%
	8388609	216	261	20.9%	106	102	-3.6%
	16777217	193	202	4.6%	95	100	5.7%
Average				59.39%			5.32%

Table 8. Results for many short cycles graphs PNC03

	#Vertices	MGORC	GORC	Improve	MBFCT	BFCT	Improve
PNC03	8193	461	2713	489.1%	327	865	164.6%
	16385	473	3046	543.5%	256	451	75.7%
	32769	468	3843	721.2%	248	513	106.9%
	65537	341	3030	788.0%	165	829	402.9%
	131073	967	3842	297.1%	697	686	-1.7%
	262145	381	3670	863.8%	214	339	58.7%
	524289	1144	4174	265.0%	620	525	-15.3%
	1048577	177	2743	1453.5%	113	244	116.9%
	2097153	1125	3747	233.0%	617	552	-10.6%
	4194305	314	2063	557.5%	175	394	124.9%
	8388609	632	2628	315.5%	292	409	40.1%
Average				593.38%			96.65%

Table 9. Results for many short cycles graphs SQNC03

	#Vertices	MGORC	GORC	Improve	MBFCT	BFCT	Improve
SQNC03	4097	109	487	345.3%	55	173	216.1%
	16385	74	584	686.5%	34	249	625.0%
	65537	250	1101	340.5%	130	493	280.1%
	262145	477	3779	691.6%	243	1077	343.2%
	1048577	1818	7839	331.1%	950	2441	157.0%
	4194305	2934	7534	156.8%	1536	4593	199.0%
	16777217	5896	28282	379.7%	3066	8719	184.4%
Average				418.79%			286.40%

Table 10. Results for a few long cycles graphs LNC04

	#Vertices	MGORC	GORC	Improve	MBFCT	BFCT	Improve
LNC04	8193	26711	54962	105.8%	12910	20687	60.2%
	16385	92773	148990	60.6%	41107	56489	37.4%
	32769	251488	349870	39.1%	122379	145197	18.6%
	65537	881848	948447	7.6%	365853	376227	2.8%
	131073	1980993	2097090	5.9%	874778	898293	2.7%
	262145	4467623	5460818	22.2%	1892207	2215706	17.1%
	524289	11883080	11385873	-4.2%	4968220	5017279	1.0%
	1048577	25741146	28462486	10.6%	10792331	12567955	16.5%
	2097153	62102224	61088246	-1.6%	26089374	27892978	6.9%
	4194305	121983730	128064383	5.0%	53771346	59798591	11.2%
	8388609	275905623	286903304	4.0%	112925426	135532732	20.0%
	16777217	538729562	664472322	23.3%	231721552	299999324	29.5%
Average				23.19%			18.66%

Table 11. Results for a few long cycles graphs PNC04

	#Vertices	MGORC	GORC	Improve	MBFCT	BFCT	Improve
PNC04	8193	96459	82610	-14.4%	55914	37737	-32.5%
	16385	363660	189675	-47.8%	222163	89015	-59.9%
	32769	1032742	416393	-59.7%	633831	215134	-66.1%
	65537	2475918	932459	-62.3%	1574038	485710	-69.1%
	131073	3959249	2244526	-43.3%	2482065	1042791	-58.0%
	262145	11673063	4881046	-58.2%	6672867	2282332	-65.8%
	524289	26999818	10280214	-61.9%	16888696	4707412	-72.1%
	1048577	56517150	21103100	-62.7%	35114982	9594361	-72.7%
	2097153	118365443	44260038	-62.6%	74302031	20881883	-71.9%
	4194305	225343295	95936613	-57.4%	147809535	43256228	-70.7%
	8388609	517666048	189688611	-63.4%	328137483	88085199	-73.2%
Average				-53.97%			-63.74%

Table 12. Results for a few long cycles graphs SQNC04

	#Vertices	MGORC	GORC	Improve	MBFCT	BFCT	Improve
SQNC04	4097	32707	30880	-5.6%	15213	13944	-8.3%
	16385	184893	170162	-8.0%	81463	67445	-17.2%
	65538	573055	746841	30.3%	254273	297501	17.0%
	262145	1313928	3264595	148.5%	560271	1389863	148.1%
	1048577	2338506	15068659	544.4%	1099832	6288913	471.8%
	4194305	6204248	68242481	999.9%	2710118	27430729	912.2%
	16777217	14024018	272653950	1844.2%	6681120	121993247	1725.9%
Average				507.67%			464.21%

Table 13. Results for Hamiltonian cycle graphs LNC05

	#Vertices	MGORC	GORC	Improve	MBFCT	BFCT	Improve
LNC05	8193	163438	172029	5.3%	98947	91658	-7.4%
	16385	346868	364541	5.1%	215632	202262	-6.2%
	32769	753945	803979	6.6%	458880	436855	-4.8%
	65537	1633918	1733490	6.1%	999207	934728	-6.5%
	131073	3499646	3789556	8.3%	2082565	2027643	-2.6%
	262145	7599126	7828071	3.0%	4637622	4329335	-6.6%
	524289	15608355	16876480	8.1%	9456863	9179454	-2.9%
	1048577	33572684	35232411	4.9%	20445301	19287842	-5.7%
	2097153	69453620	74266615	6.9%	42213747	40477734	-4.1%
	4194305	144322754	156777150	8.6%	89613042	84456141	-5.8%
	8388609	314273826	331798694	5.6%	189346386	181584841	-4.1%
	16777217	637684719	668256739	4.8%	387236035	364980712	-5.7%
Average				6.11%			-5.20%

Table 14. Results for Hamiltonian cycle graphs PNC05

	#Vertices	MGORC	GORC	Improve	MBFCT	BFCT	Improve
PNC05	8194	82594	90718	9.8%	54836	47319	-13.7%
	16386	179350	203306	13.4%	116631	100007	-14.3%
	32770	361404	401511	11.1%	249244	210670	-15.5%
	65538	785227	870523	10.9%	516925	443592	-14.2%
	131074	1732688	1680790	-3.0%	1096942	901936	-17.8%
	262146	3462811	3965235	14.5%	2232233	1928211	-13.6%
	524290	7714697	7216528	-6.5%	4579138	4039317	-11.8%
	1048578	16872712	16794782	-0.5%	9786195	8450277	-13.7%
	2097154	32924136	37489022	13.9%	19973817	17243048	-13.7%
	4194306	63798970	71903161	12.7%	41054033	35357906	-13.9%
	8388610	154010041	151672136	-1.5%	86223637	74532047	-13.6%
Average				-1.50%			-14.16%

Table 15. Results for Hamiltonian cycle graphs SQNC05

	#Vertices	MGORC	GORC	Improve	MBFCT	BFCT	Improve
SQNC05	4098	70976	79742	12.4%	44744	41610	-7.0%
	16386	360423	365960	1.5%	207067	197097	-4.8%
	65538	1644198	1712316	4.1%	983910	928755	-5.6%
	262146	7650803	8070454	5.5%	4679524	4297402	-8.2%
	1048578	32523850	34451433	5.9%	19818466	18737042	-5.5%
	4194306	151666794	149113446	-1.7%	91909813	82151343	-10.6%
	16777218	648487235	648211387	0.0%	394621759	353276864	-10.5%
Average				3.96%			-7.46%

Table 16. Results for random graphs RAND-5

Negative Weight	MGORC	GORC	Improve	MBFCT	BFCT	Improve
0	0	7780968	INF	0	4578450	INF
-1000	1133358	8892373	684.6%	341958	5350722	1464.7%
-2000	2080317	9588453	360.9%	762572	6481434	749.9%
-4000	260253	3665786	1308.5%	211730	5139325	2327.3%
-6000	328345	884558	169.4%	218531	1634924	648.1%
-8000	224668	41588	-81.5%	115428	721280	524.9%
-16000	603	541	-10.4%	328455	237747	-27.6%
-32000	175	174	-0.5%	329102	143538	-56.4%
-64000	51	48	-4.3%	380534	55175	-85.5%

Table 17. General results for GORC

	Improve	Degrade
100% < X	36.1%	0.0%
10% < X ≤ 100%	27.8%	17.0%
X ≤ 10%	12.8%	6.3%
Total	76.7%	23.3%

Table 18. General results for BFCT

	Improve	Degrade
100% < X	24.8%	0.0%
10% < X ≤ 100%	29.3%	25.7%
X ≤ 10%	9.6%	10.6%
Total	63.7%	36.3%

Appendix II

Proof of Lemma 1

Consider a negative cost cycle C in a directed graph $G(V, E)$ where $C = \{v_0, v_1, \dots, v_{n-1}\}$. The accumulated costs on C starting from v_0 are $\{C_{1,0}, C_{1,1}, \dots, C_{1,n-1}\}$, where $C_{1,k}$ is the accumulated cost at vertex v_k starting from v_0 .

First of all, at the end of the cycle, i.e. v_{n-1} , the accumulated cost (W_{cycle}) along this cycle must be negative. Secondly, there is at least one maximum accumulated cost C_{max} on this cycle. Finally, there must be at least one edge, (v_i, v_j) , where $j = i+1$, that is negative since there must be at least one negative edge on a negative cycle. Note that we can change the ordering of the vertices by choosing another “starting vertex”. Suppose that we take v_k as the starting vertex where v_k has the largest accumulated cost (chose the last one if there is more than one), we will have the accumulated costs $\{C_{2,k+1}, C_{2,k+2}, \dots, C_{2,n-1}, C_{2,0}, C_{2,1}, \dots, C_{2,k}\}$.

Now we will have the path $\{v_{k+1}, v_{k+2}, \dots, v_{n-1}, v_0, v_1, \dots, v_k\}$. Then we will have two sets of vertices. The first set contains the vertices in $\{v_{k+1}, v_{k+2}, \dots, v_{n-1}\}$. For any vertex v in this set, $C_{2,v} = C_{1,v} - C_{max} < 0$, where $C_{max} \geq 0$ because otherwise, v_0 is already the correct starting point. The second set contains the vertices in $\{v_0, v_1, \dots, v_k\}$. For any vertex v in this set, $C_{2,v} = C_{1,v} + W_{cycle} - C_{max} < 0$.

An Optimal Algorithm for Querying Priced Information: Monotone Boolean Functions and Game Trees

Ferdinando Cicalese^{1,*} and Eduardo Sany Laber²

¹ Institut für Bioinformatik, Universität Bielefeld, Germany
nando@cebitec.uni.bielefeld.de

² Department of Informatics, PUC, Rio de Janeiro, Brasil
laber@inf.puc-rio.br

Abstract. We study competitive function evaluation in the context of computing with priced information. A function f has to be evaluated for a fixed but unknown choice of the values of the variables. Each variable x of f has an associated cost $c(x)$, which has to be paid to read the value of x . The problem is to design algorithms that compute the function querying the values of the variables sequentially while trying to minimize the total cost incurred. The evaluation of the performance of the algorithms is made by employing competitive analysis. We determine the best possible extremal competitive ratio for the classes of threshold trees, game trees, and monotone boolean functions with constrained minterms, by providing a polynomial-time algorithm whose competitiveness matches the known lower bounds.

1 Introduction

Priced information sources have recently been studied in many different contexts. Notably, priced information has been favorably compared to free access policies as a means of improving the access to the information on the Internet [5]. With appropriate priced information schemes, the end-user may have guaranteed more efficient access to the desired data [5]; however, the need arises for competitive methodologies for reducing the cost incurred.

Our approach to the development of competitive procedures for accessing and processing priced information follows the line of research initiated by the seminal paper of Charikar et al. [2], where the basic problem of computing a given function by adaptively querying the values of its variables is addressed. In this model, reading the value of a variable is subject to the payment of some cost and, in general, different variables may incur different costs. An algorithm that solves the problem has to decide how to sequentially query the variables in order to evaluate the function.

* Supported by the Sofja Kovalevskaja Award 2004 of the Alexander von Humboldt Foundation.

Problem Statement. More formally, we consider the following scenario. A function $f(x_1, \dots, x_n)$ has to be evaluated for a fixed but unknown choice of the values for the set of variables $V = \{x_1, x_2, \dots, x_n\}$. Each variable x_i has an associated non-negative cost $c(x_i)$ which is the cost incurred to probe x_i , i.e., to read its value. Given $U \subseteq V$, we define the cost $c(U)$ of U as the sum of the costs of the variables in U , i.e., $c(U) = \sum_{x \in U} c(x)$. The goal is to *adaptively* identify and probe a minimum cost set of variables $U \subseteq V$ whose values uniquely determine the value of f , regardless of the value of the variables not probed.

An *assignment* σ for f is a choice of a value for each one of its variables. We shall denote by $x_i(\sigma)$ the value assigned to x_i in the assignment σ . We use $f(\sigma)$ to denote the value of f w.r.t. σ , i.e., $f(\sigma) = f(x_1(\sigma), \dots, x_n(\sigma))$. Given a subset $U \subseteq V$, we use $\sigma|_U$ to denote the restriction of σ to the variables in U . We say that U is *sufficient* with respect to a given assignment σ of V if the value of f is determined by the partial assignment $\sigma|_U$. A set of variables which is sufficient is also called a *proof* of the value of f for the given assignment σ .

An evaluation algorithm \mathbb{A} for f under an assignment σ is a rule to adaptively read the variables in V until the set of variables read so far is sufficient with respect to σ . The cost of the algorithm \mathbb{A} for an assignment σ is the total cost incurred by \mathbb{A} to evaluate f under the assignment σ . All the algorithms considered here have access to the variables costs.

Given a cost function $c(\cdot)$, we let $c_{\mathbb{A}}^f(\sigma)$ denote the cost of the algorithm \mathbb{A} for an assignment σ and $c^f(\sigma)$ be the cost of the cheapest proof for f under the assignment σ . We say that \mathbb{A} is ρ -competitive if $c_{\mathbb{A}}^f(\sigma) \leq \rho c^f(\sigma)$, for every possible assignment σ . We use $\gamma_{\mathbb{A}}^{\Delta}(f)$ to denote the competitive ratio of \mathbb{A} , that is, the minimum ρ for which \mathbb{A} is ρ -competitive. The best possible competitive ratio for any deterministic algorithm, then, is $\gamma_c^f = \min_{\mathbb{A}} \gamma_{\mathbb{A}}^{\Delta}(f)$, where the minimum is computed over all possible deterministic algorithms \mathbb{A} . With the aim of evaluating the dependence of the competitive ratio on the structure of f , one defines the extremal competitive ratio $\gamma^{\Delta}(f)$ of an algorithm \mathbb{A} as $\gamma^{\Delta}(f) = \max_c \gamma_{\mathbb{A}}^{\Delta}(f)$. The best possible extremal competitive ratio for any deterministic algorithm, then, is $\gamma(f) = \min_{\mathbb{A}} \gamma^{\Delta}(f)$. This last measure is meant to capture the structural complexity of f independent of a particular cost assignment and algorithm.

Our Results. We present a new implementation of the general approach introduced in [3], that dramatically improves upon the previously published results. The new implementation arises from a simple idea, that consists of forcing a greedy-like choice into the general approach. As a result we obtain an algorithm that achieves the optimal extremal competitive ratio for the classes of threshold trees, game trees (under a weak assumption on the allowed assignments) and general monotone boolean functions in which each variable appears in at most three minterms.

Related Work. The seminal paper for the study of the effect that priced information has on basic algorithmic problems is [2]. In this paper, the problems of evaluating the classes of monotone boolean functions, threshold trees and game trees, among others, are addressed. For monotone boolean functions, a $2\gamma(f)$ ex-

ponential time algorithm is outlined. For the subclass of the monotone boolean functions that are representable by AND-OR tree, a polynomial time algorithm with extremal competitive ratio $\gamma(f)$ and a γ_c^f -competitive pseudo-polynomial algorithm are provided. Two variants of the latter are shown to achieve $2\gamma_c^f$ -competitiveness for the classes of threshold trees and game trees, respectively. We believe that the result for game trees needs further investigation [6]. In the extended version of the paper we shall discuss the arguable points in [2] at greater length.

It is noticeable that, as opposed to ours, the results concerning competitiveness with respect to γ_c^f are obtained by pseudo-polynomial algorithms. In fact, all known results concerning the γ_c^f -competitiveness are attained by non-polynomial procedures, unless additional assumptions on the structure of the costs are made [4].

In [3], many of the results of [2] were improved, in terms of extremal competitiveness. For monotone boolean function a $(k + l - \sqrt{\min\{k, l\}})$ -competitive algorithm was given, where k and l denote the sizes of the largest minterm and maxterm of the function under evaluation. Moreover an algorithm with competitiveness $1.618\gamma(f)$ for threshold tree and one with competitiveness $1.5\gamma(f)$ for game trees were provided. All these algorithm are obtained as implementations of the general approach introduced in [3], and run in polynomial time.

In Table 1 our results are compared to the previously published ones.

Table 1. Summary of the results. By PP-TIME we mean pseudo-polynomial time. The superscript ⁺ indicates that the result holds for the subclass of functions such that $\min\{k, l\} \leq 2$. The superscript * restricts the result to the subclass of functions for which every variable appears in at most 3 minterms.

Function Class	Charikar et. al [2]	Our previous paper [3]	This paper
Threshold trees	$2\gamma_c(f)$ PP-TIME	$1.618\gamma(f)$ PTIME	$\gamma(f)$ PTIME
Monotone Bool	$2\gamma(f)$ EXP-TIME	$2\gamma(f)$ and $\gamma(f)^+$ PTIME	$\gamma(f)^*$ PTIME
Game trees	$2\gamma_c(f)$ PP-TIME	$1.5\gamma(f)$ PTIME	$\gamma(f)$ PTIME

2 The General Approach

In this section we present the *general approach* introduced in [3] as a novel schema for the design of algorithm in the framework of querying with priced information.

Before we start we need to fix some notation. Let $Y \subseteq V$ and let σ_Y be an assignment for the variables of Y . We use f_Y to denote the restriction of f obtained by fixing the values of the variables in Y as given by σ_Y .

The general approach, presented in the pseudo-code below, consists of inter-actively executing the following steps until the function f is evaluated: a set U of unevaluated variables is selected; then the variable u of minimum cost in U is evaluated and the current cost of each variable in U is decreased by the current

cost of u . The procedure is iterated on the function f_u obtained from f by fixing the value of u . Note that the rule employed to select the set of variables U is left unspecified. It is such a rule that determines both the competitiveness and the computational complexity of the resulting algorithm.

```

Algorithm GENAPP( $f, V, c$ )
   $Y = \emptyset$ 
  While the value of  $f_Y$  is not known
    Select  $U_Y \subseteq V - Y$ 
     $u \leftarrow$  variable of  $U_Y$  with minimum cost w.r.t.  $\mathbf{c}$ 
    Read( $u$ )
     $Y = Y \cup \{u\}$ 
    For each  $v \in U_Y - \{u\}$  do  $c(v) \leftarrow c(v) - c(u)$ .
  EndWhile
End Algorithm
    
```

We say that an algorithm \mathbb{A} is an implementation of the general approach if \mathbb{A} is obtained from GENAPP by fixing the rule for selecting the set U_Y among all subsets of $V - Y$. The following lemma is the main tool for analyzing the implementations of the general approach.

Lemma 1. [3] *Let \mathbb{G} be an implementation of GENAPP. Let \mathcal{U} be the family of sets U_Y selected by \mathbb{G} . Let β_Y be the largest integer such that $|U_Y \cap P| \geq \beta_Y$ for every proof P for f_Y . Then, $\gamma^{\mathbb{G}}(f) \leq \max_{U_Y \in \mathcal{U}} \left\{ \frac{|U_Y|}{\beta_Y} \right\}$.*

3 Optimal Implementations of the General Approach

In this section we present the new algorithm obtained by implementing the general approach with a greedy selection of the set U_Y .

3.1 Monotone Boolean Functions

We shall first consider the case when f is a monotone boolean function over the set of variable $V = \{x_1, \dots, x_n\}$. Recall that a function f is monotone (non-decreasing) if it is not possible to decrease the value of f by increasing $x(\sigma)$ for some variable $x \in V$.

A 1-witness for f is a set of variables $T \subseteq V$ such that if $x(\sigma) = 1$ for every $x \in T$, then f evaluates to 1, no matter how are assigned the values for the remaining variables. On the other hand, a 0-witness for f is a set of variable $S \subseteq V$ such that if $x(\sigma) = 0$ for every $x \in S$, then f evaluates to 0. A minterm is a minimal 1-witness and a maxterm is a minimal 0-witness.

An immediate consequence of these definition is that each minterm of f has non-empty intersection with each maxterm of f .

We use $k(f)$ and $l(f)$ to denote the size of the largest minterm and the largest maxterm of f , respectively. We use the term *certificate* to either refer to a minterm or to a maxterm. Note that every proof for f contains a certificate.

Theorem 1 ([2]). *If f is a monotone function then $\gamma(f) \geq \max\{k(f), l(f)\}$.*

Implementing the General Approach. The following lemma analyses the particular implementation of the general approach that always selects U_Y as a minterm for f_Y . It turns out that such a choice results in optimal extremal competitive ratio for all assignments σ such that $f(\sigma) = 0$.

Lemma 2. *Let \mathbb{A} be an implementation of the general approach in which U_Y is a minterm for f_Y . Then, for every assignment σ for which $f(\sigma) = 0$, we have $c_{\mathbb{A}}^f(\sigma) \leq k(f)c^f(\sigma)$.*

Proof. Since $f(\sigma) = 0$, then, trivially, for every set Y of variable read by \mathbb{A} , we have $f_Y(\sigma) = 0$, whence every proof P of f_Y contains a maxterm of f_Y . On the other hand, since the set U_Y is a minterm for f_Y , it intersects every maxterm of f_Y , i.e., $|U_Y \cap P| \geq 1$ for every proof P for f_Y . Moreover, we have that $|U_Y| \leq k(f_Y) \leq k(f)$. Then, the desired result $c_{\mathbb{A}}^f(\sigma) \leq k(f)c^f(\sigma)$ immediately follows by Lemma 1.

The Greedy Choice. Although the implementation \mathbb{A} in Lemma 2 has an optimal performance for each assignment σ such that $f(\sigma) = 0$, there is no guarantee about its performance for the assignments σ' such that $f(\sigma') = 1$. Note that \mathbb{A} selects an arbitrary minterm of f . Now we discuss how to optimize this choice in order to obtain algorithms with optimal competitiveness.

Let \mathcal{F} denote the family of minterms of f . We shall now define an total order χ on \mathcal{F} that induces a sorting of the minterms of f in order of non-decreasing cost.

Definition 1 (Ranks). *Let $f = f(x_1, x_2, \dots, x_n)$ be a boolean function and let $c(\cdot)$ be a cost function on the variables of f . Let π be the total order on the variables of f defined by stipulating that, for each $i = 1, 2, \dots, n - 1$, x_i precedes x_{i+1} in the order π . Therefore, c and π induce a total order χ on the minterms of f as follows. In χ a minterm C precedes a minterm D if and only if one of the following conditions holds: (a) $c(C) < c(D)$; (b) $c(C) = c(D)$ and the list of variables in C (listed according to π) precedes in the lexicographical order the list of variables in D (listed according to π).*

For each minterm C of f we define $rank_f(C)$ as the ordinal position of C in χ (i.e., the number of minterms that precede C in χ , plus 1). When the function f is clear from the context we shall write $rank(C)$ instead of $rank_f(C)$.

The algorithm GREEDY below examines the minterms of \mathcal{F} following an increasing order of their ranks. In the pseudo-code for GREEDY, we use C_i to denote the minterm C of \mathcal{F} such that $rank(C) = i$, according to Definition 1.

By the value of a minterm we mean the AND of the values of its variable. Therefore, the value of a minterm becomes known either when one of its variables has been found to have value 0 or, otherwise, when all of its variables have been found to have value 1.

We say that a minterm is *active* iff its value is not known yet. We say that a set U of unread variables is *strongly active* iff : (i) it is exactly the set of unread

variables of some active minterm of f and (ii) U is minimal, i.e., no proper subset of U satisfies condition (i). Note that U is always a minterm for f_Y , where Y is the set of the variables evaluated so far.

```

Algorithm GREEDY( $f, V, \mathbf{c}$ )
For  $i = 1 \dots |\mathcal{F}|$ 
    While the values of  $f$  is unknown and  $C_i$  is active
         $U \leftarrow$  a strongly active subset of  $C_i$ 
        Read a variable of  $U$ 
    End While
    If the value of  $f$  becomes known return
End For
    
```

We shall say that a minterm C_i is *evaluated* by GREEDY if and only if at least one variable of C_i is read during the i th iteration of the **For**-loop of GREEDY. Note that, according to this definition, it may happen that C_i is not evaluated although some of its variables are read before and after the i -th iteration of the **For**-loop.

Definition 2 (Implementations). *An implementation of GREEDY is a rule that defines both the strongly active set U contained in C_i and the variable of U to be selected.*

Lemma 3. *Let $f = f(x_1, \dots, x_n)$ be a monotone boolean function such that for each $i = 1, \dots, n$ the variable x_i appears in at most 3 minterms of f . Let \mathbb{G} be an arbitrary implementation of GREEDY. Then for every assignment σ such that $f(\sigma) = 1$, we have $c_{\mathbb{G}}^f(\sigma) \leq l(f)c^f(\sigma)$.*

Proof. Let $\mathcal{I} = \{C \mid C \text{ is evaluated by } \mathbb{G}\}$ and let \mathcal{I}' be a minimal subfamily of \mathcal{I} such that $\bigcup_{C' \in \mathcal{I}'} C' = \bigcup_{C \in \mathcal{I}} C$. Let C_{max} the minterm in \mathcal{I}' of maximum cost. We have the following.

Claim 1. $c_{\mathbb{G}}^f(\sigma) \leq |\mathcal{I}'|c^f(\sigma)$.

Because of the greedy choice employed by \mathbb{G} , we have that $c^f(\sigma) \geq c(C_{max})$. Thus, $c_{\mathbb{G}}^f(\sigma) \leq c(\bigcup_{C \in \mathcal{I}'} C) \leq \sum_{C \in \mathcal{I}'} c(C) \leq |\mathcal{I}'|c(C_{max}) \leq |\mathcal{I}'|c^f(\sigma)$.

Now, let \mathcal{F} be the family of minterms of f . Let $\mathcal{F}' \subseteq \mathcal{F}$ be the family of minterms of f which contain at least one variable that appears at most once in the minterms of \mathcal{I}' . Note that $\mathcal{I}' \subseteq \mathcal{F}'$. Let H' be the minimal hitting set for \mathcal{F}' obtained from the union of those variables. Thus $|H'| \geq |\mathcal{I}'|$.

Let $\overline{\mathcal{F}} = \mathcal{F} \setminus \mathcal{F}'$. By definition, every variable of a minterm in $\overline{\mathcal{F}}$ appears exactly twice in \mathcal{I}' . Note that we have $C \cap C' = \emptyset$ for every pair of different minterms $C, C' \in \overline{\mathcal{F}}$, for otherwise one of the variables would appear four times. A key claim for our proof is the following. Due to the space constraints, its proof is omitted in this extended abstract.

Claim 2. For each minterm $\overline{C} \in \overline{\mathcal{F}}$, there is a minterm $C' \in \mathcal{I}'$ such that $\overline{C} \cap C' \neq \emptyset$ and $c(C') \leq c(\overline{C})$.

Let $\mathcal{I}' = \{C_{i_1}, \dots, C_{i_{|\mathcal{I}'|}}\}$. By Claim 2, we can partition the family $\overline{\mathcal{F}}$ into $|\mathcal{I}'|$ subfamilies as follows. For $j = 1, 2, \dots, |\mathcal{I}'|$, let

$$\overline{\mathcal{F}}_j = \left\{ C \in \overline{\mathcal{F}} \setminus \left(\bigcup_{s=1}^{j-1} \overline{\mathcal{F}}_s \right) \mid C \cap C_{i_j} \neq \emptyset \text{ and } c(C) \geq c(C_{i_j}) \right\}.$$

Obviously $\overline{\mathcal{F}}_i \cap \overline{\mathcal{F}}_j = \emptyset$, for each $1 \leq i < j \leq |\mathcal{I}'|$, and, moreover, because of Claim 2, $\overline{\mathcal{F}} = \overline{\mathcal{F}}_1 \cup \dots \cup \overline{\mathcal{F}}_{|\mathcal{I}'|}$. For each $C \in \overline{\mathcal{F}}_j$, let $\text{var}(C)$ denote the variable of C with minimum index which also appears in C_{i_j} (the only utility of the index is to uniquely determine the variable). Let $J = \{j \mid \overline{\mathcal{F}}_j \neq \emptyset\}$ and let $\overline{H}_j = \{\text{var}(C) \mid C \in \overline{\mathcal{F}}_j\}$, for $j \in J$. Note that $H = H' \cup_{j \in J} \overline{H}_j$ is a hitting set for \mathcal{F} . Let H^f be a minimal hitting set for \mathcal{F} obtained by removing the redundant variables of H . Note that $H^f \cap \overline{H}_j = \overline{H}_j$ for every $j \in J$ and $\overline{H}_i \cap \overline{H}_j = \emptyset$ for every $i, j \in J$. Thus, $|H^f| = |H^f \cap H'| + \sum_{j \in J} |\overline{H}_j|$. Since \overline{H}_i covers at most $|\overline{H}_i| + 1$ minterms of \mathcal{I}' and $H^f \cap H'$ covers exactly $|H^f \cap H'|$ minterms, we must have $|H^f \cap H'| + \sum_{j \in J} (|\overline{H}_j| + 1) \geq |\mathcal{I}'|$. It follows that $|H^f| \geq |\mathcal{I}'| - |J|$, whence

$$l(f) \geq |\mathcal{I}'| - |J|.$$

For each $j \in J$, let C'_j be an arbitrary minterm in $\overline{\mathcal{F}}_j$. By definition, $c(C'_j) \geq c(C_{i_j})$ and since every variable of C'_j appears in exactly two minterms of \mathcal{I}' , we have

$$\begin{aligned} c_{\mathbb{G}}^f(\sigma) &\leq c\left(\bigcup_{j=1}^{|\mathcal{I}'|} C_{i_j}\right) \leq \sum_{j=1}^{|\mathcal{I}'|} c(C_{i_j}) - \sum_{j \in J} c(C'_j) \leq \sum_{j \in \{1, \dots, |\mathcal{I}'|\} \setminus J} c(C_{i_j}) \\ &\leq (|\mathcal{I}'| - |J|)c(C_{max}). \end{aligned}$$

Finally, using $l(f) \geq |\mathcal{I}'| - |J|$, we have the desired result $c_{\mathbb{G}}^f(\sigma) \leq l(f)c(C_{max}) \leq l(f)c^f(\sigma)$

Achieving the Best Extremal Competitive Ratio. Let us now consider the implementation $\mathbb{A}^{\mathbb{G}}$ of the general approach in which the set U_Y is defined as a strongly active set contained in the active minterm of f of minimum rank. It is not hard to verify that $\mathbb{A}^{\mathbb{G}}$ coincides with an implementation of GREEDY which selects the variables according to the rule employed in the general approach. Therefore, by Lemmas 2 and 3, we immediately get the following.

Theorem 2. *Let f be a monotone boolean function such that each one of its variables is in at most three minterms of f . Then for the implementation $\mathbb{A}^{\mathbb{G}}$ of the general approach, it holds that $c_{\mathbb{A}^{\mathbb{G}}}^f(\sigma) \leq \max\{k(f), l(f)\}c^f(\sigma)$ for every assignment σ .*

3.2 Threshold Trees

A threshold tree over a set of boolean variables V is a rooted tree T , where each internal node is associated to an integer number and each leaf is associated with a distinct variable of V . The value of a leaf is the value of its associated variable.

The value of a node whose associated integer is t (a t -node) is 1 if at least t of its children have value 1 and it is 0, otherwise. The boolean function computed by a threshold tree T is the one mapping the values of the leaves of T to the value of the root of T .

Given a threshold tree T , we use $leaves(T)$ to denote its set of leaves and $|T|$ to denote its number of leaves. Abusing notation, we use T to denote also the function, say f , computed by the tree T . Accordingly, for every $Y \subset V$, T_Y denotes both the threshold tree computing f_Y and the function f_Y itself.

The Certificates of a Threshold Tree. Let T be a threshold tree rooted on a t -node r and let T_1, \dots, T_p be the subtrees of T rooted at the children of r . Then, C is a minterm for T if and only if there exists a subset $R \subseteq \{1, \dots, p\}$, with $|R| = t$, such that: (i) $C \cap leaves(T_i)$ is a minterm for T_i , for $i \in R$; (ii) $C \cap leaves(T_i) = \emptyset$ for $i \notin R$. Analogously, C is a maxterm for T if and only if there exists a subset $S \subseteq \{1, \dots, p\}$, with $|S| = p - t + 1$, such that: (i) $C \cap leaves(T_j)$ is a maxterm for T_j , for $j \in S$; (ii) $C \cap leaves(T_j) = \emptyset$ for $j \notin S$.

Let π and π' be permutations of $\{1, \dots, p\}$ such that: $k(T_{\pi(i)}) \geq k(T_{\pi(i+1)})$ and $l(T_{\pi'(i)}) \geq l(T_{\pi'(i+1)})$, for $i = 1, \dots, p-1$. The characterization above implies

$$\text{that } k(T) = \sum_{i=1}^t k(T_{\pi(i)}) \quad \text{and} \quad l(T) = \sum_{i=1}^{p-t+1} l(T_{\pi'(i)}).$$

Since the functions that can be represented by threshold trees are monotone, then both Theorem 1 and Lemma 2 also hold for threshold trees. We shall now show that for the class of the threshold trees a result analogous to Lemma 3 holds, i.e., for every threshold tree function f and for each assignment σ s.t. $f(\sigma) = 1$, the algorithm GREEDY has the best possible competitive ratio. We shall need some preliminary results.

Definition 3 (Execution). Let \mathbb{I} be an implementation of GREEDY. For each function f and for each assignment σ , the execution $\mathbb{I}(f, \sigma)$ of the implementation \mathbb{I} of GREEDY on the function f with assignment σ is the sequence of pairs $(x_i, C(x_i))$, $i = 1, 2, \dots$ where x_i is the i -th variable that \mathbb{I} reads and $C(x_i)$ is the minterm of f that is being evaluated when x_i is probed.

The following lemma allows to evaluate the cost incurred by GREEDY on a threshold tree in a recursive way.

Lemma 4. Let \mathbb{I} be an arbitrary implementation of GREEDY. Let T_m be a subtree of T , rooted at one of the children of r . Let x_1, x_2, \dots, x_q be the leaves of T_m listed in the order that they appear in $\mathbb{I}(T, \sigma)$. Then, there exists an implementation \mathbb{I}_m for GREEDY that satisfies

(i) The q first variables of $\mathbb{I}_m(T_m, \sigma|_{T_m})$ are x_1, x_2, \dots, x_q

For $i = 1, 2, \dots, q$ let $C(x_i)$ (respectively $C_m(x_i)$) denote the minterm of T (resp. T_m) that is evaluated in $\mathbb{I}(T, \sigma)$ (resp. $\mathbb{I}_m(T_m, \sigma|_{T_m})$) when x_i is probed.

(ii) Then, for $i = 1, 2, \dots, q$, we have $C_m(x_i) = C(x_i) \cap T_m$.

Theorem 3. Let \mathbb{I} be an arbitrary implementation of GREEDY. If f can be represented by a threshold tree, then for every assignment σ such that $f(\sigma) = 1$, we have $c_{\mathbb{I}}^f(\sigma) \leq l(f)c^f(\sigma)$.

Proof. (Sketch) We prove by induction on the height of the tree that for every assignment σ' , the cost incurred by \mathbb{I} before determining the value of a minterm C is at most $l(f)c(C)$. This will suffice to establish the theorem. In fact, we can then consider the particular case where σ' is an assignment for which f evaluates to 1 and C is the cheapest proof for f under assignment σ' .

For the basis we assume that T has height 1, p leaves, and it is rooted at a t -node. Let $C = \{x_1, x_2, \dots, x_t\}$ be a minterm for T and let c_{max} be the cost of the variable with maximum cost of C . The key observation is that, because of the greedy way of selecting the minterms implemented by GREEDY no variable with cost larger than c_{max} is read before the value of C is determined.

Then, the cost incurred before the value of C is determined is at most $c(C) + (p - t)c_{max} \leq (p - t + 1)c(C) = l(T)c(C)$.

Let us assume that the claim holds for every threshold tree of height at most h . Let T be a tree with height $h + 1$ rooted at a t -node r . In addition, let T_1, \dots, T_p be the subtrees rooted at the children of r . We assume w.l.g. that $C \cap T_i \neq \emptyset$ for $i = 1, \dots, t$. This implies that $C \cap T_i = \emptyset$ for $i = t + 1, \dots, p$.

We have the following claim, whose proof is omitted from this extended abstract due to the space constraints.

Claim. Let C' be a minterm of T such that $rank(C') < rank(C)$. If C' is evaluated before the value of C is determined we must have

- (i) $c(C' \cap T_i) \leq c(C \cap T_i)$ for $i = 1, \dots, t$
- (ii) $c(C' \cap T_j) \leq \max_{i=1, \dots, t} \{c(C \cap T_i)\}$, for $j > t$.

Let X_i be the sequence of leaves of T_i that are in the execution $\mathbb{I}(T, \sigma)$ before determining the value of C , listed in the order in which they are read by \mathbb{I} . In order to bound the sum of the costs of these variables, we use the fact, assured by Lemma 4, that there is an implementation \mathbb{I}_i such that the sequence of variables in the execution $\mathbb{I}_i(T_i, \sigma|_{T_i})$ coincides exactly with X_i .

In fact, the second statement in Lemma 4 together with the previous claim guarantees that, for each $i \leq t$ (respectively $i > t$), \mathbb{I}_i only evaluates minterms of cost not larger than $c(C \cap T_i)$ (respectively c_{max}) while reading the variables in X_i . Thus, by induction hypothesis we have that the cost incurred due to the variables of T_i is at most $l(T_i)c(C \cap T_i)$ (respectively $l(T_i)c_{max}$).

Therefore, the cost spent by \mathbb{I} before determining the value of C is at most

$$\sum_{i=1}^t l(T_i)c(C \cap T_i) + \sum_{i=t+1}^p l(T_i)c_{max} \leq \left(\max_{i=1}^t l(T_i) + \sum_{i=t+1}^p l(T_i) \right) c(C) \leq l(T)c(C),$$

where the last inequality follows from the definition of $l(T)$.

Since \mathbb{A}^G is simultaneously a valid implementation for GREEDY and for the general approach, then putting together Theorem 3 and Lemma 2 we have the following result.

Theorem 4. *For every monotone boolean function f represented by a threshold tree, we have $\gamma^{\mathbb{A}^G}(f) = \max\{k(f), l(f)\} = \gamma(f)$.*

3.3 Game Trees

A game tree T is a tree, rooted at r , where every internal node has either a MIN or a MAX label and the parent of every MIN (MAX) node is a MAX (MIN) node. Let V be the set of leaves of T . Every leaf of V is associated with a real number, its value. The value of a MIN (MAX) node is the minimum (maximum) of the values of its children. The function computed by T (the value of T) is the value of its root. Like in the previous section we shall identify T with the function it computes. Thus, if f is the function computed by the game tree T , we shall also write T for f and T_Y for f_Y .

We extend the notion of maxterms and minterms to the case of a game tree. By a minterm (maxterm) of a game tree we shall understand a minimal set of leaves whose values allow to state a lower (upper) bound on the value of the game tree. More precisely, a minterm (maxterm) for a game tree T rooted at r is a minimal set C of leaves of T such that if $x(\sigma) \geq \ell$ ($x(\sigma) \leq \ell$), for each $x \in C$ then $r(\sigma) \geq \ell$ ($r(\sigma) \leq \ell$) regardless of the values of the leaves $y \notin C$. As with boolean functions, we shall use the more general term *certificate* to either refer to a minterm or to a maxterm.

We use \mathcal{F}_T^L and \mathcal{F}_T^U to denote the family of all minterms and the family of all maxterms of T , respectively.

For the game tree function $T = \max\{\min\{x_1, x_2\}, \min\{x_3, \max\{x_4, x_5\}\}\}$, we have $\mathcal{F}_T^U = \{\{x_1, x_3\}, \{x_1, x_4, x_5\}, \{x_2, x_3\}, \{x_2, x_4, x_5\}\}$ and $\mathcal{F}_T^L = \{\{x_1, x_2\}, \{x_3, x_4\}, \{x_3, x_5\}\}$.

The Certificates of a Game Tree. Let T_1, \dots, T_p be the subtrees of T rooted at the children of r . If r is a MAX node then C^L is a minterm for T if and only if C^L is also a minterm for some subtree T_i , with $i \in \{1, \dots, p\}$. Furthermore, C^U is a maxterm for T if and only if $C^U \cap T_i$ is a maxterm of T_i , for $i = 1, \dots, p$.

On the other hand, if r is a MIN node, then C^L is a minterm for T if and only if $C^L \cap T_i$ is a minterm of T_i , for $i = 1, \dots, p$. Finally, C^U is a maxterm for T if and only if C^U is also a maxterm for some subtree T_i , with $i \in \{1, \dots, p\}$.

We define the value $C(\sigma)$ of a minterm (maxterm) C w.r.t. assignment σ as the minimum (maximum) of the values of its leaves. In order to study the structure of a proof for T , it is useful to express the function computed by T in terms of its certificates as follows. $T(\sigma) = \min_{C^U \in \mathcal{F}_T^U} \{C^U(\sigma)\} = \max_{C^L \in \mathcal{F}_T^L} \{C^L(\sigma)\}$.

A proof for T , under an assignment σ , contains a minterm C^L and a maxterm C^U such that $C^U(\sigma) = C^L(\sigma) = T(\sigma)$.

Given a set of minterms (maxterms) for a Game tree T , the maximum (minimum) over the values of these minterms (maxterms) is a lower bound (upper bound) on the value of T .

In analogy with the treatment of the boolean functions, we shall use $k(T)$ and $l(T)$ to denote the largest minterm and maxterm of T , respectively. These quantities play a critical role in the following lower bound on the competitiveness of every algorithm that evaluates a game tree.

Theorem 5. [3] *Let T be a game tree. If each certificate of T has size at least 2 then $\gamma(T) \geq \max\{k(T), l(T)\}$.*

Consider a run of an algorithm for evaluating a game tree. Let Y be the set of leaves that have already been read. At this point of the execution, we know the value of T is in the interval $[LB, UB]$, with $LB < UB$. We observe that if none of the maxterms (mintervals) has been completely read then $UB = \infty$ ($LB = -\infty$). Otherwise, $UB(LB)$ is the minimum (maximum) among the values of the maxterms (mintervals) that have been fully read.

We say that a maxterm (mininterval) C is *active* if for each leaf $x \in C \cap Y$, we have $x(\sigma) < UB$ ($x(\sigma) > LB$). In words, a maxterm (mininterval) C is active if the evaluation of its unevaluated leaves can still lead to an improvement of the upper bound UB (lower bound LB), i.e., can provide additional information on the value of the game tree. On the other hand, if a maxterm (mininterval) is non-active it means that the evaluation of its unread leaves is not necessary to determine the value of T , since it will not affect the bound UB (LB). It must be observed that the definitions above imply that if all leaves of a certificate C have already been read, then C is non-active.

The GREEDY Algorithm for Game Trees. The algorithm GREEDY can be easily applied to Game Trees since, with the generalized notion of mintervals and active mintervals stated above, the definitions of *ranks* and strongly active sets naturally extend to Game Trees.

The following lemma gives an upper bound on the cost spent by GREEDY to prove that $f(\sigma) \geq B$, for each assignment σ and for each lower bound B . The proof of the lemma is based on an extension of Lemma 4 to the implementations of GREEDY for game trees. Due to the space limit the proof of Lemma 5 is omitted from this extended abstract.

Lemma 5. *Let \mathbb{I} be an arbitrary implementation of GREEDY. If f can be represented by a game tree, then for every assignment σ and for every B such that $f(\sigma) \geq B$, we have $c_{\mathbb{I}}^{f, B}(\sigma) \leq l(f)c_B^f(\sigma)$ where $c_{\mathbb{I}}^{f, B}(\sigma)$ denotes the cost spent by \mathbb{I} to find a certificate that $f(\sigma) \geq B$ and $c_B^f(\sigma)$ denotes the cost of the cheapest certificate that allows to proving that $f(\sigma) \geq B$.*

Theorem 6. *Let f be a function over the set of real variables V that can be represented by a game tree. Let \mathbb{G} be the implementation of the general approach that always chooses U_Y to be the same as the set U selected in the inner loop of GREEDY. Therefore, \mathbb{G} is also a valid implementation of GREEDY.*

For every assignment σ such that there is only one variable in V , say x , for which $x(\sigma) = f(\sigma)$ we have that $c_{\mathbb{G}}^f(\sigma) \leq \max\{k(f), l(f)\}c^f(\sigma)$.

Proof. Let $B = f(\sigma)$ and let C^- (C^+) cheapest mininterval (maxinterval) that certifies $f(\sigma) \geq B$ ($f(\sigma) \leq B$).

Claim $C^- \cup C^+$ is a cheapest proof for f under assignment σ .

Proof of the Claim Clearly, $C^- \cup C^+$ is a proof. Let us consider a proof P for f and let D^- (D^+) be a mininterval (maxinterval) contained in P that certifies $f(\sigma) \geq B$

$(f(\sigma) \leq B)$. Let x be the unique variable of V such that $x(\sigma) = B$. We have that $x \in D^- \cap D^+ \cap C^- \cap C^+$. In addition, since every pair of minterm and maxterm in a game tree has intersection of size 1 [3], we have $D^- \cap D^+ = C^- \cap C^+ = \{x\}$. Thus, it follows from the minimality of both C^+ and C^- that $c(C^- \cup C^+) = c(C^+) + c(C^-) - c(x) \leq c(D^- \cup D^+) = c(D^-) + c(D) - c(x) \leq c(P)$.

We split $c_{\mathbb{G}}^f(\sigma)$ into $cost_L$ and $cost_U$, where $cost_L$ is the cost incurred before \mathbb{G} proves that $f(\sigma) \geq B$ and $cost_U = c_{\mathbb{G}}^f(\sigma) - cost_L$ is the remaining cost. Since \mathbb{G} is an implementation of GREEDY it follows from Lemma 5 that $cost_L \leq l(T)c(C^-)$.

Now, we use the fact that \mathbb{G} is also an implementation of the general approach to prove $cost_U \leq k(T) \times c(C^+ \setminus C^-)$. Let Y be the set of variables that have already been read at the point where the lower bound B is determined. We have that $C^+ \setminus Y$ is a proof for the value of f_Y . Since $x \in Y$, then the cost of the cheapest proof for f_Y is at most $c(C^+ \setminus C^-)$. On the other hand, every set U_Y selected by \mathbb{G} must intersect every proof for f_Y . To see that, let C^* be the minterm for which U_Y is an active strong set and let z be the variable with minimum value in $C^* \cap Y$. If U_Y does not intersect a proof P for f_Y we could prove a lower bound larger than B by fixing all the variables of U_Y on $z(\sigma)$, which is a contradiction. Since $|U_Y| \leq k(f)$ always holds, then it follows from Lemma 1 that $cost_U \leq k(f) \times c(C^+ \setminus C^-)$. Thus, $c_{\mathbb{G}}^f(\sigma) = cost_U + cost_L \leq \max\{k(f), l(f)\}c(C^+ \cup C^-) = \max\{k(f), l(f)\}c^f(\sigma)$.

Due to the space constraints, the polynomial implementation of both the algorithm $\mathbb{A}^{\mathbb{G}}$ for threshold trees and the algorithm \mathbb{G} for game trees is omitted. The interested reader is referred to the full version of the paper.

References

1. R. Carmo, T. Feder, Y. Kohayakawa, E. Laber, R. Motwani, L. O’Callaghan, R. Panigrahy, and D. Thomas. Querying priced information in databases: the conjunctive case. 2004. submitted.
2. M. Charikar, R. Fagin, V. Guruswami, J. Kleinberg, P. Raghavan, and A. Sahai. Query strategies for priced information. *JCSS: Journal of Computer and System Sciences*, 64:785–819, 2002.
3. F. Cicalese and E. S. Laber. A new strategy for querying priced information. In *Proceedings of the 37th ACM Symposium on Theory of Computing*, Baltimore, 2005. ACM Press. to appear.
4. A. Gupta and A. Kumar. Sorting and selection with structured costs. In IEEE, editor, *42nd IEEE Symposium on Foundations of Computer Science*, pages 416–425, 2001.
5. A. Gupta, D. O. Stahl, and A. B. Whinston. Pricing of services on the internet. In *IMPACT: How IC2 Research Affects Public Policy and Business Markets*. Quorum Books, forthcoming.
6. V. Guruswami and E. Laber. Personal communications. 2003.
7. S. Kannan and S. Khanna. Selection with monotone comparison costs. In *Proceedings of the fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA-03)*, pages 10–17, 2003.

8. Karchmer, Linial, Newman, Saks, and Wigderson. Combinatorial characterization of read-once formulae. *Discrete Mathematics*, 114, 1993.
9. S. Khanna and W. Tan. On computing functions with uncertainty. In *Symposium on Principles of Database Systems*, pages 171–182, 2001.
10. E. Laber. A randomized competitive algorithm for evaluating priced AND/OR trees. In *STACS: Annual Symposium on Theoretical Aspects of Computer Science*, pages 501–512, 2004.
11. E. Laber, O. Parekh, and R. Ravi. Randomized approximation algorithms for query optimization problems on two processors. In *ESA: Annual European Symposium on Algorithms*, pages 649–661, 2002.

Online View Maintenance Under a Response-Time Constraint*

Kamesh Munagala, Jun Yang, and Hai Yu

Department of Computer Science, Duke University,
Durham, NC 27708-0129, USA
{kamesh, junyang, fishhai}@cs.duke.edu

Abstract. A *materialized view* is a certain synopsis structure precomputed from one or more data sets (called *base tables*) in order to facilitate various queries on the data. When the underlying base tables change, the materialized view also needs to be updated accordingly to reflect those changes. We consider the problem of batch-incrementally maintaining a materialized view under a *response-time constraint*. We propose techniques for *selectively* processing updates to some base tables while keeping others batched, with the goal of minimizing the total maintenance cost while meeting the response-time constraint. We reduce this to a generalized paging problem, where the cost of evicting a page is a concave non-decreasing function of the number of continuous requests seen since the last time it was evicted. Our main result is an online algorithm that achieves a constant competitive ratio for all concave cost functions while relaxing the response-time constraint by a constant factor. For several special classes of cost functions, the competitive ratio can be improved with simpler, more intuitive algorithms. Our algorithms are based on emulating the behavior of an online paging algorithm on a page request sequence carefully designed from the cost function. The key novel technical ideas are twofold. The first involves discretizing the cost function, so that there is a collection of periodic paging sequences, with page sizes decreasing geometrically, which approximates the behavior of the original function. The second involves designing an online view maintenance algorithm based on the paging process, by emulating the behavior of the paging scheme in recursively defined phases.

1 Introduction

A *materialized view* is a certain synopsis structure precomputed from one or more data sets (called *base tables*) in order to facilitate various queries on the data [8]. Materialized views have a wide range of traditional and new applications, such as data warehousing, database caching, continuous queries, and publish/subscribe systems, just to name a few. Since a materialized view is a form of derived data, which is computed from the underlying base tables, it needs to be *refreshed* if the base tables change. Instead of recomputing the view from scratch in order to refresh it, we can incrementally maintain the view, i.e., compute and apply only the incremental changes to the view given the base table updates. Furthermore,

* Research is supported in part by NSF CAREER award under grant IIS-0238386.

for many applications, incremental maintenance does not have to be performed eagerly for each base table update; instead, the system can defer maintenance until the view content needs to be accessed. Hence, the system maintains the view in a *batch incremental* fashion: Base table updates are accumulated into a batch and then processed together when needed.

The cost of processing the updates is typically a concave non-decreasing function of the number of updates in the batch. In other words, processing a batch of updates is usually more efficient than processing them one at a time. Therefore, batch incremental maintenance can be used to improve efficiency for many applications where deferred view maintenance is acceptable. For example, recent publish/subscribe systems, e.g., OpenCQ [11], NiagraCQ [4], Xyleme [13], all provide a feature that allows subscribers to specify a *notification condition* in addition to the subscription content query. Examples include: “*report mean and median house prices in North Carolina once every 200 house sales*”, or “*notify me with a detailed view of my portfolio whenever the price of a stock in my portfolio has changed by more than 5 percent since the last notification*”. Only when the notification condition is met, the system needs to compute updates to the subscribed content (which can be regarded as a materialized view) and notify the subscriber.

At the same time, in many such applications, it is often desirable to provide a quality-of-service guarantee in the form of a *response-time constraint*. That is, whenever the content of a materialized view is requested, the system should be able to refresh the view under a prescribed time limit. This constraint prevents the system from deferring view refresh indefinitely; if the batch of unprocessed base table updates becomes too big, it may be impossible to process the batch in time upon request. This paper addresses the problem of maintaining a materialized view under a response-time constraint, with the goal of minimizing the total cost of view maintenance over time.

Work in [9] mainly considers the offline version of the problem, where the system has some knowledge of the arrival sequence of future base table updates. In this paper, we propose online algorithms that are constant-competitive against any adversary, assuming the response-time constraint is relaxed by a constant factor. We show that this problem is an interesting generalization of paging with concave cost functions; this connection is of independent interest. We need new ideas to extend paging algorithms to concave functions, as we point out below.

Problem Statement. Formally, we want to incrementally maintain a view defined over n base tables, v_1, v_2, \dots, v_n . The cost of refreshing the view by performing x units of updates to base table v_i is $f_i(x)$. Each $f_i(x)$ is a concave, non-decreasing function.¹ By appropriate scaling, we enforce $f_i(1) \geq 1$ for

¹ Strictly speaking, these cost functions are *subadditive*; that is, processing $x + y$ updates together cannot be more expensive than processing x updates in one batch and then y updates in another, since the option of doing the latter is still available given all $x + y$ updates. We note that results for subadditive and concave functions are equivalent within a factor of 2, so the same algorithms can still apply in the subadditive case.

each i . At any time instant t , an update vector $\langle x_{1t}, x_{2t}, \dots, x_{nt} \rangle$ arrives online, where x_{it} is an integer denoting the number of updates to base table v_i . Let $\langle X_{1t}, X_{2t}, \dots, X_{nt} \rangle$ denote the total updates pending for each of these base tables respectively after the update vector at the current step arrives. The algorithm can now *schedule* a vector $\langle Y_{1t}, Y_{2t}, \dots, Y_{nt} \rangle$ of updates on the base tables, and incur an *update cost* of $C_t = \sum_{i=1}^n f_i(Y_{it})$, and carry forward the remaining updates as pending to the next time step. Let $\langle Z_{1t}, Z_{2t}, \dots, Z_{nt} \rangle$ denote the vector of updates carried forward, or pending. Note the relations $Z_{it} = X_{it} - Y_{it}$ and $X_{it} = x_{it} + Z_{i,t-1}$ hold for all v_i and all t . There is a bound H on the total update cost that can be pending at any point of time, which we call the *pending update cost constraint*. That is, the constraint on the algorithm is $\sum_{i=1}^n f_i(Z_{it}) \leq H$ for all t .

The goal is to design an online algorithm to minimize the total update cost $\sum_t C_t$, when the update vectors $\langle x_{1t}, x_{2t}, \dots, x_{nt} \rangle$ arrive online at every time step.

This problem generalizes the problem of paging with arbitrary page sizes, where the goal is to minimize the total cost of page faults, or equivalently, the total size of pages evicted (under the BIT model of paging [10]). Given a paging problem with n pages, where page p_i has size s_i (an integer), and a cache with size k , the equivalent view maintenance problem has base table v_i for page p_i , with update cost $f_i(x) = s_i$ independent of x , and $H = k$. The constraint that the pending update cost is at most H translates exactly to the cache not overflowing. The update cost at any time step is precisely the size of the pages evicted at that step.

Our Results. Our main result in Section 4 presents a constant-competitive online algorithm for the view maintenance problem, with a constant-factor relaxation in the pending update cost constraint. Before that, in Section 2, we present a very simple $O(\log H)$ -competitive algorithm, with the pending update cost constraint being relaxed to $8H$. In Section 3, we describe a constant-competitive algorithm for a natural special case, which forms the basis for the more general constant-competitive algorithm of Section 4. Our algorithms are deterministic and work against adaptive adversaries. The competitive ratio is essentially best possible (to within constant factors) if the response-time constraint is allowed to be relaxed by a constant factor.

Our main idea is to emulate the behavior of an online paging algorithm on an appropriately defined paging sequence, and use this behavior to guide the view maintenance algorithm. We first convert an online sequence of updates for the original problem into a page request sequence. Next, we run the online paging algorithm in [5,16], with the best known competitive ratio of $O(1)$ on this page request sequence using a cache of size $2H$. We then convert the resulting online paging scheme back into an online scheme for view maintenance.

There are constant-competitive paging algorithms known for arbitrary page sizes and eviction costs [5,10,16] when the cache size is relaxed. Although we emulate paging on a suitable page request sequence, and hence use these paging algorithms as subroutines, this is in no way straightforward. The main problem

is the concave nature of the eviction cost. If we try defining multiple pages with decreasing eviction costs to model concavity, we run into the problem that their evictions have to be correlated: the cheaper page cannot be retained in the cache by the paging algorithm if the more expensive page has been evicted. A problem with using a paging algorithm *as is* is that it becomes non-trivial to convert the behavior of the paging algorithm into a well-defined view maintenance scheme. We show how to tackle both these issues by carefully constructing the paging sequence after discretizing the concave cost function, and by grouping the page evictions into recursive phases and performing view maintenance depending on the behavior of the paging algorithm in each phase. Both these details are non-trivial, requiring new ideas, and are expounded in Section 4.

All our algorithms need to relax the response-time constraint by a constant factor. An interesting open question is to decide the complexity of the problem (especially with respect to oblivious adversaries) if the response-time constraint is not relaxed. We also note that our results can be made bicriteria on the competitive ratio and the pending update cost relaxation in a fairly straightforward manner.

Related Work. The classical online paging problem with unit sized pages is well studied in the offline [2] and the online settings [6,12,14].

When the pages have arbitrary sizes and eviction cost proportional to size (assuming the smallest page has size 1), the offline paging problem becomes NP-Hard, and the best known polynomial-time approximation algorithm achieves a factor of $O(\log k)$ to the optimal cost [10]. This shows that the offline view maintenance problem is NP-Hard (using the reduction above), with the best approximation algorithm achieving a factor of $O(\log H)$. For the online version of the problem, LRU is k -competitive, and is the best possible deterministic paging scheme [10]. Randomized marking algorithms perform much better; there is a $O(\log^2 k)$ -competitive paging scheme against an oblivious adversary, due to Irani [10].

The paging problem can be further generalized to allow both the sizes of the pages and the cost for evicting the pages to be arbitrary. In the offline setting, Albers *et al.* [1] obtained a constant-approximation algorithm that uses an additional amount of memory of size $O(1)$ times the largest page size. In the online setting, Cao and Irani [3] generalized a greedy-dual algorithm of Young [15] and showed that this deterministic online algorithm is k -competitive. Young [16] and Cohen and Kaplan [5] proved that it is $h/(h-k+1)$ -competitive, by running the same algorithm on a cache of size $h \geq k$. We use this algorithm as our paging subroutine for $h = 2k$ (meaning we double the cache size used by the offline algorithm in order to be 2-competitive). We note that since the best possible competitive ratio for the paging problem is constant when the cache size is relaxed by a constant factor, the same guarantee is a lower bound for the more general view maintenance problem.

2 Simple $O(\log H)$ -Competitive Algorithm

We now show a simple randomized reduction of the online view maintenance problem to the online paging problem. The reduction can be easily made deter-

ministic, but randomization slightly simplifies our analysis. The resulting algorithm achieves a competitive ratio of $O(\log H)$ against an oblivious adversary, provided that the pending update cost constraint can be relaxed to $8H$. This section provides the background for the more involved algorithms in later sections.

The reduction is accomplished in two steps. At the first step, we “translate” an online sequence of updates for the view maintenance problem into a page request sequence, and run a competitive online paging algorithm on this page request sequence. At the second step, we “translate” the resulting online paging scheme back into an online scheme to the original problem.

For each base table v_i , we assume f_i is a continuous function. If not, the proofs below go through at the loss of an additional constant factor. Let $r_0 = 1$, and r_1, \dots, r_h be the values of update size such that $f_i(r_j) = 2^j f_i(r_0)$. Further, $f_i(r_h) \geq H$, but $f_i(r_{h-1}) < H$, which implies $h \leq \log H$. Corresponding to each r_j , we have a page p_{ij} of size $f_i(r_j)$.

For an online view maintenance problem, we generate an instance of the online paging problem as follows. The size of the cache is $2H$. Whenever one unit of update is input for base table v_i , for each j , we request for page p_{ij} with probability $\min(1, 1/r_j)$. These requests are independent from one unit of update to the next, even within the same time step. Let OPT denote the cost of the optimal offline view maintenance algorithm on a certain update sequence. Our analysis relies on the following lemma.

Lemma 1 ([9]). *There exists a view maintenance scheme such that for all i and t , if $Y_{it} > 0$, then $Z_{it} = 0$, and whose total update cost is at most $2 \cdot \text{OPT}$.*

Intuitively, this lemma implies that one can find a 2-competitive view maintenance scheme so that whenever any update is processed for a base table, the entire pending updates for that base table are processed. The proof of the above lemma proceeds by modifying an optimal offline scheme to process all pending updates of a base table whenever it processes any update of that base table, so that any update operation in the original scheme is charged by at most two update operations in the modified scheme. Details can be found in [9].

Lemma 2. *There is a paging scheme for the page request sequence whose expected cost is $O(\log H) \cdot \text{OPT}$, and which uses $2H$ amount of cache space.*

Proof. We will argue the bound for a certain base table v_i , since the update costs for different base tables are additive. Consider the view maintenance scheme in Lemma 1. Consider a time interval $[t_1, t_2]$ such that $Z_{i,t_1-1} = 0$, $Z_{it_2} = 0$, and $Z_{it} > 0$ for all $t \in [t_1, t_2)$. Note that $Z_{it} = \sum_{t'=t_1}^t x_{it'}$, and at time t_2 the view maintenance scheme processes all X_{it_2} pending updates to base table v_i . We will pretend $Z_{it_2} = X_{it_2}$ in the proof below to unify notation.

At any time $t \in [t_1, t_2]$, we allocate space $2f_i(Z_{it})$ to the paging algorithm for caching a subset of the pages p_{i0}, \dots, p_{ih} . More precisely, let p_{ij} be the largest page whose size is at most $f_i(Z_{it})$; we will cache $p_{i0}, p_{i1}, \dots, p_{ij}$ in this space at time t . Since the page sizes scale by a factor of 2, this space allocated is

sufficient to cache all these pages. If there is a page request for a page of size larger than $f_i(Z_{it})$ at time t , we do not cache this page, but evict it as soon as it is encountered, leading to a page fault. At time t_2 , we evict all pages p_{ij} in the cache.

Consider a page p_{ij} which is in the cache at time t_2 . We brought this page into cache at time t when $Z_{it} \geq r_j$ and $Z_{i,t-1} < r_j$. In the interval $[t_1, t - 1]$, the expected number of times page p_{ij} is requested is at most $Z_{i,t-1}/r_j \leq 1$, which means the expected cost for these page faults is at most $f_i(r_j)$. After time t , there are no additional page faults on p_{ij} , until we evict it at time t_2 . Let k denote the largest such j , i.e., $f_i(r_k) \leq f_i(Z_{it_2}) < 2f_i(r_k)$. The total expected cost of all page faults for $j \leq k$ in the interval $[t_1, t_2]$ is then at most $2 \sum_{j \leq k} f_i(r_j) \leq 4f_i(r_k)$, where there is a factor of 2 since we evict all pages p_{i0}, \dots, p_{ik} at time t_2 .

Consider now a page p_{ij} which is not in the cache at time t_2 . Hence $Z_{it_2} < r_j$. The expected number of times page p_{ij} is requested in the time interval $[t_1, t_2]$ is at most Z_{it_2}/r_j . Thus the expected cost of page faults due to this page is at most $(Z_{it_2}/r_j) \cdot f_i(r_j) \leq f_i(Z_{it_2}) < 2f_i(r_k)$, where the first inequality holds because $f_i(x)$ is a concave function. Therefore, the total cost of these page faults is at most $O(\log H) \cdot f_i(r_k)$, as there are at most $\log H$ such pages.

We have thus shown that the paging scheme pays cost at most $O(\log H) \cdot f_i(r_k)$ in the interval $[t_1, t_2]$, while the view maintenance scheme pays cost $f_i(Z_{it_2}) \geq f_i(r_k)$ at time t_2 . Therefore, overall, the cost of the paging scheme is at most $O(\log H) \cdot \text{OPT}$.

Note that it does not help the adversary to inject updates costing more than H at time t_2 , since doing so would result in the same cost for both. □

The above lemma shows that there is a paging scheme whose cost is $O(\log H)$ -competitive with respect to the optimal offline scheme of the view maintenance problem. The online algorithm for the view maintenance problem generates the random page request sequence, and runs the constant-competitive paging algorithm [5,16] on this sequence using a cache of size $4H$. At any time t , let W_{it} be the size of the largest page corresponding to base table v_i in cache; the algorithm allocates cost at most $2W_{it}$ to the pending updates of v_i . There are two situations where we process all pending updates of v_i : (1) if the total cost of these updates becomes larger than $2W_{it}$; and (2) if the paging algorithm evicts the largest page currently in its cache corresponding to v_i .

Lemma 3. *The online view maintenance algorithm is constant-competitive against the cost of the corresponding online paging scheme.*

Proof. Consider two consecutive time instances t_1 and t_2 when the online view maintenance algorithm processes updates of base table v_i . If the reason for processing updates at time t_2 is because the largest page was evicted from the cache, the cost of the update can be accounted for by the cost of the page evicted (whose size is at least $f_i(X_{it_2})/2$). If the reason is that $f_i(X_{it_2})$ exceeds twice the size of the largest page, we charge the cost of the update to the present or next instant when a page of largest size less than $f_i(X_{it_2})$ is requested (and subsequently

evicted); note that the size of this page is at least $f_i(X_{it_2})/2$. Since the behavior of the paging scheme is independent of the distribution of future page requests, the expected number of such charges made to any page is at most one. Therefore, overall, the expected competitive ratio of the online view maintenance algorithm is 2 against the online paging algorithm. \square

The above two lemmas immediately imply the following theorem (noting that we lose a factor of 2 in the pending update cost due to the previous lemma).

Theorem 1. *There is an $O(\log H)$ -competitive online algorithm for the view maintenance problem that relaxes the pending update cost constraint to $8H$.*

3 Improved Algorithms for Special Cases

For several important special cases of cost functions, we can obtain simpler and more intuitive algorithms with better performance guarantees. Here we consider the following three cases: (1) $f_i(x) = \min(a_i x, b_i)$; (2) $f_i(x) = a_i(x - 1) + b_i$; and (3) $f_i(x) = \min(a_i(x - 1) + b_i, c_i)$.

The first case, $f_i(x) = \min(a_i x, b_i)$, can arise in the following situation. The cost of processing a batch initially increases linearly with the size of the batch. When the size of the batch reaches a certain point, however, it becomes more efficient to simply recompute the view, whose dominating cost becomes independent of the batch size. The second case, $f_i(x) = a_i(x - 1) + b_i$, can arise if update processing incurs a fixed amount of startup cost. Details for these two cases will appear in the full paper.

In the remainder of this section, we focus on the third case, $f_i(x) = \min(a_i(x - 1) + b_i, c_i)$. This special case is a generalization of the first two cases, and serves as a preparation for our subsequent discussions in Section 4. The paging sequence that we construct for this case is deterministic and *periodic*. Let us assume c_i is a multiple of b_i , which can be enforced at a loss of factor of 2 in the competitive ratio. We have $k = \frac{c_i}{b_i}$ types of pages corresponding to base table v_i . Let us denote them $p_{i1}, p_{i2}, \dots, p_{ik}$. We assume $k > 4$; the case for smaller k is simple to deal with, and is therefore omitted.

We maintain two counters, c and r , which are initialized to 0 and 1 respectively. Whenever there is a unit update received for the base table v_i , we request p_{ir} and increment c . If $c \geq \frac{b_i}{a_i}$, we set $c \leftarrow 0$, and $r \leftarrow r \bmod k + 1$. The size of the cache is H . As before, we can prove the following emulation result.

Lemma 4. *There is a paging scheme for the page request sequence whose cost is at most $2 \cdot \text{OPT}$.*

The online view maintenance algorithm runs the constant-competitive paging algorithm on this page request sequence using a cache of size $2H$. If the paging algorithm caches a total size x of pages corresponding to base table v_i , the algorithm allocates at most $6x$ pending update cost to v_i . The algorithm proceeds in phases. A phase corresponds to the “period” of the request sequence, that is,

the time interval in which all k distinct pages are requested. At the beginning of each phase, all pending updates of v_i are *untagged*, and if the number of pages corresponding to v_i in the cache is less than $k/2$, the algorithm processes all these untagged updates to v_i . During the phase, for each received update of v_i , suppose page p_{ir} is requested and is in or being brought into the cache. The algorithm keeps the corresponding update pending and *tags* this update to that page. If the page is not cached, this update is processed immediately. When a page is evicted, the updates tagged to that page are processed. If the number of cached pages is larger than $k/2$ at the beginning of the phase, but drops below $k/4$ sometime during the phase, the algorithm processes all untagged pending updates to v_i . Finally, all pending updates (either tagged or untagged) are reset to untagged at the end of the phase before entering the next phase.

Theorem 2. *The online algorithm has a competitive ratio of $O(1)$, while relaxing the pending update cost to $12H$.*

Proof. We lose a factor of 2 in pending update cost upfront simply because the online paging algorithm uses cache $2H$. First note that the total cost of updates that can be tagged to any page is at most $2b_i$, so that the cost of processing these tagged updates can be charged to the eviction of the corresponding page.

Secondly, if there are untagged updates pending, the total cost allocated to the updates of v_i is at least c_i . To verify this claim, observe that the untagged updates must have been from the previous phase, and their presence indicates that the number of pages in the cache must be at least $k/4$, since if there were either less than $k/2$ pages at the beginning of the phase, or less than $k/4$ pages sometime during the phase, these updates would have been processed then. With a factor 4 relaxation in the pending updates cost constraint, the algorithm would allocate cost at least c_i for these updates. The total factor of 6 in the relaxation comes from the sum of the factor 4 for the untagged updates, and the factor of 2 for the tagged updates.

Next, we observe that whenever the algorithm is forced to process all the pending updates (by spending cost of at most c_i) at the beginning of a phase, the number of pages cached is at most $k/2$, which means that the number of pages requested in the phase that will not be present in the cache is at least $k/2$. The cost of fetching these pages (and subsequently evicting them) is at least $c_i/2$. Therefore, the update cost can be charged to the cost of evicting these pages.

If the untagged pending updates are processed during a phase, the number of pages evicted by the algorithm since the beginning of the phase is at least $k/4$, and the cost of these evictions is at least $c_i/4$. The cost for processing the updates can therefore be charged to the cost of evicting these pages. \square

4 Constant-Competitive Algorithm

In this section, we present an algorithm for general concave functions $f_i(x)$, which achieves a constant competitive ratio, with a constant factor relaxation in the

pending update cost constraint. The algorithm is based on the periodic paging sequences constructed in Section 3, combined with an emulation in recursive phases. We first need to discretize the cost function so that a phase-by-phase accounting of cost can be performed. An idea similar to the following lemma appeared in Guha *et al.* [7], and is key to the design of the algorithm. We note that most realistic cost functions would satisfy this lemma upfront.

Lemma 5. *The function $f_i(x)$ can be approximated to a constant factor by a piecewise linear concave function $g_i(x)$ that connects by line segments consecutive points (c_r, d_r) ($c_1 = 1$), so that the points satisfy:*

1. d_{r+1} is a multiple of d_r ;
2. $d_{r+1} \geq 2d_r$;
3. $\frac{d_r}{c_r} \geq 2\frac{d_{r+1}}{c_{r+1}}$.

Proof. Let $b = f_i(1)$. Consider those points on the curve $f_i(x)$ with y coordinates $b, 2b, 4b, \dots$, and denote them by (c_r, d_r) , $r \geq 1$. We scan these points in decreasing order of y coordinates. We connect the closest two (in terms of r value) such points whose d_r/c_r values differ by at least a factor of 2 and ignore the intermediate points. The curve ends at $(1, b)$. This new curve $g_i(x)$ is a 2-approximation to the original curve. To see why, consider two points (c_j, d_j) and (c_k, d_k) that are connected (see Figure 1 (a)). For any $x \in [c_{j+1}, c_k]$, by concavity we have $f_i(x) \leq xd_{j+1}/c_{j+1} \leq x(2d_k/c_k)$, and $g_i(x) \geq xd_k/c_k$. For any $x \in [c_j, c_{j+1}]$, we have $f_i(x) \leq f_i(c_{j+1}) = 2b_j$ and $g_i(x) \geq b_j$. Therefore $g_i(x) \geq f_i(x)/2$.

Note that the third condition may not be true at $r = 1$. This problem can be fixed by first approximating $f_i(x)$ by an additional constant factor and then applying the above procedure. We omit the details here. □

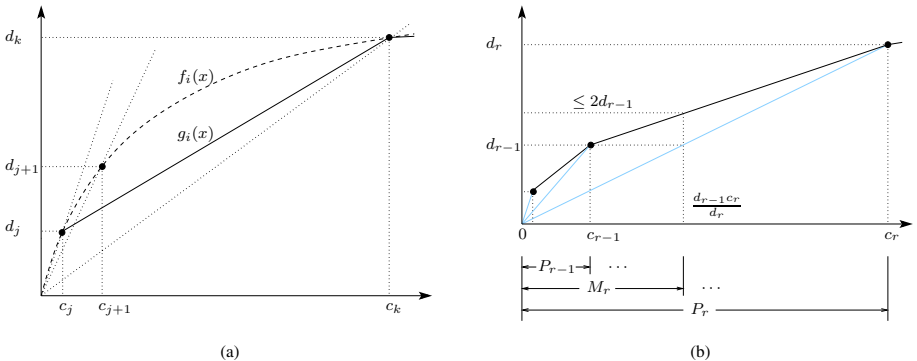


Fig. 1. (a) Approximating $f_i(x)$ by a piecewise linear concave function. (b) Phases and mini-phases.

In the subsequent discussions, we pretend that $f_i(x) = g_i(x)$. Let (c_r, d_r) be the set of non-smooth points on the curve. For convenience, we further assume $\frac{d_r}{c_r}$ is a multiple of $\frac{d_{r+1}}{c_{r+1}}$; the proof remains the same even if it is not. We have a

paging sequence for each “level” r . For $r \geq 1$, we have c_r pages of size $\frac{d_r}{c_r}$. We request a different one every unit update in a cyclic fashion, so that the same page is requested after exactly c_r updates. Let us denote a complete cycle of pages at level r by P_r . Note that at $r = 1$, this process requests the same page of size d_1 every unit update. Note also that this process requests many pages for the same unit update, but these page sizes decrease by a factor of at least 2, so that they sum to at most $2d_1$.

Lemma 6. *There is a paging scheme for the page request sequence that is constant-competitive against the optimal offline view maintenance algorithm, provided that the cache size is $4H$.*

Proof. The paging scheme emulates the view maintenance scheme of Lemma 1 in the following way. Let $[t_1, t_2)$ be a time interval during which the view maintenance scheme continuously keeps updates to base table v_i pending. At any time, the paging scheme caches all pages requested so far corresponding to v_i . The total size of the cached pages is at most four times the pending update cost of the view maintenance scheme at that time instant. To see why, consider the time when x updates have been pending, and suppose x lies in between c_r and c_{r+1} . The paging scheme has cached the entire set of pages for all levels up to r . The total size of these pages is at most $\sum_{i=1}^r d_i \leq 2d_r \leq 2f_i(x)$. For levels greater than r , the total size cached is at most $\sum_{i>r} x \cdot d_i/r_i \leq 2x \cdot d_{r+1}/c_{r+1} \leq 2f_i(x)$. All cached pages corresponding to base table v_i are evicted at time t_2 , whose cost can be accounted for by the cost of the view maintenance scheme for processing all the pending updates of v_i at that time. □

As before, we construct the page request sequence and run the constant-competitive online algorithm on the sequence using a cache of size $8H$. We now show how to convert the paging scheme into an online algorithm for the view maintenance problem. The emulation again proceeds in phases as in Section 3, but now consists of recursive phases. Phase J_{rg} marks the end of the g -th complete cycle of the request sequence P_r . We call the value of r the “level” of the phase. Each J_{rg} is composed of consecutive mini-phases M_{rl} , each of which has exactly as many level- r pages as to make the total size exactly d_{r-1} . Note that a mini-phase is composed of many consecutive periods of P_{r-1} . Let B_{rl} be the set of distinct level- r pages corresponding to M_{rl} . Note that $B_{rx} = B_{ry}$ if $y \equiv x \pmod{d_r/d_{r-1}}$. The idea behind defining a mini-phase M_{rl} is that the total cost of all updates in this mini-phase is at most $2d_{r-1}$; see Figure 1 (b).

For every unit update, if the $r = 1$ level page is not in the cache, the algorithm processes the corresponding update, else the algorithm pends it. This mechanism corresponds to the base case level $r = 1$. For larger r , the algorithm for the phases is more complicated (see Figure 2). Consider any level r , and a corresponding phase J_{rg} . Suppose the time step is currently in this phase, and in mini-phase M_{rl} . This mini-phase is composed of many sub-phases $J_{r-1,x}$. Suppose the current time step is in phase $J_{r-1,x}$. All updates which arrived within J_{rg} but in $M_{r'l}$ for $l' < l$ are tagged to $B_{r'l'}$. Updates arriving in $J_{r-1,x'}$

for $x' < x$, but within M_{rl} are tagged to the set P_{r-1} . This tagging scheme is recursively maintained at all levels r .

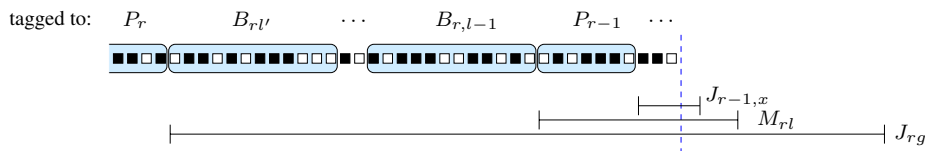


Fig. 2. The recursive tagging scheme. The solid squares represent yet unprocessed (or pending) updates, and the empty squares represent processed updates.

At the end of mini-phase M_{rl} , all pending updates tagged to P_{r-1} are now tagged to B_{rl} . Intuitively, the reason for doing so is that we can no longer afford to tag more updates to P_{r-1} , because otherwise the total cost of these updates may exceed $2d_{r-1}$, while the total size of the pages in P_{r-1} is only d_{r-1} ; so we re-tag these updates one level up to B_{rl} , in order to “free” P_{r-1} for tagging future updates.

At the end of every mini-phase M_{rl} , for all $l' \leq l$, if the size of pages from $B_{rl'}$ present in the cache is at most $d_{r-1}/2$, the algorithm processes the updates tagged to $B_{rl'}$. During any mini-phase M_{rl} , if the size of pages for any $B_{rl'}$ for $l' < l$ is at least $d_{r-1}/2$ at the beginning of the phase, but drops below $d_{r-1}/4$ at the current time step, the algorithm also processes all updates tagged to $B_{rl'}$.

At the end of J_{rg} , all updates tagged to each B_{rl} are now tagged to P_r . At the end of J_{rg} , if the size of pages from P_r present in the cache is at most $d_r/2$, the algorithm processes all updates tagged to P_r . If during J_{rg} , this size falls below $d_r/4$, but was larger than $d_r/2$ at the beginning of the phase, the algorithm also processes the pending updates tagged to P_r .

Lemma 7. *The total cost of pending updates is at most 8 times the total size of pages at any point during the execution of the algorithm.*

Proof. Any update is tagged either to a P_r or to a B_{rl} for some r and some l . The maximum cost of updates that can be tagged to a P_r before it passes on to a $B_{r+1,l}$ is at most $2d_r$. The maximum cost of updates that can be tagged to a B_{rl} is at most $2d_{r-1}$. We have also maintained the invariant that if the total size of pages in the cache corresponding to P_r is less than $d_r/4$, or those corresponding to B_{rl} is less than $d_{r-1}/4$, there are no updates tagged to these sets. Therefore, the cost of the pending updates can be charged to the size of the pages in the cache. \square

Lemma 8. *The cost of processing the pending updates is at most 12 times the cost of evictions of the corresponding page sets to which they are tagged, implying the algorithm is constant-competitive against the cost of the online paging scheme.*

Proof. The proof of this claim uses exactly the same argument as the one given in the proof of Theorem 2 in the previous section, charging the update cost to the cost of page evictions in each phase. \square

Theorem 3. *The online view maintenance algorithm is constant-competitive, with an $O(1)$ relaxation in the pending update cost constraint.*

References

1. S. Albers, S. Arora, and S. Khanna. Page replacement for general caching problems. In *Proc. 10th ACM-SIAM Sympos. Discrete Algorithms*, pages 31–40, 1999.
2. L. Belady. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5:78–101, 1966.
3. P. Cao and S. Irani. Cost-aware www proxy caching algorithms. In *Proc. USENIX Symposium on Internet Technologies and Systems*, pages 193–206, 1997.
4. J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagraCQ: A scalable continuous query system for internet databases. In *Proc. 19th ACM SIGMOD Intl. Conf. Management of Data*, pages 379–390, 2000.
5. E. Cohen and H. Kaplan. LP-based analysis of greedy-dual size. In *Proc. 10th ACM-SIAM Sympos. Discrete Algorithms*, pages 879–880, 1999.
6. A. Fiat, R. Karp, M. Luby, L. McGeoch, D. Sleator, and N. Young. Competitive paging algorithms. *J. Algorithms*, 12:685–699, 1991.
7. S. Guha, A. Meyerson, and K. Munagala. Hierarchical placement and network design problems. *Proc. 41st IEEE Sympos. Foundations of Comput. Sci.*, pages 603–612, 2000.
8. A. Gupta and I. S. Mumick, editors. *Materialized Views: Techniques, Implementations and Applications*. MIT Press, June 1999.
9. H. He, J. Xie, J. Yang, and H. Yu. Asymmetric batch incremental view maintenance. In *Proc. 21st Intl. Conf. Data Engineering*, pages 106–117, 2005.
10. S. Irani. Page replacement with multi-size pages and applications to web caching. *Algorithmica*, 33(3):384–409, 2002.
11. L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *IEEE Trans. Knowledge and Data Engineering*, 11(4):610–628, 1999.
12. L. McGeoch and D. Sleator. A strongly competitive randomized paging algorithm. *Algorithmica*, 6:816–825, 1991.
13. B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda. Monitoring XML data on the web. In *Proc. 20th ACM SIGMOD Intl. Conf. Management of Data*, pages 437–448, 2001.
14. D. Sleator and R. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, 1985.
15. N. Young. The k -server dual and loose competitiveness for paging. *Algorithmica*, 11:525–541, 1994.
16. N. Young. On-line file caching. *Algorithmica*, 33:371–383, 2002.

Online Primal-Dual Algorithms for Covering and Packing Problems

Niv Buchbinder and Joseph Naor*

Computer Science Department, Technion, Haifa 32000, Israel
{nivb, naor}@cs.technion.ac.il

Abstract. We study a wide range of online covering and packing optimization problems. In an online covering problem a linear cost function is known in advance, but the linear constraints that define the feasible solution space are given one by one in an online fashion. In an online packing problem the profit function as well as the exact packing constraints are not fully known in advance. In each round additional information about the profit function and the constraints is revealed. We provide general deterministic schemes for online *fractional* covering and packing problems. We also provide deterministic algorithms for a couple of *integral* covering and packing problems.

1 Introduction

We study a wide range of online covering and packing optimization problems. In an “offline” (fractional) covering problem the objective is to minimize the total cost given by a linear cost function $\sum_{i=1}^n c(i)x(i)$. The feasible solution space is defined by a set of m linear constraints of the form $\sum_{i=1}^n a(i, j)x(i) \geq b(j)$, where the entries $a(i, j)$ and $b(j)$ are non negative.

The *general online fractional covering problem* is an online version of the covering problem, described as a game between an algorithm and an adversary. In this setting the cost function is known in advance, but the linear constraints that define the feasible solution space are given to the algorithm one by one in an online fashion. In order to maintain a feasible solution to the current set of given constraints, the algorithm is allowed to augment the variables $x(i)$. It may not, however, decrease any previously augmented variable. We also extend our study to cases where the value of each variable has an upper bound $u(i)$, referred to as a *box constraint*. The box constraints are known to the online algorithm in advance. This captures online settings in which the amount of resources is limited. As usual, the performance of an online algorithm on a given input is defined to be the ratio between the total cost of its solution and the minimal (optimal) cost of any solution for the given instance. The maximum ratio, taken over all input sequences, is defined to be the *competitive ratio* of the algorithm. Our setting generalizes the fractional graph optimization problems discussed in [1,2]. There, the constraints are of the form $\sum_{i=1}^n a(i, j)x(i) \geq 1$ with each

* Research supported in part by US-Israel BSF Grant 2002276.

coefficient $a(i, j) \in \{0, 1\}$ and no upper bounds on the values of the variables are given.

We show that the online covering problem is closely related to a dual online packing problem. In an offline packing problem we are given n packing constraints of the type $\sum_{j=1}^m a(j, i)y(j) \leq c(i)$. The goal is to find a feasible solution that maximizes a profit function $\sum_{i=1}^m y(j)$. The *general online fractional packing problem* is an online version of this problem. In the online setting the values $c(i)$ ($1 \leq i \leq n$) are known in advance, but the profit function and the exact packing constraints are not known in advance. In the j th round a new variable $y(j)$ is introduced to the algorithm, along with its set of coefficients $a(i, j)$ ($1 \leq i \leq n$). Note that other variables that were not yet introduced may also appear in the packing constraints. This means that each packing constraint is revealed to the algorithm gradually. The algorithm may increase the value of a variable $y(j)$ only in the round where it is given and may not decrease or increase the values of any previously given variables. The performance of the algorithm is measured with respect to the maximum possible profit of the packing problem instance.

Although this online setting seems a bit unnatural, it generalizes the well known (fractional) problems of throughput-competitive online routing of virtual circuits problem and online load balancing [4]. In the (integral) online virtual circuits routing problem the goal is to maximize the number of calls accepted. The corresponding packing constraints are given by the limited edge capacities in the network. Note that in this case all the coefficients $a(i, j)$ associated with some call j are either the bandwidth of the current call (for an edge on the route) or zero otherwise.

Online algorithms for fractional problems may in some cases yield good online algorithms for their corresponding online integral versions. Such an approach is applicable when the fractional solution can be rounded into an integral solution in an online fashion. This usually results in randomized algorithms that may sometimes be derandomized to achieve a deterministic online algorithm for the integral problem. The idea of rounding fractional solutions in an online fashion was demonstrated in [2], yielding randomized online algorithms for many interesting cases. The same idea was implicitly used in [1] to achieve a deterministic algorithm for the online set-cover problem.

Previous Work: Covering and packing optimization problems, as well as their online versions, have been studied extensively. Yet, in most cases only the integral version of the problem was considered. Examples of such online integral covering optimization problems are the online steiner problem that was considered in [10] and the generalized steiner problem that was considered in [3,5]. Recently, Alon et al. [2] suggested a general two-phase approach for a wide class of online network optimization problems. First, a fractional solution to the online problem is generated. The solution is then rounded in an online fashion in the second phase. The two phases, although separated, are run simultaneously. In [2], the first phase is performed by a general $O(\log n)$ -competitive method which is applicable to a wide class of online graph optimization problems. The rounding phase, on the other hand, is problem dependent. The approach was shown useful

by the development of randomized algorithms for many interesting integral graph optimization problems. Examples are the online group steiner problem and the online non-metric facility location. The same approach was also implicitly used in [1], where a deterministic online $O(\log n \log m)$ -competitive algorithm for the integral set cover problem was provided. Our methods for generating fractional solutions for online covering problems generalize and improve upon the methods of [2]. An integral online packing problem was considered in [4]. They considered the problem of throughput-competitive online routing of virtual circuits.

There is a long line of work on generating a near-optimal fractional solution for *offline* covering and packing problems, e.g. [14,16,9,7,11]. Generating such a solution for the *offline* covering problem with upper bounds on the variables was considered in [6,8]. All these methods take advantage of the offline nature of the problems. Offline randomized rounding of covering and packing problems, along with a derandomization method, was considered in [13,12]. A slightly better rounding was provided in [15].

Results: We study the *general online fractional covering* problem with and without box constraints and the *general online fractional packing* problem. We show that the problems are closely related and provide general deterministic primal-dual schemes that compute a near-optimal fractional solution for each problem. In Section 3 we suggest a scheme for the *general online fractional packing*. Let n denote the number of packing constraints. The scheme gets the desired competitive ratio $B > 0$ and produces a solution that does not violate any of the packing constraints by more than a function of $1/B$ (see theorem 1 for the exact expression). In particular, when all coefficients $a(i, j) \in \{0, 1\}$, we show an $O(\log n)$ -competitive algorithm that does not violate any of the constraints. We also prove tight lower bounds on any online algorithm for the problem, proving that our scheme is optimal for any $B > 0$.

The scheme for the covering problem is very similar. In fact, when each coefficient $a(i, j) \in \{0, 1\}$, the same scheme is applicable for both problems. Unfortunately, the above scheme is not suitable for the general online fractional covering problem where the coefficients $a(i, j) \geq 0$ are not limited to $\{0, 1\}$. For this problem we design a more complicated scheme in Section 4. For any $B > 0$, our scheme is $O(\frac{\log n}{B})$ -competitive and covers each constraint up to a factor of $1/B$ (i.e. $\sum_{i=1}^n a(i, j)x(i) \geq \frac{1}{B}$). We also extend the scheme to handle box constraints, where we are given an upper bound on the value of each variable. A simple modification of the lower bounds in [2] proves our scheme to be tight.

The online fractional covering problem with $a(i, j) \in \{0, 1\}$ was studied previously in the context of online graph optimization problems [1,2]. The scheme we propose for this case is simpler than the algorithms of [1,2], since we do not need to guess the value of the optimum solution, and thus avoid the need for phases. More importantly, our scheme is more precise, and thus improves the competitive ratio of the algorithm. Specifically, we get a competitive ratio of $O(\log d)$ instead of $O(\log n)$, where d is the maximum number of variables in each given constraint ($d \leq n$). This improvement immediately reflects on the competitive ratio of any of the corresponding integral problems considered in

[1,2]. The competitive ratio for the online set-cover improves to $O(\log d \log n)$, where d is the maximum number of sets that an element appears in. In the fractional connectivity problems considered in [2], we improve the $O(\log m)$ factor, where m is the number of edges, to $O(\log C)$, where C is the size of the maximum cut in the graph. For certain integral versions of the problems considered in [2] the competitive ratio can be further improved. For instance, in the online group steiner problem we improve one of the $\log m$ factors to $\log |g|$, where $|g|$ is the maximum size of a group. Similarly, in the fractional cuts problems considered in [2] we improve the $O(\log m)$ factor to $O(\log L)$, where L is the length of the longest simple path in the graph.

In Section 5 we show the applicability of our schemes for solving their corresponding integral versions via randomized rounding. To do so, we use the schemes for the fractional cases as a “black box”. We focus on converting the randomized algorithm we obtain into a deterministic algorithm. To this end, we suggest a method for rounding the fractional solution deterministically by transforming an (offline) pessimistic estimator into an online potential function. We note that, in general, the existence of an offline derandomization method does not necessarily yield an online deterministic algorithm. In particular, while all randomized rounding methods in [2] can be derandomized offline, it is an open question whether online deterministic algorithms for these problems exist. Still, the perspective of deterministic rounding motivates us to consider derandomization methods that were previously suggested for various offline problems, and these enable us to obtain better deterministic online algorithms.

We demonstrate our derandomization method on two online problems, one covering problem and one packing problem. We first consider the online unweighted set-cover problem [1] in Section 5.1. For this problem we provide a better deterministic algorithm in terms of competitive ratio. We draw ideas from the improved offline randomized rounding and derandomization methods of [15] to come up with an improved potential function for the online problem. This yields a competitive ratio of $O(\log d \log \frac{n}{OPT})$ instead of $O(\log d \log n)$, where OPT is the optimal number of sets needed to cover the requested elements. The improvement in the competitive ratio is significant when the number of sets needed for the cover is large, and in particular when the sets are small. Due to space limitations we omit all proofs.

Finally, in Section 5.2 we consider the problem of throughput-competitive online routing of virtual circuits [4]. We show that a deterministic algorithm equivalent to the one from [4] can be derived easily by combining our schemes with the method of conditional expectations of the randomized rounding algorithm presented in [13]. This is an interesting way of viewing the algorithm of [4], which also provides a systematic method for deriving it. In particular, we note that our scheme produces in an online fashion a near-optimal *fractional solution* to the problem that does not violate the capacity constraints. Producing such a fractional solution, unlike an integral solution, is applicable to any values of edge capacities. We also note that our scheme may be used to deal with a more general setting of the problem. See section 5.2 for more details.

2 Preliminaries

In this section we formally define our problems and discuss their dual nature. In an “offline” (fractional) covering problem the objective is to minimize the total cost given by a linear cost function $\sum_{i=1}^n c(i)x(i)$. The feasible solution space is defined by a set of m linear constraints of the form $\sum_{i=1}^n a(i, j)x(i) \geq b(j)$, where the entries $a(i, j)$ and $b(j)$ are non negative. Given an instance of a covering problem we may first normalize each constraint to the form: $\sum_{i=1}^n a(i, j)x(i) \geq 1$. By the duality theorem, any primal covering instance has a corresponding dual packing problem that provides a lower bound on any feasible solution to the instance. A general form of a (normalized) primal covering problem along with its dual packing problem is given in Figure 1.

The *general online fractional covering problem* is an online version of the covering problem. In this setting the cost function is known in advance, but the linear constraints that define the feasible solution space are given to the algorithm one-by-one. In order to maintain a feasible solution to the current set of given constraints, the algorithm is allowed to augment the variables $x(i)$. It may not, however, decrease any previously augmented variable. We also define an online version of the packing problem. In the *general online fractional packing problem* the values $c(i)$ ($1 \leq i \leq n$) are known in advance. However, the profit function and the exact packing constraints are not known in advance. In the j th round a new variable $y(j)$ is introduced to the algorithm, along with its set of coefficients $a(i, j)$ ($1 \leq i \leq n$). Note that other variables that were not yet introduced may also appear in the packing constraints. This means that each packing constraint is revealed to the algorithm gradually. The algorithm may increase the value of a variable $y(j)$ only in the round where it is given, and may not decrease or increase the values of any previously given variables.

We observe that these two online settings form a primal-dual pair in the following sense: At any time an algorithm for the general online fractional covering problem maintains a subset of the final linear constraints. This subset defines a *sub-instance* of a final covering instance. The dual packing problem of this sub-instance is a *sub-instance* of the final dual packing problem. In the dual packing sub-instance only part of the dual variables are known along with their corresponding coefficients. The two sub-instances form a primal-dual pair. In each round of the general online fractional covering problem a new constraint on the feasible solution space is given. The primal covering sub-instance is updated by adding the new constraint. To update the dual sub-instance we add a new dual variable to the profit function along with its coefficients that are defined by the

Primal (Covering)	Dual (Packing)
Minimize: $\sum_{i=1}^n c(i)x(i)$	Maximize: $\sum_{j=1}^m y(j)$
Subject to:	Subject to:
For each $1 \leq j \leq m$: $\sum_{i=1}^n a(i, j)x(i) \geq 1$	For each $1 \leq i \leq n$: $\sum_{j=1}^m a(i, j)y(j) \leq c(i)$
For each $1 \leq i \leq n$: $x(i) \geq 0$	For each $1 \leq j \leq m$: $y(j) \geq 0$

Fig. 1. Primal (covering) and dual (packing) problems

new primal constraint. Note that the dual update is the same as in the setting of the online fractional packing problem.

The schemes we propose maintain at each step primal and dual solutions for these primal-dual sub-instances. When a new constraint is given to the scheme for the online fractional covering problem, it also considers the new *corresponding* dual variable and its coefficients. When the scheme for the online fractional packing problem receives a new variable along with its coefficients, it also considers the *corresponding* new constraint in the primal sub-instance. In the rest of the paper we use the notion of primal and dual sub-instances and the notion of corresponding dual variable and primal constraint.

3 The General Online Fractional Packing Problem

In this section we describe an online scheme for computing a near-optimal fractional solution for the general online fractional packing problem. The scheme gets the desired competitive ratio $B > 0$ and returns a solution within B of the optimal that does not violate the packing constraints by too much. The scheme simultaneously maintains primal (covering) and dual (packing) solutions for the primal and dual sub-instances.

Initially, each variable $x(i)$ is initialized to zero. In each round a new variable $y(j)$ is introduced along with its coefficients $a(i, j)$ ($1 \leq i \leq n$). In the corresponding primal sub-instance a new constraint is introduced of the form $\sum_{i=1}^n a(i, j)x(i) \geq 1$. This constraint is non empty, since otherwise, it means that there is no bound on the value of $y(j)$ and the profit function is unbounded. The algorithm increases the value of the new variable $y(j)$ and the values of the primal variables $x(i)$ until the new primal constraint is satisfied. The augmentation method is described here in a continuous fashion, but it is not hard to implement the augmentation in a discrete way in any desired accuracy. In our continuous description the variables $x(i)$ behave according to a monotonically increasing function of $y(j)$. To implement the scheme in a discrete fashion, one should find the minimal $y(j)$ such that the new primal constraint is satisfied. Note that any variable $y(j)$ is being increased only in the j th round and the values of the primal variables never decrease. A single round is described by the following scheme. The performance of the scheme is analyzed in theorem 1.

1. $y(j) \leftarrow 0$; For each $x(i)$: $a_i(\max) \leftarrow \max_{k=1}^j \{a(i, k)\}$.
2. While $\sum_{i=1}^n a(i, j)x(i) < 1$:
 - (a) Increase $y(j)$ continuously.
 - (b) Increase each variable $x(i)$ by the following increment function:

$$x(i) \leftarrow \max \left\{ x(i), \frac{1}{na_i(\max)} \left[\exp \left(\frac{B}{2c(i)} \sum_{k=1}^j a(i, k)y(k) \right) - 1 \right] \right\}$$

Theorem 1. *For any $B > 0$, the scheme is B -competitive algorithm for the general online fractional packing problem and for any constraint:*

$$\sum_{k=1}^m a(i, k)y(k) = c(i) \cdot O\left(\frac{\log n + \log \frac{a_i(\max)}{a_i(\min)}}{B}\right)$$

Where, $a_i(\max) = \max_{k=1}^m \{a(i, k)\}$ and $a_i(\min) = \min_{k=1}^m \{a(i, k) | a(i, j) \neq 0\}$.

Corollary 1. *When $a(i, j) \in \{0, 1\}$ there exists an $O(\log n)$ -competitive algorithm for the fractional packing problem that does not violate the constraints.*

Remark 1. When all entries $a(i, j) \in \{0, 1\}$, we do not need to update the value $a_i(\max)$ in line 2(b) and the scheme is simplified. In this case, if we know in advance that each primal constraint consists of at most ℓ non-zero coefficients, it is possible to improve the competitive factor from $\log n$ to $\log \ell$. This is done by replacing the value n in line 2(b) by ℓ . Moreover, in this case, the same scheme is applicable to the online fractional covering problem (with $a(i, j) \in \{0, 1\}$). The improvement of the competitive ratio to $\log \ell$ immediately reflects on the competitiveness of several corresponding integral covering problems. See Results Section for more details.

Remark 2. The algorithm can handle an unbounded number of dual variables in each iteration, if there exists an oracle that finds at each iteration a variable whose primal constraint is not satisfied, or states that there is no such variable.

3.1 Lower Bounds

In this section we state two lower bounds that show that our scheme is optimal up to constants.

Lemma 1. *There is an instance of the general fractional packing problem with a single constraint such that $\sum_{j=1}^m a(i, j)y(j) \geq c \frac{H(a(\max)/a(\min))}{B}$ for any online B -competitive algorithm. $H(n)$ is the n th harmonic number, and $a(\max)/a(\min)$ is the ratio between the maximal and the minimal value in the constraint.*

Lemma 2. *There is an instance of the general online fractional packing problem with n constraints and $a(i, j) \in \{0, 1\}$, such that for any B -competitive online algorithm there exists a constraint such that $\sum_{j=1}^m a(i, j)y(j) \geq c(i) \frac{\log n}{2B}$.*

4 The General Online Fractional Covering Problem

In this section we describe our online scheme for computing a near-optimal fractional solution for the online fractional covering problem. As stated in Remark 1, in case the coefficients $a(i, j) \in \{0, 1\}$ the scheme in section 3 is also applicable for the online fractional covering problem. Unfortunately, the scheme is not applicable to the general online fractional covering problem, where the coefficients $a(i, j) \geq 0$ are not limited to $\{0, 1\}$. This happens since the scheme described in Section 3 does not always produce a feasible dual solution that can bound the primal solution efficiently. In this section we design a more complicated scheme for the general online fractional covering problem.

Primal (Covering)	Dual (Packing)
Minimize: $\sum_{i=1}^n c(i)x(i)$	Maximize: $\sum_{j=1}^m y(j) - \sum_{i=1}^n u(i)z(i)$
Subject to:	Subject to:
$\forall j : 1 \leq j \leq m: \sum_{i=1}^n a(i,j)x(i) \geq 1$	$\forall i : 1 \leq i \leq n: \sum_{j=1}^m a(i,j)y(j) - z(i) \leq c(i)$
$\forall i : 1 \leq i \leq n: 0 \leq x(i) \leq u(i)$	$\forall i, j: y(j), z(i) \geq 0$

Fig. 2. Primal (covering) with box constraints and its dual (packing) problem

Our scheme for the general online fractional covering problem gets a parameter $B > 0$. With $B > 0$ the competitive ratio of the scheme is $O(\frac{\log n}{B})$ and for each constraint $\sum_{i=1}^n a(i, j)x(i) \geq \frac{1}{B}$. The scheme works in phases: When the first constraint is introduced the scheme generates a first lower bound $\alpha(1) \leftarrow \frac{1}{B} \min_{i=1}^n \{c(i)/a(i, 1)\} \leq \frac{OPT}{B}$. It runs with the lower bound $\alpha(r)$ on the optimum as long as the total primal cost does not exceed the value $\alpha(r)$. When the cost exceeds this value the scheme “forgets” about all the values given to the primal and dual variables so far, updates the value of α by doubling it, and starts a new phase with $\alpha(r + 1) \leftarrow 2\alpha(r)$. Nevertheless, the values of the “forgotten” variables are accounted in the total cost of the solution. In the following we describe one round of our scheme in the r th phase. Let $\sum_{i=1}^n a(i, j)x(i) \geq 1$ be the new primal constraint that was introduced. Let $y(j)$ be the new corresponding dual variable. The scheme increases the values of the primal and dual variables as described by the following scheme. Note that during each phase $x(i)$ only increases. The performance of the scheme is analyzed in theorem 2.

1. $y(j) \leftarrow 0$
2. While $\sum_{i=1}^n a(i, j)x(i) < \frac{1}{B}$:
 - (a) increase $y(j)$ continuously.
 - (b) Increase each variable $x(i)$ by the following increment function:

$$x(i) \leftarrow \frac{\alpha(r)}{2nc(i)} \exp\left(\frac{\log 2n}{c(i)} \sum_{k=1}^j a(i, k)y(k)\right)$$

Theorem 2. For any $B > 0$, the scheme for the general online fractional covering problem achieves a competitive ratio of $O(\frac{\log n}{B})$, such that for each constraint $\sum_{i=1}^n a(i, j)x(i) \geq \frac{1}{B}$.

Adding Box Constraints: We now extend our methods to handle upper bounds on the variables $x(i)$. Let $u(i)$ be the upper bound on the variable $x(i)$. Such an upper bounds may result in an instance of the covering problem with no feasible solution. We next sketch the main ideas and changes that are needed in order to deal with the upper bounds. Adding box constraints to a covering problem results in new negative variables $z(i)$ in the dual program. The primal covering with box constraints and the new corresponding dual program are described in Figure 2. The performance of the scheme is analyzed in theorem 3.

Theorem 3. For any $B > 0$, the scheme for the general online fractional covering problem with box constraints achieves a competitive ratio of $O(\frac{\log n}{B})$, such that for each constraint $\sum_{i=1}^n a(i, j)x(i) \geq \frac{1}{B}$ and for each variable $x(i) \leq \frac{u(i)}{B}$

Our proposed scheme works in phases, where each phase has an upper bound on the total cost. When some variable reaches a value of $\frac{u(i)}{B}$, we start augmenting its corresponding dual variable $z(i)$. The augmentation of $z(i)$ causes the increment of the primal variable $x(i)$ to stop. Let X be the set of *tight variables* with value $\frac{u(i)}{B}$. If all variables in some unsatisfied constraint are tight, the scheme returns - "No feasible solution". A single round during the r th phase is described in the following:

1. $y(j) \leftarrow 0$
2. While $\sum_{i=1}^n a(i, j)x(i) < \frac{1}{B}$:
 - (a) Increase $y(j)$ continuously.
 - (b) For each variable $x(i) \in X$ increase $z(i)$ with rate $a(i, j)y(j)$.
 - (c) Augment each variable $x(i)$ by the following increment function:

$$x(i) \leftarrow \min\left\{\frac{u(i)}{B}, \frac{\alpha(r)}{2nc(i)}\right\} \exp\left(\frac{\log 2n}{c(i)} \sum_{k=1}^j a(i, k)y(k) - z(i)\right)$$

5 Integral Online Problems: Derandomization

In this section we show the applicability of our fractional schemes for solving online integral problems. To do so, we use the schemes as "black boxes" and deterministically round the fractional solutions obtained. The deterministic rounding is obtained by transforming an (offline) pessimistic estimator into an online potential function. This was done implicitly in [1]. The existence of an offline derandomization method does not necessarily yield an online deterministic algorithm. For example, finding deterministic algorithms for the problems considered in [2] is still an open problem. (They all have offline derandomizations.) Thus, we note that the ability to transform a (offline) pessimistic estimator into an online potential function is quite surprising.

We demonstrate our derandomization method on two online problems. We first consider the online unweighted set-cover problem [1]. For this problem we provide a better deterministic algorithm in terms of competitive ratio. To do so, we draw on ideas from the improved offline rounding and derandomization methods of [15] to generate an improved potential function. We then consider the online problem of throughput-competitive routing of virtual circuits. We show that the algorithm presented in [4] can be derived by a deterministic rounding of the fractional solution produced by our packing scheme.

This approach provides us with the following insight to the competitive factors obtained by [4]. The $O(\log m)$ competitive factor follows from an online generation of a fractional solution. The minimum edge capacity determines the

scaling of the fractional solution that is needed in order to guarantee high probability of success in the rounding phase. We note that producing a near-optimal fractional solution that does not violate the capacity constraints is applicable with any values of edge capacities.

5.1 Improved Competitive Ratio for Online Unweighted Set Cover

We define the online unweighted set cover problem as follows. Let $X = \{e_1, \dots, e_n\}$ be a ground set of n elements, and let \mathcal{S} be a family of subsets of X , $|\mathcal{S}| = m$. A *cover* is a collection of sets such that their union is X . In an online setting the elements are given one-by-one. Once a new element is given, the algorithm has to cover it by some set of \mathcal{S} containing it. Denote by $X' \subseteq X$ the set of elements given by the adversary. Our assumption is that the set cover instance, i.e., X and \mathcal{S} , is known in advance. The objective is to minimize the total number of sets chosen by the algorithm. Let \mathcal{C} denote the family of sets in \mathcal{S} that the algorithm chooses, and let C denote the set of elements covered by sets belonging to \mathcal{C} .

Note, first, that we may assume, by doubling, that the cardinality of OPT is known up to a factor of 2. Indeed, we can start guessing $OPT = 1$, and run the algorithm with this value of the optimal solution. If it turns out that the cardinality of the optimal solution is already at least twice our current guess for it, (that is, the cost of \mathcal{C} exceeds $\Theta(OPT \log d \log(n/OPT))$), then we can “forget” all sets chosen so far to \mathcal{C} , update the value of OPT by doubling it, and keep going. It can be easily verified that this only multiply the competitive ratio by constants. Next, we transform the pessimistic estimator that appear in [15] into the following online potential function used by our algorithm:

For each element e_i ($1 \leq i \leq n$): $f(e_i) = \min \left\{ 1, \exp \left(-\alpha + \alpha \sum_{s|e_i \in s} w(s) \right) \right\}$

$$\Phi = \left[1 - \prod_{e_i | e_i \notin C} (1 - f(e_i)) \right] + \exp \left(\sum_{s \in \mathcal{S}} (\ln 2 \chi_{\mathcal{C}}(s) - \alpha w(s)) - OPT \right)$$

The function $\chi_{\mathcal{C}}(s) = 1$ if $s \in \mathcal{C}$, and $\chi_{\mathcal{C}}(s) = 0$ otherwise. The parameter $\alpha = O(\log(n/OPT))$ will determine our competitive ratio. The first term of the potential function ensures that each element that was given to the algorithm is covered. The second term is used to bound the cardinality of the solution.

Using the above potential function the online algorithm is simple: Run the algorithm presented in Section 3 to produce a fractional solution. Note that since all coefficients $a(i, j) \in \{0, 1\}$, the simpler scheme in section 3 is applicable. When the weight of some set is augmented, add the set to the cover \mathcal{C} , only if by adding it the potential function decreases. We prove claim 5.1 on the properties of the potential function and then prove our main lemma (lemma 3).

Claim. The potential function Φ satisfies the following properties:

1. At start $\Phi < 1$, and at any time during the run of the algorithm $\Phi > 0$.
2. When a set is augmented either taking it to \mathcal{C} or excluding it does not increase Φ .

$$\begin{array}{l|l}
 \text{Minimize: } \sum_{e \in E} u(e)Y(e) + \sum_{c_i} U(c_i) & \text{Maximize: } \sum_{c_i} \sum_{P \in P(c_i)} f(c_i, P) \\
 \text{Subject to:} & \text{Subject to:} \\
 \forall P \in P(c_i): \sum_{e \in P} Y(e) + U(c_i) \geq 1 & \forall \text{ client } c_i: \sum_{P \in P(c_i)} f(c_i, P) \leq 1 \\
 & \forall \text{ edge } e: \sum_{P|e \in P} \sum_{c_i} f(c_i, P) \leq u(e)
 \end{array}$$

Fig. 3. The routing problem (Maximize) and its corresponding primal problem

Lemma 3. *The online algorithm satisfies the following:*

1. *Each element that was given to the algorithm is covered.*
2. *The cardinality of \mathcal{C} is at most $OPT \cdot O(\log d \log(n/OPT))$.*

5.2 Throughput-Competitive Online Routing of Virtual Circuits

The online problem of maximizing the throughput of scheduled virtual circuits was considered in [4]. In its simplest version we are given in advance a graph with capacities $u(e)$ on the edges. A set of clients $c_i = (s_i, t_i)$ ($1 \leq i \leq n$) arrive in an online fashion. To serve a client, the algorithm chooses a path between s_i and t_i and allocates a bandwidth of 1 on this path. The total bandwidth allocated on any edge may never exceed its capacity. The total profit of the algorithm is the number of clients served and its performance is measured with respect to the maximum number of clients that could have received service.

In a fractional version of the problem the allocation is not restricted to integral bandwidth of either zero or one, instead we can allocate to each client a fractional bandwidth in the range $[0, 1]$. In addition, the bandwidth allocated to a client can be divided between several paths. This is an online version of maximum multicommodity flow. We describe the problem as a packing problem in Figure 3. For $c_i = (s_i, t_i)$ let $P(c_i)$ be the set of simple paths between s_i and t_i . The first set of constraints ensures that each client gets at most a fractional flow (bandwidth) of 1. The second set of constraints are the capacity constraints of the edges. In the primal problem we assign a variable $U(c_i)$ to each client c_i and a variable $Y(e)$ to each edge in the graph. The problem is a special case of the general online fractional packing problem with $a(i, j) \in \{0, 1\}$. We note that our methods can generate a fractional solution to an extension of the problem to any non-negative coefficients $a(i, j)$. The extension can be viewed as giving each edge a different capacity with respect to the clients that are using it. This models a more complex relationship between the clients and the network capacities.

An algorithm for the problem was proposed in [4]. We provide here an alternative equivalent algorithm. We build our algorithm systematically by using our two phase approach. First, we use our scheme to generate a *feasible* fractional solution which is $O(\log P(\max))$ far from the optimum, where $P(\max)$ is the length of the longest path in the graph. Next, we convert the standard pessimistic estimator designed for the offline version of the problem into an online potential function. A randomized method to achieve a feasible integral solution is to scale down all the flows by some factor $B \geq 1$. We then choose to route the flow on a path P , carrying fractional flow of $f(P)$, with probability $f(P)/B$. When B is large enough, there is a positive probability that no edge capacity

is violated, and the total integral flow is large enough [13]. This algorithm was derandomized using the method of conditional expectations via a pessimistic estimator [12]. We transform this pessimistic estimator into an online potential function. When the flow on a path to a client is increased, the algorithm serves the client on that path if by doing so the potential function does not increase. Our online potential function Φ is the sum of the following functions:

$$\Phi_1 = \frac{1}{2} \exp \left\{ \ln 2 \left[\sum_{c_i} \sum_{P \in P(c_i)} \frac{f(c_i, P)}{B} - 2 \sum_{c_i} \chi(c_i) \right] \right\}$$

$$\Phi_2 = \frac{1}{2n} \sum_{e \in E} \exp \left\{ \left(1 + \frac{1 + \ln n}{u(e)} \right) \chi(e) - \sum_{P|e \in P} \sum_{c_i} f(c_i, P) \right\}$$

Where, $\chi(c_i)$ is the characteristic function of c_i (i.e. $\chi(c_i) = 1$ iff it is serviced), and $\chi(e)$ is the total number of paths that use the edge. Choosing $B = \exp\left\{\left(1 + \frac{1 + \ln n}{u(\min)}\right)\right\} - 1$, where $u(\min)$ is the minimal capacity of an edge, suffices to prove claim 5.2 that is used to prove lemma 4.

Claim. Initially, $\Phi < 1$ and throughout the algorithm, $\Phi > 0$. In addition, each time a flow on some path is increased, then either serving the client on this path or doing nothing reduces the potential function.

Lemma 4. *The algorithm does not violate the capacity constraints and accepts a least $O\left(\frac{1}{B \log P(\max)}\right)OPT$ clients, where $B = \exp\left\{\left(1 + \frac{1 + \ln n}{u(\min)}\right)\right\} - 1$*

Acknowledgements. We thank Yossi Azar for many helpful discussions.

References

1. N. Alon, B. Awerbuch, Y. Azar, N. Buchbinder, and J. Naor. The online set cover problem. In *the 35th annual ACM Symp. on the Theory of Computation*, 2003.
2. N. Alon, B. Awerbuch, Y. Azar, N. Buchbinder, and J. Naor. A general approach to online network optimization problems. In *SODA*, 2004.
3. B. Awerbuch, Y. Azar, and Y. Bartal. On-line generalized steiner problem. In *Proc. of the 7th Ann. ACM-SIAM Symp. on Discrete Algorithms*, 1996.
4. B. Awerbuch, Y. Azar, and S. Plotkin. Throughput-competitive online routing. In *Proc. of 34th FOCS*, pages 32–40, 1993.
5. P. Berman and C. Coulston. On-line algorithms for steiner tree problems. In *Proc. of the 29th annual ACM Symp. on the Theory of Computation*, 1997.
6. L. Fleischer. A fast approximation scheme for fractional covering problems with variable upper bounds. In *SODA 2004*, pages 1001–1010.
7. L. K. Fleischer. Approximating fractional multicommodity flow independent of the number of commodities. *SIAM Journal on Discrete Mathematics*, 2000.
8. N. Garg and R. Khandekar. Fractional covering with upper bounds on the variables: Solving lps with negative entries. In *ESA 2004*, pages 371–382.
9. N. Garg and J. Konemann. Faster and simpler algorithms for multicommodity flow and other fractional packing problems. In *FOCS 1998*, pages 300–309.
10. M. Imase and B. Waxman. Dynamic steiner tree problem. *SIAM Journal Discrete Math.*, 4:369–384, 1991.

11. S. G. Kolliopoulos and N. E. Young. Tight approximation results for general covering integer programs. In *FOCS 2001*, pages 522–528.
12. P. Raghavan. Probabilistic construction of deterministic algorithms: approximating packing integer programs. *J. Comput. Syst. Sci.*, 37(2):130–143, 1988.
13. P. Raghavan and C. Thompson. Randomized rounding: A technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7:365–374, 1987.
14. E. T. S. Plotkin and D. Shmoys. Fast approximation algorithms for fractional packing and covering problems. *Math. of Operations Research*, 20:257–301, 1995.
15. A. Srinivasan. Improved approximation guarantees for packing and covering integer programs. *SIAM Journal on Computing*, 29(2):648–670, 1999.
16. N. E. Young. Randomized rounding without solving the linear program. In *Proc. of the 6th annual ACM-SIAM symp. on Discrete algorithms*, pages 170–178, 1995.

Efficient Algorithms for Shared Backup Allocation in Networks with Partial Information

Yigal Bejerano¹, Joseph Naor^{2,*}, and Alexander Sprintson³

¹ Bell Labs, Lucent Technologies
bej@research.bell-labs.com

² Department of Computer Science, Technion-Israel Institute of Technology
naor@cs.technion.ac.il

³ Department of Electrical Engineering, California Institute of Technology
spalex@caltech.edu

Abstract. We study efficient algorithms for establishing reliable connections with bandwidth guarantees in communication networks. In the normal mode of operation, each connection uses a *primary* path to deliver packets from the source to the destination. To ensure continuous operation in the event of an edge failure, each connection uses a set of backup *bridges*, each bridge protecting a portion of the primary path. To meet the bandwidth requirement of the connection, a certain amount of bandwidth must be allocated on the edges of primary path, as well as on the backup edges. In this paper, we focus on minimizing the amount of required backup allocation by *sharing* backup bandwidth among different connections. We consider efficient sharing schemes that require only *partial* information about the current state of the network. In particular, the only information available for each edge is the total amount of primary allocation and the cost of allocating backup bandwidth on this edge. We consider the problem of finding a minimum cost backup allocation together with a set of bridges for a given primary path. We prove that this problem is \mathcal{NP} -hard and present an approximation algorithm whose performance is within $\mathcal{O}(\log n)$ of the optimum, where n is the number of edges in the primary path.

1 Introduction

Modern communication networks are expected to provide a certain level of *Quality of Service* (QoS) guarantees and also be resilient to failures. A widely used approach to achieve this goal is to provision for each connection a *primary* path and a set of backup paths. The primary QoS path is used during normal network operation; upon failure of a network element (node or edge) in the primary path, the traffic is immediately switched to a backup path. To provide QoS guarantees, a certain amount of bandwidth must be reserved on each edge of the primary path, as well as on the backup edges.

* This research is supported in part by a foundational and strategical research grant from the Israeli Ministry of Science, and by a US-Israel BSF Grant 2002276.

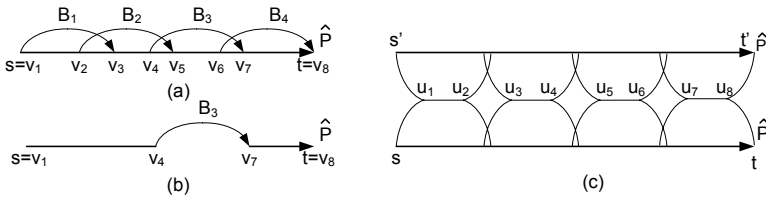


Fig. 1. (a) A primary path and a set of bridges $\{B_1, \dots, B_4\}$; (b) Activation of bridge B_3 upon failure of an edge (v_5, v_6) (c) Sharing of backup edges by two connections

In this paper we employ a *local restoration* method in order to facilitate resilience to edge failures. In this method we provision a set of *bridges*, each bridge protecting a portion of the primary path. A bridge is a path between two nodes of the primary path that shares no common edges with it. Upon failure of an edge in the primary path, the traffic is switched to one of the bridges. Fig. 1(a) depicts an example of a primary path \hat{P} and a set of bridges $\{B_1, \dots, B_4\}$ that protect edges of \hat{P} . Fig. 1(b) depicts the backup path used upon failure of edge (v_5, v_6) of \hat{P} . The backup path is formed by substituting the subpath $\{v_4, \dots, v_7\}$ of \hat{P} by bridge B_3 .

Due to budget constraints, resilience and survivability must be built into a network in an efficient manner, that is, the amount of bandwidth dedicated for this purpose must be minimized. An attractive way to achieve this goal is by *sharing* backup bandwidth among multiple connections. Sharing of backup bandwidth is possible due to low probability of simultaneous failures of multiple edges in the network. Fig. 1(c) shows an example of two connections that share several backup edges. The connections use \hat{P} and \hat{P}' as primary paths and share backup edges $(u_1, u_2), (u_3, u_4), (u_5, u_6),$ and (u_7, u_8) .

We consider a practical network setting, in which the source node of the connection computes both the primary path and a set of bridges. The source node has only *partial* information about the current state of the network. In particular, for each edge e , the following two parameters are given:

1. f_e - The total amount of bandwidth provisioned by the primary paths that use edge e ;
2. $c_e(b)$ - The cost of allocating b units of backup bandwidth on edge e .

We use the first parameter, f_e , to ensure that upon failure of edge e , all primary paths that use e are protected. To this end, we reserve along each edge of the bridge B that protects e at least f_e units of bandwidth. Indeed, in the worst case scenario all traffic that uses a failed edge e may be re-routed via bridge B .

The function $c_e(b)$ specifies the cost of allocating b units of backup bandwidth on edge e . This function may depend, for example, on the amount of backup bandwidth provisioned on e by prior connections. Indeed, in the case when edge e has a large amount of bandwidth provisioned by other connections, the cost of

allocating b units of backup bandwidth for the current connection may be small or even zero.

The partial information model used in this paper is inspired by the paper of Kodialam and Lakshman [1]. This paper provides an empirical evidence that partial information model facilitates efficient sharing of backup bandwidth, resulting in a reduction in the total amount of backup bandwidth reserved throughout the network. At the same time, the model requires only a small amount of routing information that needs to be disseminated in the network (only a few parameters per each edge in the network).

Our Results. In this study we consider the problem of finding a minimum cost backup allocation together with a set of bridges for a given primary path. We prove that this problem is \mathcal{NP} -hard and present an approximation algorithm whose performance is within $\mathcal{O}(\log n)$ of the optimum, where n is the number of edges in the primary path.

We prove that this problem is \mathcal{NP} -hard. However, we show that, by exploiting a certain combinatorial structure of the problem, we can devise an efficient approximation algorithm that achieves an approximation ratio of $8 \log n$. The computational complexity of our algorithm is $\mathcal{O}(n^3(|E| + |V| \log |V|))$, where n is the maximum number of edges in the primary path. Our algorithm can be easily extended for the problem of finding both a primary path and backup allocation. While the ideas that led to the development of our algorithm as well as the performance proofs are rather involved, the algorithm *per se* is relatively simple and easy to implement.

Proof Techniques. We analyze important properties of the problems related to bandwidth allocation with backup sharing. Specifically, we identify *internal edge sharing* as a major obstacle to finding an efficient solution. Internal edge sharing refers to the situation in which bridges that protect the same primary path share edges (see e.g. Fig. 2). We prove by employing involved combinatorial techniques that for any given primary path there is an optimal set of bridges such that each edge is included in at most $8 \log n$ bridges, where n is the number of edges in the primary path.

This property allows us to construct an approximation algorithm based on local optimization. Specifically, we define the *local cost* of a bridge to be the sum of the costs incurred at its edges. Our goal is then to find a set of bridges that protect all edges in the primary path such that the total local cost of all bridges is minimal. We show that such a solution has a certain hierarchical property and present an algorithm based on dynamic programming that identifies an optimal set of bridges with respect to local costs. We note that this approach effectively ignores internal sharing. Indeed, since the cost of each bridge is computed independently of other bridges, an edge can contribute to the cost of several bridges. However, since each edge appears in at most $8 \log n$ bridges, the cost of our solution is at most $8 \log n$ times higher than the optimum.

Related Work. Sharing of backup resources as a means for improving network performance was proposed by [1,2]. The network design problems in the context

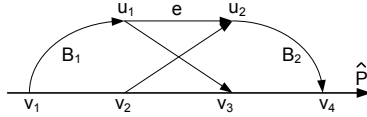


Fig. 2. Internal sharing. Bridges $B_1 = \{v_1, u_1, u_2, v_2\}$ and $B_2 = \{v_3, u_1, u_2, v_4\}$ share an edge (u_1, u_2) .

of backup sharing were investigated by [3,4]. In particular, [3] studied a simple model in which edges that belong to primary and backup paths are assigned different costs while [4] focused on the restoration problems in the *Hose* model. Papers [5–7] investigated practical aspects of path restoration. Several heuristic methods, based on linear programming for partial information model, were proposed in [1]. The current study is the first one to provide efficient approximation algorithms for computing shared backup allocation in the partial information model.

Due to space constraints, some proofs and technical details are omitted and can be found in the full version of this paper.

2 Model and Definitions

We represent the network by an undirected graph $G(V, E)$, where V is the set of nodes and E is the set of edges. An (s, t) -path is a sequence of distinct nodes $P = \{s = v_0, v_1, \dots, t = v_n\}$, such that, for $1 \leq i \leq n$, $(v_{i-1}, v_i) \in E$. Here, $n = |P|$ is the *hop count* of P . We denote by $E(P)$ the set of edges of P . The subpath of P that extends from v_i to v_j is denoted by $P_{(v_i, v_j)}$.

A unicast connection links a source node s with a destination node t . In a normal mode of operation, the packets are sent over the *primary* path $\hat{P} = \{s = v_0, v_1, \dots, t = v_n\}$. To ensure continuous operation in the event of an edge failure, we provision a set of *bridges* $\mathbb{B} = \{B_1, B_2, \dots, B_k\}$ that protect edges in the primary path. A bridge $B_i = \{s_i, \dots, t_i\}$ is a path between $s_i \in \hat{P}$ and $t_i \in \hat{P}$ that has no common edges with the subpath $\hat{P}_{(s_i, t_i)}$ of \hat{P} . We say that a bridge $B_i = \{s_i, \dots, t_i\}$ *protects* an edge $e \in \hat{P}_{(s_i, t_i)}$ if upon failure of edge e the traffic is switched from $\hat{P}_{(s_i, t_i)}$ to B_i . We denote by $\varphi(B_i)$ the set of edges in the primary path protected by bridge B_i . The set of edges that belong to bridges in \mathbb{B} is denoted by E^r , i.e., $E^r = \{e \in B_i \mid B_i \in \mathbb{B}\}$. For each edge $(v_{i-1}, v_i) \in \hat{P}$ we denote by $f_{(v_{i-1}, v_i)}$ the total amount of primary bandwidth reserved on (v_{i-1}, v_i) .

To ensure continuous operation in the event of an edge failure, we reserve a certain amount of bandwidth on each edge in E^r . In the partial information model, the backup reservation $\omega(e)$ on each edge $e \in B_j$ must satisfy:

$$\omega(e) \geq \max_{(v_{i-1}, v_i) \in \varphi(B_j)} f_{(v_{i-1}, v_i)}. \tag{1}$$

The reason for such a conservative approach is to guarantee that all primary paths that use edge (v_{i-1}, v_i) can be restored in the event of a failure of (v_{i-1}, v_i) .

Indeed, since we have no information about the backup paths used by other connections, we make a worst-case assumption that all backup traffic triggered by a failure of (v_{i-1}, v_i) is routed via edges of bridge B_j .

Since an edge $e \in E^r$ can belong to multiple bridges, the backup $\omega(e)$ reservation on edge e must satisfy:

$$\omega(e) \geq \max_{B_j, e \in B_j} \max_{(v_{i-1}, v_i) \in \varphi(B_j)} f_{(v_{i-1}, v_i)} \tag{2}$$

For each edge $e \in E$ we are given a function $c_e(b)$ that specifies, for any $b \geq 0$, the cost of reserving b units of backup bandwidth on edge e . We assume that functions $c_e(b)$ are monotonically increasing and can be computed in constant time.

Backup Allocation Problem

In the backup allocation problem, our goal is to find a *restoration topology* $\hat{\mathcal{R}}$ for a given primary path $\hat{P} = \{s = v_0, v_1, \dots, t = v_n\}$. A restoration topology $\hat{\mathcal{R}}$ is specified by a 4-tuple $\{\mathbb{B}, E^r, \varphi, \omega\}$, where \mathbb{B} is a set of bridges $\{B_1, B_2, \dots, B_k\}$; $E^r = \{e \in B_i \mid B_i \in \mathbb{B}\}$ is the set of edges that belong to bridges in \mathbb{B} ; $\varphi : \mathbb{B} \rightarrow 2^{E(\hat{P})}$ is a function that specifies, for each bridge $B_j \in \mathbb{B}$, the set of edges in the primary path protected by B_j ; and $\omega : E^r \rightarrow Z$ is a function that specifies the amount of backup bandwidth we need to allocate on each edge of E^r .

A *feasible restoration topology* must satisfy the following conditions:

1. Each edge in the primary path \hat{P} must be protected by a bridge in \mathbb{B} , i.e., for each $(v_{i-1}, v_i) \in \hat{P}$ there is a bridge $B_j \in \mathbb{B}$, such that $(v_{i-1}, v_i) \in \varphi(B_j)$.
2. The backup allocation $\omega(e)$ on each edge $e \in E^r$ must satisfy Equation (2).

The cost $C(\hat{\mathcal{R}})$ of the restoration topology is defined to be the cost of its edges, i.e., $C(\hat{\mathcal{R}}) = \sum_{e \in E^r} c_e(\omega(e))$, where c_e is the cost function associated with edge e . Our goal is to find a restoration topology of minimal cost. We refer to this problem as Problem BA (Backup Allocation) and denote the minimal cost of a solution to Problem BA by OPT. In the full version of this paper we show that Problem BA is \mathcal{NP} -hard.

3 Properties of the Optimal Backup Allocation

The main obstacle in finding an optimal solution to Problem BA is the fact that different bridges in $\hat{\mathcal{R}}$ can share edges. Such *internal sharing* is one of the reasons of the \mathcal{NP} -hardness of the problem. In this section we prove that there exists a restoration topology $\hat{\mathcal{R}} = \{\mathbb{B}, E^r, \varphi, \omega\}$ such that $C(\hat{\mathcal{R}}) = \text{OPT}$ and each edge $e \in E^r$ belongs to at most $8 \log n$ bridges of \mathbb{B} , where n is the number of edges in the primary path.

Theorem 1. *Given a primary path \hat{P} , there exists a restoration topology $\hat{\mathcal{R}} = \{\mathbb{B}, E^r, \varphi, \omega\}$ for \hat{P} such that $C(\hat{\mathcal{R}}) = \text{OPT}$ and each edge $e \in E^r$ belongs to at most $8 \log n$ bridges of $\hat{\mathbb{B}}$, where $n = |\hat{P}|$.*

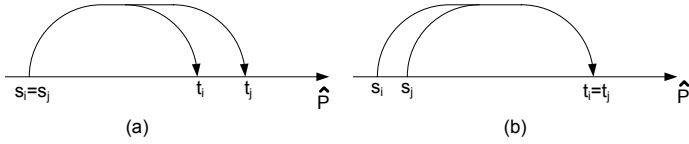


Fig. 3. Bridges with common prefix (a) and suffix (b)

In Section 4, we use Theorem 1 in order to devise an approximation algorithm for Problem BA. The algorithm finds a restoration topology $\hat{\mathcal{R}}$ whose cost is at most $8 \log n$ times more than OPT. The rest of this section is devoted to the proof of Theorem 1.

We begin by introducing the following notation. Let $B_i = \{s_i, \dots, t_i\}$ be a bridge in \mathbb{B} . A subpath $\{s_i, \dots, v\}$ of B_i is referred to as a *prefix* of B_i . Similarly, a subpath $\{v, \dots, t_i\}$ is referred to as a *suffix* of B_i . We say that two bridges $B_i = \{s_i, \dots, t_i\}$ and $B_j = \{s_j, \dots, t_j\}$ *share a prefix* if there exists a node $v \in B_i$ such that the prefix $\{s_i, \dots, v\}$ of B_i is identical to the prefix $\{s_j, \dots, v\}$ of B_j and the suffixes $\{v, \dots, t_i\}$ and $\{v, \dots, t_j\}$ of B_i and B_j are mutually edge-disjoint. Similarly, we say that two bridges $B_i = \{s_i, \dots, t_i\}$ and $B_j = \{s_j, \dots, t_j\}$ *share a suffix* if there exists a node $v \in B_i$ such that the suffix $\{v, \dots, t_i\}$ of B_i is identical to the suffix $\{v, \dots, t_j\}$ of B_j and the prefixes $\{s_i, \dots, v\}$ and $\{s_j, \dots, v\}$ of B_i and B_j are mutually edge-disjoint. Fig. 3 depicts examples of bridges that share a prefix and a suffix.

3.1 Outline of the Proof

Let $\hat{P} = \{s = v_0, v_1, \dots, t = v_n\}$ be a primary path and let $\mathcal{R} = \{\mathbb{B}, E^r, \varphi, \omega\}$ be a restoration topology for \hat{P} such that $C(\mathcal{R}) = \text{OPT}$ and the number of edges in E^r is minimal. We construct a restoration topology $\hat{\mathcal{R}} = \{\hat{\mathbb{B}}, E^r, \hat{\varphi}, \omega\}$ that satisfies condition of the theorem. The restoration topology $\hat{\mathcal{R}}$ is formed from \mathcal{R} by modifying the set of bridges \mathbb{B} and the bridge assignment function φ . The set of backup edges E^r and backup allocation ω of $\hat{\mathcal{R}}$ is identical to that of \mathcal{R} .

Our proof includes the following steps:

1. First, we construct a restoration topology $\bar{\mathcal{R}} = \{\bar{\mathbb{B}}, E^r, \bar{\varphi}, \omega\}$ that satisfies the following property. Let $\bar{B}_i = \{s_i, \dots, t_i\}$ and $\bar{B}_j = \{s_j, \dots, t_j\}$ be two bridges of $\bar{\mathbb{B}}$ such that there exists an edge (x, y) that appears in both bridges (in the same direction). Then, bridges \bar{B}_i and \bar{B}_j either share a suffix or share a prefix (see Fig. 4).
2. Then, we show that there exist a partition $\pi = \{S_i\}$ of $\bar{\mathbb{B}} = \{\bar{B}_1, \bar{B}_2, \dots, \bar{B}_k\}$ such that: (i) For each subset S_i of π it holds that either all bridges in S_i share a prefix or all bridges in S_i share a suffix. (ii) Each edge $e \in E^r$ belongs to bridges of at most four different subsets of π .
3. Finally, we prove the following assertion. Let S_i be a subset of bridges in $\bar{\mathbb{B}}$ such that either all bridges in S_i share a prefix or all bridges share a suffix. Also, let $E(S_i) \subseteq E^r$ be the set of edges that belong to bridges in S_i , i.e., $E(S_i) = \{e \in E^r \mid B \in S_i\}$. Then, there exists a set of bridges \hat{S}_i such that

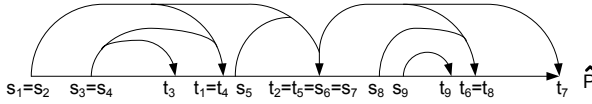


Fig. 4. An example of a restoration topology that satisfies the condition of Step 1

- (i) $E(\hat{S}_i) \subseteq E(S_i)$, where $E(\hat{S}_i) = \{e \in \hat{P} \mid \hat{B} \in \hat{S}_i\}$; (ii) Set \hat{S}_i protects the same set of edges as S_i , i.e., each edge $(v_{i-1}, v_i) \in \hat{P}$ protected by a bridge in S_i is protected by a bridge in \hat{S}_i ; (iii) Each edge $e \in E(\hat{S}_i)$ belongs to at most $2 \log n$ bridges of \hat{S}_i .

It is easy to verify that the union of all sets \hat{S}_i that correspond to sets in π satisfies the requirement of the theorem.

Step 1. Let $\hat{P} = \{s = v_0, v_1, \dots, t = v_n\}$ be a primary path and let $\mathcal{R} = \{\mathbb{B}, E^r, \varphi, \omega\}$ be an optimal restoration topology for \hat{P} . We show how to construct a set of bridges \mathbb{B} and the corresponding bridge assignment function φ such that any two bridges in \mathbb{B} that have an edge in common either share a prefix or share a suffix.

Let G^r be a subgraph of G induced by edges in E^r . Each edge $e \in G^r$ has a backup reservation $\omega(e)$ in \mathcal{R} . We introduce a new bottleneck metric for paths in G^r with respect $\omega(e)$. Specifically, given a path P in G^r we define the weight of the path $B(P)$ to be the smallest amount of backup reservation of an edge in P , i.e., $B(P) = \min_{e \in P} \omega(e)$.

Next, we define two functions, $\tau(v)$ and $\gamma(v)$, for each node $v \in G^r$. Function $\tau(v)$ maps a node $v \in G^r$ to a node $v_i \in \hat{P}$, while function $\gamma(v)$ maps a node $v \in G^r$ to a path between $\tau(v)$ and v in G^r . In order to define $\tau(v)$ and $\gamma(v)$, we identify a tree \mathcal{T}_{v_i} that connects $v_i \in \hat{P}$ with the rest of the nodes in G^r such that each path $P = \{v_i, \dots, u\} \in \mathcal{T}_{v_i}$ has the maximum bottleneck weight among all paths that connect v_i and u in G^r .

The function $\tau(v)$ is defined as follows. If v is a node in the primary path \hat{P} , then $\tau(v) = v$. Otherwise, $\tau(v)$ is equal to the node $v_i \in \hat{P}$ that satisfies the following conditions (i) The path between v_i and v in \mathcal{T}_{v_i} has more bandwidth than any other path between $v_j \in \hat{P}$ and v . (ii) The distance (in hops) between s and v_i in \hat{P} is smaller than that of any other node $v_j \in \hat{P}$ that satisfy condition (i).

The second function, $\gamma(v)$, is defined as follows. If v belongs to the primary path \hat{P} , then $\gamma(v)$ is an empty path. Otherwise, $\gamma(v)$ is a path between $v_i = \tau(v)$ and v in \mathcal{T}_{v_i} .

We are ready to describe the construction of the set of bridges \mathbb{B} . For each edge $(v_{i-1}, v_i) \in \hat{P}$, we identify a bridge \bar{B}_i as follows. Let $B_j = \{s_j, \dots, t_j\} \in \mathbb{B}$ be a bridge in \mathcal{R} that protects edge (v_{i-1}, v_i) . Let (x, y) be an edge of B_j for which it holds that $\tau(x)$ is a predecessor of v_{i-1} in \hat{P} and $\tau(y)$ is a successor of v_i in \hat{P} . Note that such an edge must exist because $\tau(s_j)$ is a predecessor of v_{i-1} and $\tau(t_j)$ is a successor of v_i . Then, we set \bar{B}_i to be a concatenation of path $\gamma(x)$, edge (x, y) , and a path $\gamma(y)$ (in a reverse direction). Note that for any

node $v \in \gamma(x)$ it holds that $\tau(v) = \tau(x)$ and for each node $v \in \gamma(y)$ it holds that $\tau(v) = \tau(y)$. This implies that paths $\gamma(x)$ and $\gamma(y)$ are mutually node-disjoint. This fact, in turn, implies that \bar{B}_i is a simple path (i.e., does not include a cycle).

In the following lemma we prove that bridges in $\bar{\mathbb{B}}$ satisfy the required property.

Lemma 2. *Let \bar{B}_i and \bar{B}_j be two bridges in $\bar{\mathbb{B}}$ that share an edge (x, y) , i.e., $(x, y) \in \bar{B}_i$ and $(x, y) \in \bar{B}_j$. Then, \bar{B}_i and \bar{B}_j share either a prefix or a suffix.*

Proof. We consider three cases. In the first case it holds that $\tau(x) = \tau(y)$ and x is an ancestor of y in \mathcal{T}_{v_i} . In this case $\gamma(y)$ is a prefix of both bridges \bar{B}_i and \bar{B}_j . Indeed, if $\gamma(y)$ is a suffix of \bar{B}_i then \bar{B}_i contains a loop $\{x, y, x\}$, resulting in a contradiction. In the second case $\tau(x) = \tau(y)$ and y is an ancestor of x in \mathcal{T}_{v_i} . In this case $\gamma(x)$ is a suffix of both bridges \bar{B}_i and \bar{B}_j . Finally, in the third case $\tau(x) \neq \tau(y)$. In this bridge \bar{B}_i is identical to \bar{B}_j . Indeed, in this case $\gamma(x)$ is a prefix of both \bar{B}_i and \bar{B}_j , while $\gamma(y)$ is a suffix of both \bar{B}_i and \bar{B}_j .

Step 2. We show that $\bar{\mathbb{B}} = \{\bar{B}_1, \bar{B}_2, \dots, \bar{B}_k\}$ can be partitioned into subsets $\pi = \{S_i\}$ such that: (i) For each subset S_i it either holds that any two bridges in S_i share a prefix or any two bridges in S_i share a suffix. Further, any edge $e \in E^r$ belongs to bridges of at most four different subsets of π .

We construct the $\pi = \{S_i\}$ through the following *partitioning* procedure. The procedure uses an undirected auxiliary graph $G'(V', E')$, described below. The vertices V' of G' correspond to the nodes of the primary path \hat{P} and the edges E' of G' correspond to the bridges in $\bar{\mathbb{B}}$. Specifically, for each bridge $\bar{B}_i = \{s_i, \dots, t_i\} \in \bar{\mathbb{B}}$ we add an edge between s_i and t_i in G' . In the full version of this paper we prove that the subgraph G^r induced by edges in E^r , and, in turn, the auxiliary graph G' do not contain cycles.

The procedure includes the following steps for each connected component G'_i of G' :

1. Select a node $v \in G'_i$ and find an orientation of edges in G'_i such that the resulting graph \vec{G}'_i is a directed tree rooted at v ;
2. For each node $u \in \vec{G}'_i$ create two subsets S^1_u and S^2_u in the partition π . Both subsets include bridges of $\bar{\mathbb{B}}$ that correspond to edges incident to u . The first subset includes bridges for which u is the starting node, while the second set includes bridges for which u is the end node.

Lemma 3. *Let e be an edge in E^r and let S be a set of bridges in $\bar{\mathbb{B}}$ that include e . Then, the bridges of S belong to at most four different subsets of π .*

Step 3. Let $\pi = \{S_i\}$ be a partition of $\bar{\mathbb{B}}$ that satisfies the conditions of Step 2. Also, let $S_i \in \pi$ be a set of bridges such that all bridges in S_i share a prefix. We begin by removing from S_i all *redundant* bridges. A bridge $\bar{B} \in S_i$ is said to be redundant if all edges in $\varphi(\bar{B})$ can be protected by other bridges in S_i .

We denote by $E(S_i) = \{e \in \hat{B} \mid \hat{B} \in S_i\}$ the set of edges that belong to bridges in S_i and show that there exists a set of bridges \hat{S}_i that satisfies the

following conditions: (i) $E(\hat{S}_i) \subseteq E(S_i)$, where $E(\hat{S}_i) = \{e \in \hat{B} \mid \hat{B} \in \hat{S}_i\}$; (ii) Each edge $(v_{i-1}, v_i) \in \hat{P}$ protected by a bridge in S_i is protected by a bridge in \hat{S}_i ; (iii) Each edge $e \in E(\hat{S}_i)$ belongs to at most $2 \log n$ bridges of \hat{S}_i .

We denote by s' the starting node of all bridges in S_i and by $T = \{t^j\}$ the set of end nodes of bridges in S_i , such that t^{j-1} is a predecessor of node t^j in \hat{P} . For each $j, 1 \leq j \leq T$, we denote by B^j the bridge in S_i with end node t^j . We also denote by G' the subgraph of G induced by $E(S_i)$. Note that G' is a subgraph of G^r (recall that G^r is a subgraph of G induced by edges in E^r). Since G^r does not contain a cycle, G' is a tree. In addition, we denote by Γ the set of edges in \hat{P} protected by bridges in S_i , i.e., $\Gamma = \cup_{\hat{B} \in S_i} \varphi(\hat{B})$. We partition Γ to $|T|$ subsets $\Gamma_1, \dots, \Gamma_{|T|}$ such that subset Γ_1 include edges in Γ located between nodes s' and t^1 and a subset Γ_j includes edges of Γ located between nodes t^{j-1} and t^j in T .

Lemma 4.

1. For each $j, 1 \leq j \leq T$, it holds that bridge B^j can protect all edges in Γ_j , i.e., $\min_{e \in B^j} \omega(e) \geq \max_{(v_{x-1}, v_x) \in \Gamma_j} f_{(v_{x-1}, v_x)}$
2. Let P be a path in G' between t^k and t^j , where t^k is a predecessor of t^j in \hat{P} . Then, P is a bridge that can protect all edges in Γ_j , i.e., $\min_{e \in P} \omega(e) \geq \max_{(v_{x-1}, v_x) \in \Gamma_j} f_{(v_{x-1}, v_x)}$.

We are ready to describe a procedure that constructs the set of bridges \hat{S}_i . The procedure includes the following steps.

1. For each $B_j \in S_i$ set $\varphi(B_j) = \Gamma_j$.
2. While S_i is not empty, perform the following operations:
 - (a) Denote by $E(S_i)$ the set of edges that belong to bridges in S_i , i.e., $E(S_i) = \{e \in \hat{B} \mid \hat{B} \in S_i\}$. Denote by G' the subgraph of G^r induced by edges in $E(S_i)$.
 - (b) Identify an Euler tour $W = \{s', \dots, s'\}$ in G' .
 - (c) Break W into paths $\{P_k\}$ by cutting W at nodes s and $t_k \in T$.
 - (d) Denote by s_k the starting node of P_k and by t_k the end node of P_k , such that s_k is a predecessor of t_k in \hat{P} .
 - (e) For each path $P_k = \{s_k, \dots, t_k\}$ for which there is a bridge $B^j = \{s', \dots, t^j\} \in S_i$ with the same end node (i.e., $t_k = t^j$), perform the following operations:
 - i. Add P_k into \hat{S}_i and set $\varphi(P^k) = \varphi(B^j)$.
 - ii. Remove B^j from S_i .

Fig. 5 demonstrates a single iteration of the procedure. A set of bridges that share a common prefix is depicted in Fig. 5(a). An Euler tour W on G' is depicted in Fig. 5(b). We break W into four paths, two of which are added to \hat{S}_i , as depicted in Fig. 5(c).

Theorem 5. *The set of bridges \hat{S}_i satisfies the following conditions: (i) $E(\hat{S}_i) \subseteq E(S_i)$, where $E(\hat{S}_i) = \{e \in \hat{B} \mid \hat{B} \in \hat{S}_i\}$; (ii) Each edge $(v_{i-1}, v_i) \in \hat{P}$ protected by a bridge in S_i is protected by a bridge in \hat{S}_i ; (iii) Each edge $e \in E(\hat{S}_i)$ belongs to at most $2 \log n$ bridges of \hat{S}_i .*

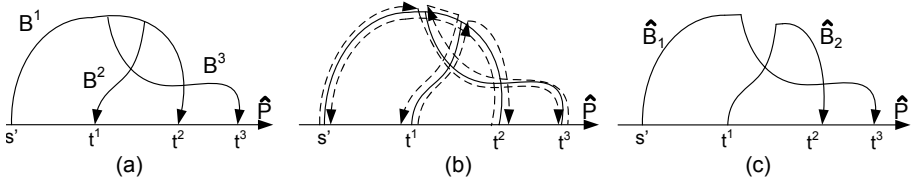


Fig. 5. (a) A set S_i of bridges that share a common prefix (b) An Euler tour on G' (c) Two new bridges are added to S_i

The case in which all bridges in S_i have a common suffix can be proven by using similar arguments.

4 Approximation Algorithm for Problem BA

4.1 Locally Optimal Restoration Topologies

In this section we present an approximation algorithm for Problem BA. We begin by defining the notion of the *local cost* of a restoration topology.

Definition 1 (Local Cost). Let $\mathcal{R} = \{\mathbb{B}, E^r, \varphi, \omega\}$ be a restoration topology for \hat{P} . The local cost $C^*(B)$ of a bridge $B \in \mathbb{B}$ is defined to be the sum of local costs incurred at its edges, i.e., $C^*(B) = \sum_{e \in B} c_e(\omega(e))$. The local cost of a restoration topology is defined to be the sum of the local costs of its bridges:

$$C^*(\mathcal{R}) = \sum_{B \in \mathbb{B}} C^*(B) = \sum_{B \in \mathbb{B}} \sum_{e \in B} c_e(\omega(e)). \tag{3}$$

Definition 2 (Partial Restoration Topology). Let P be a subpath of the primary path \hat{P} . A restoration topology $\mathcal{R} = \{\mathbb{B}, E^r, \varphi, \omega\}$ is referred to as a partial restoration topology for P if each edge in P is protected by a bridge $B \in \mathcal{R}$. The backup allocations $\omega(e)$ on each edge $e \in E^r$ must satisfy the conditions of (2).

We say that a partial restoration topology \mathcal{R} for P is *locally optimal* if its local cost is less than or equal to the local cost of any other partial restoration topology \mathcal{R} for P . The minimum cost of a locally optimal restoration topology for P is denoted by $\text{OPT}^*(P)$. From Theorem 1 it follows that $\text{OPT}^*(\hat{P}) \leq 8 \cdot \text{OPT} \log n$, where $n = |\hat{P}|$.

The next lemma establishes a hierarchical property of restoration topologies with minimal local cost. This lemma allows us to use the methods of dynamic programming in order to find restoration topologies of minimal local cost.

Lemma 6. Let $P' = \{s', \dots, t'\}$ be a subpath of the primary path \hat{P} . Then one of the following conditions hold:

1. Path P can be partitioned into edge-disjoint subpaths P^1, \dots, P^k such that $\text{OPT}^*(P') = \sum_{i=1}^k \text{OPT}^*(P^i)$;

2. *The are exist edge-disjoint subpaths P^1, \dots, P^k of P' , a bridge $B = \{s', \dots, t'\} \in G \setminus P'$, and a value ζ , such that:*
 - (a) ζ is the maximum value $f_{(v_{i-1}, v_i)}$ of an edge $(v_{i-1}, v_i) \in P'$ that does not belong to paths P^1, \dots, P^k ;
 - (b) $B = \{s', \dots, t'\}$ is a minimum cost path between s' and t' with respect to the cost $\sum_{e \in B} c_e(\zeta)$;
 - (c) $\text{OPT}^*(P') = \sum_{i=1}^k \text{OPT}^*(P^i) + \sum_{e \in B} c_e(\zeta)$.

The hierarchical property allows us to efficiently find an optimal restoration topology $\hat{\mathcal{R}}$ by using the methods and tools of dynamic programming.

4.2 Dynamic Programming Algorithm

The algorithm exploits the hierarchical property of locally optimal restoration topologies, established by Lemma 6. The algorithm begins with the subpaths of \hat{P} that include a single edge and computes for each subpath an optimal restoration topology that protects it. Then, it computes locally optimal restoration topologies for all subpaths of length 2, 3 and so on, until the locally optimal restoration topology for the entire primary path is found. The order of processing the subpaths of P by the algorithm ensures that when the algorithm computes an optimal restoration topology for a subpath of length i , it already has available the optimal restoration topologies for all subpaths of length smaller than i .

We observe that optimal restoration topology for a subpath of length 1 includes a single bridge, which is easy to identify. Indeed, let P be a subpath that includes a single edge (v_{i-1}, v_i) . Note that each edge e of bridge B must be allocated $f_{(v_{i-1}, v_i)}$ units of backup bandwidth. Thus, to find a locally optimal bridge that protects (v_{i-1}, v_i) we compute a shortest path in $G \setminus \hat{P}$ with respect to edge costs $c_e(f_{(v_{i-1}, v_i)})$ between s_i and t_i , where s_i is a predecessor of v_{i-1} in \hat{P} and t_i is a successor of node v_i in \hat{P} . Such a path can be computed by a single invocation of a shortest path algorithm such as Dijkstra’s algorithm.

For a subpath P of \hat{P} of length more than 1, we need to consider two possible cases, described in Lemma 6. For the first case, we need to determine a partition P^1, \dots, P^k of P such that total cost of restoration topologies $\mathcal{R}^1, \dots, \mathcal{R}^k$ is minimal, where \mathcal{R}^i is a locally optimal restoration topology that protect P^i . Note that the optimal restoration topologies $\mathcal{R}^1, \dots, \mathcal{R}^k$ are already computed by the algorithm. Such an optimal partition can be determined by using the auxiliary graph \hat{G} , that includes, for each subpath $P_{(v_i, v_j)}$ of P , an edge (v_i, v_j) , whose cost is equal to the cost of the locally optimal restoration topology that protects $P_{(v_i, v_j)}$. Then, we compute a shortest path between the source node and the destination node of P and identify an optimal restoration topology for P by taking the union of all restoration topologies that correspond to the edges of the shortest path.

The second case is more complicated, as we need to determine a bridge \hat{B} and several restoration topologies $\mathcal{R}_1, \dots, \mathcal{R}_k$ that protect the edges of P . The key step is to determine the amount of backup bandwidth ζ that must be reserved

on edges of \hat{B} . Indeed, if ζ is known, then we can easily determine \hat{B} (in a manner similar to the case in which the subpath P includes a single edge) and the edges of P that are protected by \hat{B} (edges (v_{i-1}, v_i) for which it holds that $f_{(v_{i-1}, v_i)} \leq \zeta$). Then, we can identify restoration topologies that protect all other edges of P (not protected by \hat{B}) by using the same technique as in the first case.

In order to determine ζ we use the following observation. We observe that ζ must be equal to $f_{(v_{i-1}, v_i)}$ for some of edges in the P . Thus, the optimal value of ζ can be found by performing the following exhaustive search: for each value of ζ from the set $\{f_{(v_{i-1}, v_i)} \mid (v_{i-1}, v_i) \in P\}$ we compute the optimal restoration topology and choose ζ for which the local cost of a restoration topology is minimal.

Theorem 7. *Given a primary path \hat{P} , the algorithm above identifies, in $\mathcal{O}(n^3(|E| + |V| \log |V|))$ time, a solution to Problem BA whose cost is at most $8 \log n$ times more than the optimum.*

References

1. Kodialam, M.S., Lakshman, T.V.: Dynamic routing of bandwidth guaranteed tunnels with restoration. In: Proceedings of IEEE INFOCOM'2000, Tel-Aviv, Israel (2000)
2. Hwang, H., Ahn, S., Choi, Y., Kim, C.: Backup path sharing for survivable ATM networks. In: Proceedings of ICOIN-12. (1998)
3. Chekuri, C., Gupta, A., Kumar, A., Naor, J., Raz, D.: Building edge-failure resilient networks. In: Proceedings of IPCO 2002. (2002)
4. Italiano, G., Rastogi, R., Yener, B.: Restoration algorithms for virtual private networks in the hose model. In: Proceedings of IEEE INFOCOM'02, New York, NY (2002)
5. Su, X., Su, C.F.: An online distributed protection algorithm in WDM networks. In: Proceedings of IEEE ICC'01. (2001)
6. Sengupta, S., Ramamurthy, R.: Capacity efficient distributed routing of mesh-restored lightpaths in optical networks. In: Proceedings of IEEE GLOBECOM '01. (2001)
7. Li, G., Wang, D., Kalmanek, C., Doverspike, R.: Efficient distributed path selection for shared restoration connections. In: Proceedings of IEEE INFOCOM'02, New York, NY (2002)

Using Fractional Primal-Dual to Schedule Split Intervals with Demands

Reuven Bar-Yehuda¹ and Dror Rawitz²

¹ Department of Computer Science, Technion, Haifa 32000, Israel
reuven@cs.technion.ac.il

² Caesarea Rothschild Institute, University of Haifa, Haifa 31905, Israel
rawitz@cri.haifa.ac.il

Abstract. We consider the problem of scheduling jobs that are given as groups of non-intersecting intervals on the real line. Each job j is associated with a t -interval, which consists of up to t segments, for some $t \geq 1$, a demand, $d_j \in [0, 1]$, and a weight, $w(j)$. A schedule is a collection of jobs, such that, for every $s \in \mathbb{R}$, the total demand of the jobs in the schedule whose t -interval contains s does not exceed 1. Our goal is to find a schedule that maximizes the total weight of scheduled jobs.

We present a $6t$ -approximation algorithm that uses a novel extension of the *primal-dual schema* called *fractional primal-dual*. The first step in a fractional primal-dual r -approximation algorithm is to compute an optimal solution, x^* , of an LP relaxation of the problem. Next, the algorithm produces an integral primal solution x , and a new LP, denoted by P' , that has the same objective function as the original problem, but contains inequalities that may not be valid with respect to the original problem. Moreover, x^* is a feasible solution of P' . The algorithm also computes a solution y to the dual of P' . x is r -approximate, since its weight is bounded by the value of y divided by r .

We present a *fractional local ratio* interpretation of our $6t$ -approximation algorithm. We also discuss the connection between fractional primal-dual and the fractional local ratio technique. Specifically, we show that the former is the primal-dual manifestation of the latter.

1 Introduction

The Problem. We consider the problem of scheduling jobs that are given as groups of non-intersecting intervals on the real line. Each job j is associated with a t -interval, which consists of up to t non-intersecting intervals, or *segments*, for some $t \geq 1$, and a positive weight, $w(j)$. Each job requires the utilization of a given limited *resource*. The amount of resource available is fixed; we normalize it to unit size. The amount of resource required by job j , or the *demand* of j , is denoted by d_j . A *schedule* is a collection of jobs such that, for every $s \in \mathbb{R}$, the total demand of the jobs in the schedule whose t -interval contains s does not exceed 1. Our goal is to find a schedule that maximizes the total weight of scheduled jobs.

The problem of scheduling t -intervals is NP-hard even when $t = 1$, since it contains *knapsack* as a special case in which all t -intervals intersect. The problem of scheduling t -intervals where all demands are equal to 1 was studied by Bar-Yehuda et al. [1]. Two jobs are said to be in *conflict* if any of their segments intersect. The objective in this special case is to schedule a subset of non-conflicting jobs whose total weight is maximum. The unit demand problem is formulated in [1] as the problem of finding a *maximum weight independent set* (MWIS) in a t -interval graph. When $t = 1$ we get MWIS in interval graphs which is solvable in polynomial time (see, e.g., [2]).

An interesting special case that was discussed in [1] is the family of t -union graphs, in which the segments associated with each job (or vertex) can be labeled in such a way that for any two jobs j_1 and j_2 the segment i_1 of j_1 and segment i_2 of j_2 do not intersect for every $1 \leq i_1, i_2 \leq t$ and $i_1 \neq i_2$. In this case the t segments can be viewed as intervals on orthogonal axes, corresponding to a t -dimensional box. Two boxes are in conflict if their projections on any of the t axes intersect. A two dimensional example is given in Fig. 1. Note that the height of a box corresponds to its demand. In Fig. 1, Jobs 1 and 2 are in conflict, while jobs 1 and 3 are not. Jobs 2 and 3 are also in conflict, but in the more general case, in which demands are allowed, they can be scheduled together, since the sum of their demands is not more than one.

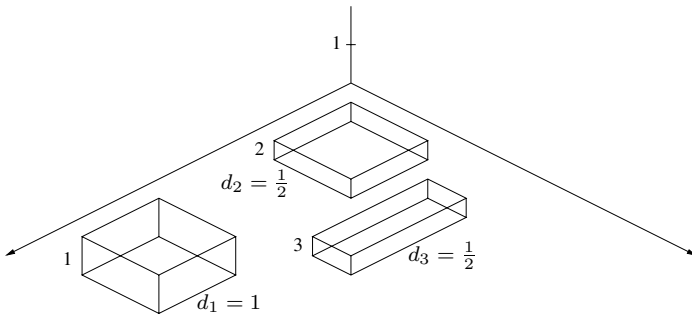


Fig. 1. A two dimensional interpretation of a 2-interval scheduling instance

We describe several applications in which the problem of scheduling t -intervals with demands arises. First, consider a multimedia-on-demand system with a limited bandwidth through which movies are broadcasted to clients (e.g., through a cable TV network). Each movie has a different bandwidth demand, which may depend on its quality. Each client requests a particular movie, and specifies the time at which she would like to start watching it. Moreover, each client specifies at which times she plans to take breaks. In the (offline) throughput maximization version of this problem, we aim to maximize the revenue by deciding which movies to broadcast, subject to the constraint that the bandwidth demands at any given moment can be supplied by the system. (See [1] for more details.) Another application is allocation of *linear resources* [3]. Requests

for a linear resource can be modeled as intervals on the line (e.g., a disk drive is a linear resource when requests are for contiguous blocks [4]). Consider a scenario in which the jobs are requests from several linear resources, and two jobs are in conflict if their requests on one of the resources overlap. (This example corresponds to a t -union graph.) Our goal is to schedule as many jobs as possible such that the total demand from any linear resource at any given moment is not more than one. The demand in this case can model transmission rate. For more details and applications, such as genomic sequence similarity [5], see [1].

Previous Results. Bar-Yehuda et al. [1] showed that the class of degree-3 graphs is contained in the class of t -union graphs. Since *maximum independent set* is APX-hard on degree-3 graphs [6,7], MWIS on t -interval graphs is also APX-hard. They proved that the k -dimensional matching problem is equivalent to MWIS in the special class of k -union graphs of unit segments. They also pointed out that k -dimensional matching cannot be approximated within an $O(k/\log k)$ ratio unless $P=NP$ [8], while the best known approximation ratio is $k/2 + \epsilon$, for any $\epsilon > 0$ [9]; and that 3-dimensional matching is APX-hard [10]. Note that it is NP-complete to determine whether a given graph is t -interval [11], or t -union [12], for any fixed $t \geq 2$. However, the hardness results remain true with respect to the problem of scheduling t -intervals with (or without) demands, since the constructions in [1] are made using t -interval scheduling instances.

Bar-Yehuda et al. [1] presented a $2t$ -approximation algorithm for MWIS in t -interval graphs that uses a new extension of local ratio [13] called *fractional local ratio*. The novelty of the fractional approach is that, in weight decomposition steps, the construction of a new weight function is based on an optimal solution to an LP relaxation of the original problem instance, and the analysis compares the weight of the solution returned to the weight of this optimal solution.

Our Results. A $6t$ -approximation algorithm for the problem of scheduling t -intervals with demands is given in Sect. 2. The algorithm is based on a novel and non-standard extension of the *primal-dual schema* we call *fractional primal-dual*. The first step in a fractional primal-dual r -approximation algorithm is to compute an optimal solution to an LP relaxation of the problem. Let P be the LP relaxation, and let x^* be an optimal solution of P . Next, as usual in primal-dual algorithms (see, e.g. [14]), the algorithm produces an integral primal solution x and a dual solution y , such that the value of y divided by r bounds the weight of x . However, in contrast to other primal-dual algorithms, y is not a solution to the dual of P . The algorithm produces a new LP, denoted by P' , that has the same objective function as P , but contains inequalities that may not be valid with respect to the original problem. The dual solution y is a feasible solution of the dual of P' . x is r -approximate, since we make sure that x^* is a feasible solution of P' , and therefore the optimum value of P' is not less than the optimum value of P . A general description of fractional primal-dual is given in Sect. 3.

The relation between fractional primal-dual and fractional local ratio is discussed in Sect. 4. We show that the former is the primal-dual manifestation of the latter. The connection between fractional primal-dual and fractional local ratio is based on the connection between the two methods in their standard forms [15].

We also present a fractional local ratio interpretation of the $6t$ -approximation algorithm for the problem of scheduling t -intervals with demands.

Bar-Noy et al. [14] distinguish between two types of resources, *fungible* and *non-fungible*. In a multimedia-on-demand system the resource is the bandwidth of the system. Such a resource is called fungible, since the *identity* of the bandwidth allocated to a specific movie is irrelevant. On the other hand, there are scenarios in which a specific portion of the resource is allocated to each job, e.g., memory allocation in a multi-threaded programming environment. Such resources are called non-fungible. Continuity is another issue. In some cases, such as memory allocation, the allocation of the resource must be *contiguous*. In the full version of this paper we consider the problem of scheduling t -intervals with demands where the allocation is *contiguous* and the resource is *non-fungible*. In terms of t -union graphs (Fig. 1), this means we are required to pack $(t + 1)$ -dimensional boxes without breaking them. We present a bi-criteria approximation algorithm that computes $4t$ -approximate solutions that may need up to 4 times the given amount of resource.

Related Work. Several approximation frameworks that use the primal-dual schema were published. Goemans and Williamson [16] presented an algorithm for a wide family of *network design* problems. They proposed a more general framework in [17]. A survey by Williamson [18] describes the primal-dual schema and several extensions of the primal-dual approach. Bertsimas and Teo [19] proposed a primal-dual framework for covering problems. As in [16] this framework enforces the primal complementary slackness conditions while relaxing the dual conditions. However, in contrast to previous studies, Bertsimas and Teo [19] express each advancement step as the construction of a single valid inequality, and an increase of the corresponding dual variable. Bar-Yehuda and Rawitz [15] presented a primal-dual framework that extends the one in [19]. In the analyses of both [19] and [15] it was convenient to define a new LP that contains the inequalities used by the algorithm. Since this new LP relaxes the original program, an r -approximation with respect to it is also an r -approximation with respect to the original program.

As pointed out by Williamson [18] several primal-dual algorithms were devised by first constructing a local ratio algorithm, and then transforming it into a primal-dual algorithm. Bafna et al. [20] extended the *local ratio technique* [13], and obtained a 2-approximation algorithm for the *feedback vertex set* problem. This work and the algorithm from [21] were essential in the design of primal-dual approximation algorithms for *feedback vertex set* [22]. Bar-Noy et al. [14] developed local ratio approximation algorithms for resource allocation and scheduling problems. A primal-dual interpretation was given as well. Bar-Yehuda and Rawitz [15] proved that the two methods in their standard forms are equivalent.

The fractional local ratio technique was also used by Lewin-Eytan et al. [23].

Definitions and Notation. Given a t -interval scheduling instance we denote the set of jobs by J . For $j \in J$, $N(j)$ is the set of jobs that are in conflict with j , and $N[j]$ is the set of jobs that are in conflict with j including j , i.e., $N[j] = N(j) \cup \{j\}$. (Recall that two jobs are in conflict if any of their segments intersect.)

We write $I \in j$ if I is one of the segments of j . For a segment $I \in j$, the set $R[I] \subseteq J$ contains every job k such that there exists a segment $I' \in k$ that contains the right endpoint of I (including j).

We denote the optimum value of a given problem instance by OPT . $\text{OPT}(\Pi)$ denotes the optimum of Π , where Π is usually an LP. Given a schedule S , $w(S)$ is the weight of S , i.e., $w(S) = \sum_{j \in S} w(j)$. Throughout the paper $w(j)$ denotes the weight of job j , while w_i denotes the i th weight function. For example, $w_1(j)$ is the weight of job j with respect to the weight function w_1 .

2 6t-Approximation Algorithm

The problem of scheduling t -intervals with demands can be formalized as follows:

$$\begin{aligned} & \max \sum_{j \in J} w(j)x_j \\ \text{s.t. } & \sum_{k \in R[I]} d_k x_k \leq 1 \quad \forall j \in J, \forall I \in j \\ & x_j \in \{0, 1\} \quad \forall j \in J \end{aligned}$$

The LP relaxation is obtained by replacing the integrality constraints by: $0 \leq x_j \leq 1$ for every j . We denote it by P .

The unit demand version of the following lemma was proven in [1].

Lemma 1. *Let x be a feasible solution of P . Then, there exists a job ℓ such that $\sum_{j \in N[\ell]} d_j x_j \leq 2t$.*

Proof. In order to prove this lemma, it is enough to show that

$$\sum_k \sum_{j \in N[k]} d_k x_k \cdot d_j x_j = \sum_k d_k x_k \sum_{j \in N[k]} d_j x_j \leq 2t \cdot \sum_k d_k x_k .$$

If j_1 and j_2 are in conflict then $j_1 \in N[j_2]$ and $j_2 \in N[j_1]$. Hence, the term $d_{j_1} x_{j_1} \cdot d_{j_2} x_{j_2}$ is counted twice in the sum on the LHS for every j_1, j_2 that are in conflict. Moreover, if j_1 and j_2 are in conflict then either there exists a segment $I_1 \in j_1$ such that $j_2 \in R(I_1)$, or there exists a segment $I_2 \in j_2$ such that $j_1 \in R(I_2)$. Thus,

$$\sum_k \sum_{j \in N[k]} d_k x_k \cdot d_j x_j \leq 2 \cdot \sum_k \sum_{I \in k} \sum_{j \in R[I]} d_k x_k \cdot d_j x_j .$$

Since x is a feasible solution of P ,

$$\sum_{j \in R[I]} d_k x_k \cdot d_j x_j = d_k x_k \sum_{j \in R[I]} d_j x_j \leq d_k x_k .$$

Therefore,

$$\sum_k \sum_{j \in N[k]} d_k x_k \cdot d_j x_j \leq 2 \cdot \sum_k \sum_{I \in k} d_k x_k = 2t \cdot \sum_k d_k x_k .$$

and we are done. □

To approximate the problem we consider the following two special cases: (1) all jobs are *wide*, i.e., $d_j > \frac{1}{2}$ for all j , and (2) all jobs are *narrow*, i.e., $d_j \leq \frac{1}{2}$ for all j . In the case of wide jobs the problem reduces to the special case in which all jobs have demand 1 since no pair of conflicting jobs may be scheduled together. Thus, it can be approximated using the $2t$ -approximation algorithm from [1]. In the sequel we present a $4t$ -approximation algorithm for narrow jobs. To solve the problem in the general case we solve it separately for the narrow jobs, and for the wide jobs, and return the solution of greater weight. Since either the optimum of the narrow jobs is at least $\frac{2}{3}$ OPT or the optimum for the wide jobs is at least $\frac{1}{3}$ OPT, the schedule returned is $6t$ -approximate.

The first step in $4t$ -approximation algorithm for narrow instances is to obtain an optimal solution of P, denoted by x^* . The second step is given below.

Algorithm FPD(J, w, x^*)

1. $i \leftarrow 1$
2. $J_1 \leftarrow J$
3. While $J_i \neq \emptyset$ do:
4. Let $\ell_i = \operatorname{argmin}_{\ell \in J_i} \sum_{j \in N[\ell] \cap J_i} d_j x_j^*$
5. Construct Inequality i :

$$(1 - d_{\ell_i})z_{\ell_i} + \sum_{j \in N(\ell_i) \cap J_i} d_j z_j \leq 1 - 2d_{\ell_i} + 2t$$
6. Increase y_i until

$$(1 - d_{\ell_i})y_i + \sum_{k: \ell_i \in N(\ell_k) \cap J_k} d_{\ell_i} y_k = w(\ell_i)$$
7. $J_{i+1} \leftarrow \left\{ j : \sum_{k: j \in N(\ell_k) \cap J_k} d_j y_k < w(j) \right\}$
8. $i \leftarrow i + 1$
9. $S \leftarrow \emptyset$
10. While $i > 1$ do:
11. $i \leftarrow i - 1$
12. If $S \cup \{\ell_i\}$ is a feasible solution do: $S \leftarrow S \cup \{\ell_i\}$
13. Return S

The running time of algorithm is polynomial since in each iteration at least one job (Job ℓ_i) is eliminated. Moreover, the computed solution is feasible due to Lines 9-12. It remains to show that this schedule is $4t$ -approximate.

The following LP contains the inequalities constructed by the algorithm:

$$\begin{aligned}
 & \max \sum_j w(j)z_j \\
 \text{s.t.} & (1 - d_{\ell_i})z_{\ell_i} + \sum_{j \in N(\ell_i) \cap J_i} d_j z_j \leq c_i \quad \forall i \in \{1, \dots, m\} \\
 & z_j \geq 0 \quad \forall j
 \end{aligned} \tag{P'}$$

where m is the number of iterations, and $c_i \triangleq 1 - 2d_{\ell_i} + 2t$. The dual of P' is:

$$\begin{aligned}
 \min \quad & \sum_{i=1}^m c_i y_i \\
 \text{s.t.} \quad & (1 - d_{\ell_i})y_i + \sum_{k: \ell_i \in N(\ell_k) \cap J_k} d_{\ell_i} y_k \geq w(\ell_i) \quad \forall i \in \{1, \dots, m\} \\
 & \sum_{k: j \in N(\ell_k) \cap J_k} d_j y_k \geq w(j) \quad \forall j \notin \{\ell_1, \dots, \ell_m\} \\
 & y_i \geq 0 \quad \forall i \in \{1, \dots, m\}
 \end{aligned} \tag{D'}$$

Let x be the incidence vector of the schedule S returned by the algorithm. Also, let y be the vector constructed by the algorithm. We show that: (1) y is a feasible solution of D' , (2) x^* is a feasible solution of P' , and (3) $w \cdot x \geq c \cdot y/4t$. When putting it all together we get that: $w \cdot x \geq c \cdot y/4t \geq w \cdot x^*/4t = \text{OPT}(P)/4t \geq \text{OPT}/4t$, which means that x is $4t$ -approximate.

To see that y is a feasible solution of D' observe that by the termination condition of the first while loop (Line 3) all the constraints in D' are satisfied. Next we show that x^* is a feasible solution of P' . Consider Inequality i . Let x^i be the projection of x^* on J_i . That is, $x_j^i = x_j^*$ if $j \in J_i$, and $x_j^i = 0$, otherwise. x^i is a feasible solution of P , therefore by Lemma 1 $\sum_{j \in N[\ell_i] \cap J_i} d_j x_j^i \leq 2t$. Hence,

$$\begin{aligned}
 (1 - d_{\ell_i})x_{\ell_i}^* + \sum_{j \in N(\ell_i) \cap J_i} d_j x_j^* &= (1 - d_{\ell_i})x_{\ell_i}^i + \sum_{j \in N(\ell_i) \cap J_i} d_j x_j^i \\
 &= (1 - 2d_{\ell_i})x_{\ell_i}^i + \sum_{j \in N[\ell_i] \cap J_i} d_j x_j^i \\
 &\leq 1 - 2d_{\ell_i} + 2t .
 \end{aligned}$$

Since x^* satisfies the inequalities in P' , we conclude that x^* is a feasible solution of P' , and that $w \cdot x^* \leq c \cdot y$.

Finally, we prove that $w \cdot x \geq c \cdot y/4t$. First,

$$\begin{aligned}
 \sum_j w(j)x_j &= \sum_i x_{\ell_i} \left((1 - d_{\ell_i})y_i + \sum_{k: \ell_i \in N(\ell_k) \cap J_k} d_{\ell_i} y_k \right) + \\
 &\quad \sum_{j \notin \{\ell_1, \dots, \ell_m\}} x_j \sum_{k: j \in N(\ell_k) \cap J_k} d_j y_k
 \end{aligned}$$

since $x_j = 1$ if $j = \ell_i$ for some i and the corresponding constraint is tight, and otherwise $x_j = 0$ (Line 6). By changing the summation order we get that

$$\sum_j w(j)x_j = \sum_i y_i \left((1 - d_{\ell_i})x_{\ell_i} + \sum_{j \in N(\ell_i) \cap J_i} d_j x_j \right) .$$

Observe that either $\ell_i \in S$, or the total demand of jobs in $N(\ell_i) \cap J_i \cap S$ is more than $1 - d_{\ell_i}$ (otherwise ℓ_i would have been added to S in Line 12). Hence,

$$\sum_j w(j)x_j \geq \sum_i y_i (1 - d_{\ell_i}) = \sum_i (1 - 2d_{\ell_i} + 2t) \cdot y_i \frac{1 - d_{\ell_i}}{1 - 2d_{\ell_i} + 2t} \geq \frac{1}{4t} \cdot \sum_i c_i y_i$$

where the last inequality is because $f(z) = \frac{1-2z+2t}{1-z}$ is an increasing function for $0 \leq z \leq \frac{1}{2}$ and $t \geq 1$, and that $f(\frac{1}{2}) = 4t$.

3 Using Fractional Primal-Dual

This section is written in terms of maximization problems. Similar arguments can be made in the minimization case.

Consider the following linear program, denoted by P, and its dual:

$$\begin{array}{ll} \max & \sum_{j=1}^n w(j)x_j \\ \text{s.t.} & \sum_{j=1}^n a_{ij}x_j \leq b_i \quad \forall i \in \{1, \dots, m\} \\ & x_j \geq 0 \quad \forall j \in \{1, \dots, n\} \end{array} \quad \begin{array}{ll} \min & \sum_{i=1}^m b_i y_i \\ \text{s.t.} & \sum_{i=1}^m a_{ij}y_i \geq w(j) \quad \forall j \in \{1, \dots, n\} \\ & y_i \geq 0 \quad \forall i \in \{1, \dots, m\} \end{array}$$

A primal-dual r -approximation algorithm constructs an integral primal solution x and a dual solution y , such that $w \cdot x \geq b \cdot y/r$. It follows, by weak duality, that x is r -approximate. One way to find such a pair of primal and dual solutions is to focus on pairs (x, y) satisfying the following *relaxed complementary slackness* conditions (for an appropriately chosen r):

$$\begin{array}{l} \text{Primal:} \quad \forall j, x_j > 0 \Rightarrow \sum_{i=1}^m a_{ij}y_i = w(j). \\ \text{Relaxed Dual:} \quad \forall i, y_i > 0 \Rightarrow b_i/r \leq \sum_{j=1}^n a_{ij}x_j \leq b_i. \end{array}$$

These conditions imply that x is r -approximate since

$$\sum_{j=1}^n w(j)x_j = \sum_{j=1}^n x_j \cdot \sum_{i=1}^m a_{ij}y_i = \sum_{i=1}^m y_i \cdot \sum_{j=1}^n a_{ij}x_j \geq \frac{1}{r} \cdot \sum_{i=1}^m b_i y_i.$$

Typically, such an algorithm begins with the solutions $x = 0$ and $y = 0$. It then iteratively increases dual variables until y becomes feasible. Afterwards, it adds as many elements that correspond to tight dual constraints to x as possible, ending with a primal/dual pair satisfying the relaxed conditions. This design method is commonly referred to as the *primal-dual schema*.

Our algorithm deviates from the standard primal-dual approach. First, the algorithm is based on an optimal fractional solution, x^* . Another difference is that our algorithm constructs new primal constraints during execution that are not necessarily valid for the original problem instance. That is, a feasible solution may not satisfy these constraints. However, we make sure that x^* satisfies them. Note that the construction of new primal constraints during execution was previously used by Bertsimas and Teo [19], and subsequently by Bar-Yehuda and Rawitz [15]. However, in both papers the algorithms construct constraints that are valid with respect to the original problem instance. The constraints that are constructed by our algorithm induce a new linear program P' and a dual program D' . It produces a primal solution x for the original problem instance, and a solution y for D' , whose value divided by r bounds the weight of x . Since x^* is in the feasible set of P' , y serves as an upper bound to the weight of x^* . Therefore, x is r -approximate.

Let $P' = \max \{w \cdot z : Az \leq c\}$. When using fractional primal-dual the solutions x^*, x, y produced by the algorithm satisfy the following *fractional relaxed* complementary slackness conditions:

$$\begin{aligned} \text{Primal:} \quad & \forall j, x_j > 0 \Rightarrow \sum_{i=1}^m a_{ij}y_i = w(j) \\ \text{Relaxed Dual:} \quad & \forall i, y_i > 0 \Rightarrow c_i/r \leq \sum_{j=1}^n a_{ij}x_j \text{ and } \sum_{j=1}^n a_{ij}x_j^* \leq c_i \end{aligned}$$

Note that y is actually positive, since P' contains inequalities that were used by the algorithm. Since x^*, x, y satisfy the above conditions we get that:

$$w \cdot x = \sum_{j=1}^n w(j)x_j = \sum_{j=1}^n x_j \cdot \sum_{i=1}^m a_{ij}y_i = \sum_{i=1}^m y_i \cdot \sum_{j=1}^n a_{ij}x_j \geq \sum_{i=1}^m \frac{1}{r}c_iy_i = \frac{c \cdot y}{r}$$

which this means that x is r -approximate, since $c \cdot y \geq \text{OPT}(P') \geq w \cdot x^* = \text{OPT}(P) \geq \text{OPT}$.

4 Fractional Local Ratio

In this section we present a fractional local ratio interpretation of the $4t$ -approximation algorithm for narrow jobs, and study the connection between fractional primal-dual and fractional local-ratio. We start with a brief description of the fractional local ratio technique.

A typical local ratio algorithm is recursive. It constructs, in each recursive call, a new weight function w_1 . In essence, a local ratio analysis consists of comparing, at each level of the recursion, the solution found in that level to an optimal solution for the problem instance passed to that level, where the comparison is made with respect to w_1 and with respect to $w - w_1$. Thus, in each level of the recursion there are potentially two optima (one with respect to w_1 and one with respect to $w - w_1$) against which the solution is compared, and in addition, different optima are used at different recursion levels. The *fractional local ratio* paradigm takes a different approach. It uses a single solution x^* to the *original* problem instance as the yardstick against which all intermediate solutions (at all levels of the recursion) are compared. In fact, x^* is not even feasible for the original problem instance but rather for a relaxation of it. Typically, x^* will be an optimal fractional solution to an LP relaxation.

Fractional local ratio [1] is based on a *fractional* version of the Local Ratio Theorem [13,24].

Theorem 1 (Fractional Local Ratio). *Let $w, w_1, w_2 \in \mathbb{R}^n$ be weight functions such that $w = w_1 + w_2$. Let x^* and x be vectors in \mathbb{R}^n such that $w_1 \cdot x \geq w_1 \cdot x^*/r$ and $w_2 \cdot x \geq w_2 \cdot x^*/r$. Then, $w \cdot x \geq w \cdot x^*/r$ as well.*

4.1 Fractional Local Ratio Interpretation

Let x^* be an optimal fractional solution. We now run the recursive algorithm described next to obtain a feasible schedule. The algorithm contains problem-size reduction steps, as do local ratio algorithms for packing problems (e.g., [14]). The initial call is **FLR**(J, w).

Algorithm FLR(J, w)

1. If $J = \emptyset$, return \emptyset
2. Let $\ell = \operatorname{argmin}_{\ell \in J} \sum_{j \in N[\ell]} d_j x_j^*$
3. $\epsilon \leftarrow w(\ell)/(1 - d_\ell)$
4. Define the weight functions $w_1(j) = \epsilon \cdot \begin{cases} 1 - d_\ell & j = \ell, \\ d_j & j \in N(\ell), \\ 0 & \text{otherwise,} \end{cases}$
and $w_2 = w - w_1$
5. Let J^+ be the set of positive weighted jobs
6. $S' \leftarrow \mathbf{FLR}(J^+, w_2)$
7. If $S' \cup \{\ell\}$ is a feasible solution, return $S = S' \cup \{\ell\}$
8. Else, return $S = S'$

For the analysis, let x denote the incidence vector of the schedule S returned by the algorithm. We assume that x (and w) is of size n , where n is the number of jobs in the original problem instance. This way we can compare x to x^* . We claim that $w \cdot x \geq \frac{1}{4t} w \cdot x^*$. The proof is by induction on the recursion. In the base case ($J = \emptyset$) we have $S = \emptyset$, and therefore $w \cdot x = 0$. Since the weights in the recursive base are non-positive we get that $w \cdot x^* \leq 0$. Thus, $w \cdot x \geq w \cdot x^*$. For the inductive step, let x' be the incidence vector of S' (obtained in Line 6). By the inductive hypothesis $w_2 \cdot x' \geq \frac{1}{4t} w_2 \cdot x^*$. Moreover, the weight of ℓ with respect to w_2 is zero, and therefore $w_2 \cdot x = w_2 \cdot x'$. This means that $w_2 \cdot x \geq \frac{1}{4t} w_2 \cdot x^*$. Next, we show that $w_1 \cdot x \geq \frac{1}{4t} w_1 \cdot x^*$. This completes the proof since this means that by the Fractional Local Ratio Theorem $w \cdot x \geq \frac{1}{4t} w \cdot x^*$.

It remains to show that $w_1 \cdot x \geq \frac{1}{4t} w_1 \cdot x^*$. Observe that the projection of x^* on the current instance J is feasible. Hence,

$$\begin{aligned} \sum_j w_1(j) x_j^* &= \epsilon(1 - d_\ell) x_\ell + \sum_{j \in N(\ell) \cap J} \epsilon d_j x_j^* \\ &= \epsilon(1 - 2d_\ell) x_\ell + \epsilon \sum_{j \in N[\ell] \cap J} d_j x_j^* \\ &\leq \epsilon(1 - 2d_\ell + 2t) \end{aligned}$$

where the inequality is due to Lemma 1. On the other hand, we show that $w_1 \cdot x \geq \epsilon(1 - d_\ell)$. If ℓ is added to S (in Line 8) then this is obviously true. Otherwise, if ℓ is not added to S , the total demand of jobs in $N(\ell) \cap S$ is more than $1 - d_\ell$, since otherwise ℓ would have been added to S . $w \cdot x \geq \frac{1}{4t} w \cdot x^*$, since $f(z) = \frac{1-2z+2t}{1-z}$ is an increasing function for $z \in [0, \frac{1}{2}]$ and $t \geq 1$, and $f(\frac{1}{2}) = 4t$.

4.2 Connection to Fractional Local Ratio

In [15] Bar-Yehuda and Rawitz showed that the primal-dual schema and local ratio technique in their standard forms are equivalent. This equivalence is based

on the fact that increasing a dual variable by ϵ is equivalent to subtracting the weight function obtained by multiplying the coefficients of the corresponding primal constraint by ϵ from the primal objective function. A similar equivalence exists between both fractional methods. For example, the weight function that is used in the i th recursive call of Algorithm **FLR** is equal to the vector of coefficients of the inequality that was constructed in the i th iteration of Algorithm **FPD** multiplied by y_i . Generally, in each recursive call of a fractional local ratio algorithm we utilize a weight function w_1 such that $w_1 \cdot x^* \leq r w_1 \cdot x$. Implicitly, this means that there exists some c_i such that $w_1 \cdot x \geq c_i/r$ and $w_1 \cdot x^* \leq c_i$. Thus, the inequality $w_1 \cdot z \leq c_i$ can be used by a fractional primal-dual algorithm. (Note that we do not need to know the value of c_i .) For the other direction, an inequality $\alpha \cdot z \leq \beta$ corresponds to the weight function $\epsilon \cdot \alpha$ where ϵ is the value of the corresponding dual variable.

Acknowledgments. We thank Guy Even, Ari Freund, Seffi Naor, and Moni Shahar for helpful discussions.

References

1. Bar-Yehuda, R., Halldórsson, M.M., Naor, J., Shachnai, H., Shapira, I.: Scheduling split intervals. In: 13th Annual Symposium on Discrete Algorithms. (2002) 732–741
2. Golumbic, M.C.: Algorithmic Graph Theory and Perfect Graphs. Academic Press (1980)
3. Halldórsson, M.M., Rajagopalan, S., Shachnai, H., Tomkins, A.: Scheduling multiple resources. Manuscript (1999)
4. Rotem, D.: Analysis of disk arm movement for large sequential reads. In: 11th ACM Symposium on Principles of Database Systems. (1992) 47–54
5. Bafna, V., Narayanan, B.O., Ravi, R.: Nonoverlapping local alignments (weighted independent sets of axis parallel rectangles). *Disc. Appl. Math.* **71** (1996) 41–53
6. Berman, P., Fujito, T.: Approximating independent sets in degree 3 graphs. In: 4th Workshop on Algorithms and Data Structures. Volume 995 of LNCS. (1995) 449–460
7. Halldórsson, M.M., Yoshihara, K.: Greedy approximations of independent sets in low degree graphs. In: 6th Annual International Symposium on Algorithms And Computation. Volume 1004 of LNCS. (1995) 152–161
8. Hazan, E., Safra, S., Schwartz, O.: On the hardness of approximating k -dimensional matching. In: 6th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems. Volume 2764 of LNCS. (2003) 83–97
9. Hurkens, C.A.J., Schrijver, A.: On the size of systems of sets every t of which have an SDR, with an application to the worst-case ratio of heuristics for packing problems. *SIAM Journal on Discrete Mathematics* **2** (1989) 68–72
10. Papadimitriou, C.H., Yannakakis, M.: Optimization, approximation and complexity classes. *Journal of Computer and System Sciences* **43** (1991) 425–440
11. West, D.B., Shmoys, D.B.: Recognizing graphs with fixed interval number is NP-complete. *Discrete Applied Mathematics* **8** (1984) 295–305
12. Gyárfás, A., West, D.B.: Multitrack interval graphs. In: 26th SE Intl. Conf. Graph Th. Comb. Comput. Volume 109 of Congr. Numer. (1995) 109–116

13. Bar-Yehuda, R., Even, S.: A local-ratio theorem for approximating the weighted vertex cover problem. *Annals of Discrete Mathematics* **25** (1985) 27–46
14. Bar-Noy, A., Bar-Yehuda, R., Freund, A., Naor, J., Shieber, B.: A unified approach to approximating resource allocation and scheduling. *J. ACM* **48** (2001) 1069–1090
15. Bar-Yehuda, R., Rawitz, D.: On the equivalence between the primal-dual schema and the local ratio technique. *SIAM J. on Disc. Math.* (2005) To appear.
16. Goemans, M.X., Williamson, D.P.: A general approximation technique for constrained forest problems. *SIAM J. on Comp.* **24** (1995) 296–317
17. Goemans, M.X., Williamson, D.P.: The primal-dual method for approximation algorithms and its application to network design problems. In Hochbaum, D.S., ed.: *Approximation Algorithms for NP-Hard Problem*. PWS Publishing Company (1997)
18. Williamson, D.P.: The primal dual method for approximation algorithms. *Mathematical Programming* **91** (2002) 447–478
19. Bertsimas, D., Teo, C.: From valid inequalities to heuristics: A unified view of primal-dual approximation algorithms in covering problems. *Oper. Res.* **46** (1998) 503–514
20. Bafna, V., Berman, P., Fujito, T.: A 2-approximation algorithm for the undirected feedback vertex set problem. *SIAM J. on Disc. Math.* **12** (1999) 289–297
21. Becker, A., Geiger, D.: Optimization of Pearl’s method of conditioning and greedy-like approximation algorithms for the vertex feedback set problem. *Artificial Intelligence* **83** (1996) 167–188
22. Chudak, F.A., Goemans, M.X., Hochbaum, D.S., Williamson, D.P.: A primal-dual interpretation of recent 2-approximation algorithms for the feedback vertex set problem in undirected graphs. *Oper. Res. Lett.* **22** (1998) 111–118
23. Lewin-Eytan, L., Naor, J., Orda, A.: Admission control in networks with advance reservations. *Algorithmica* **40** (2004) 293–403
24. Bar-Yehuda, R.: One for the price of two: A unified approach for approximating covering problems. *Algorithmica* **27** (2000) 131–144

An Approximation Algorithm for the Minimum Latency Set Cover Problem

Refael Hassin¹ and Asaf Levin²

¹ Department of Statistics and Operations Research,
Tel-Aviv University, Tel-Aviv, Israel

`hassin@post.tau.ac.il`

² Department of Statistics, The Hebrew University, Jerusalem, Israel
`levinas@mscc.huji.ac.il`

Abstract. The input to the MINIMUM LATENCY SET COVER PROBLEM consists of a set of jobs and a set of tools. Each job j needs a specific subset S_j of the tools in order to be processed. It is possible to install a single tool in every time unit. Once the entire subset S_j has been installed, job j can be processed instantly. The problem is to determine an order of job installations which minimizes the weighted sum of job completion times. We show that this problem is NP-hard in the strong sense and provide an ϵ -approximation algorithm. Our approximation algorithm uses a framework of approximation algorithms which were developed for the minimum latency problem.

Keywords: Minimum sum set cover, minimum latency, approximation algorithm.

1 Introduction

The MINIMUM LATENCY SET COVER PROBLEM (MLSC) is defined as follows: Let $\mathcal{J} = \{J_1, J_2, \dots, J_m\}$ be a set of *jobs* to be processed by a factory. A job J_i has non-negative weight w_i . Let $\mathcal{T} = \{t_1, t_2, \dots, t_n\}$ be a set of *tools*. Job j is associated with a nonempty subset $S_j \subseteq \mathcal{T}$. Each time unit the factory can install a single tool. Once the entire tool subset S_j has been installed, job j can be processed instantly. The problem is to determine the order of tool installation so that the weighted sum of job completion times is minimized.

We rephrase MLSC as a variant of the MINIMUM SET COVER PROBLEM in the following way. Given a set of items \mathcal{J} and a collection of subsets S_1, \dots, S_m of a ground set \mathcal{T} . We want to order the elements of \mathcal{T} so that when each item j incurs a cost that equals its weight times the last time when an element of S_j appears in the order. The goal is to minimize the total cost.

Feige et al. [6] considered the related problem where each job incurs a cost equal to the *first time* where an element of S_j appears in the order. They proved that a greedy algorithm is a 4-approximation algorithm, and showed that unless $P = NP$ there is no polynomial time algorithm with an approximation ratio $4 - \epsilon$ where $\epsilon > 0$.

The MINIMUM LATENCY PROBLEM is defined as follows: we are given a set of n points. A feasible solution is a Hamiltonian path that traverses the points. Each point j (of the n points) incurs a cost that equals the total length of the prefix of the path from its beginning towards the (first) appearance of j in the path. The goal is to find the path that minimizes the total incurred cost. Our approximation algorithm follows similar arguments to the ones used by Goemans and Kleinberg [7] and Archer, Levin and Williamson [1] for the minimum latency problem in a metric space.

The DENSEST k -SUBGRAPH PROBLEM is defined as follows. We are given a graph $G = (V, E)$ where each edge e has a non-negative weight w_e . The goal is to pick k vertices $U = \{v_1, \dots, v_k\} \subseteq V$ such that $\sum_{(u,v) \in E \cap (U \times U)} w_{(u,v)}$ is maximized. This problem is known to be NP-hard even if all weights are equal, and the current best approximation algorithm for it [5] has an approximation ratio of $O(n^{-\frac{1}{3}})$.

Given a hyper-graph $G = (V, E)$ and a subset of vertices $U \subseteq V$, a hyper-edge e is induced by U if $e \subseteq U$. We will use in our algorithm a new problem named the DENSEST k -SUB-HYPER-GRAPH PROBLEM which generalizes the densest k -subgraph problem to hyper-graphs where each hyper-edge e contributes to the goal function if and only if it is induced by U . The densest k -sub-hyper-graph problem is at least as difficult as the densest k -subgraph problem. However, we are not aware of any prior results on this new variant.

Paper Preview. In Section 2 we prove that MLSC is NP-hard in the strong sense. In Section 3 we develop a basic approximation algorithm assuming that the densest k -sub-hyper-graph problem can be solved in polynomial time. Afterwards, in Section 4 we show how to remove this assumption and obtain our e -approximation algorithm. In Section 5 we discuss bad examples for our analysis.

2 NP-Hardness of MLSC

The following problem is known as the MINIMUM WEIGHTED SUM OF JOB COMPLETION TIMES ON A SINGLE MACHINE UNDER PRECEDENCE CONSTRAINTS WITH UNIT PROCESSING TIMES PROBLEM: given m jobs $\{j_1, j_2, \dots, j_m\}$ each has a unit processing time and a non-negative weight w_j , and precedence constraints between the jobs in the form of an acyclic digraph G . A feasible schedule must satisfy that for all $(j, k) \in A$, the machine starts to process job k only after job s is finished (not necessarily immediately after). The goal is to find a feasible schedule (that satisfies the precedence constraints) that minimizes the weighted sum of job completion times. In the scheduling notation this problem is denoted as $1|prec; p_j = 1| \sum w_j C_j$. This problem is known to be NP-hard in the strong sense (see [12,13] and also [4,10]), and there is a 2-approximation algorithm for it [8].

Theorem 1. *MLSC is NP-hard in the strong sense.*

Proof. We will describe a reduction from $1|prec; p_j = 1| \sum w_j C_j$. We are given an instance I defined as $\mathcal{J} = \{j_1, j_2, \dots, j_n\}$, such that j_i has weight w_i , and an

acyclic directed graph G defining the precedence constraints. For $i = 1, \dots, n$, let P_i denote the set of all predecessors of j_i . We define an instance I' to MLSC in the following way: Define a job j'_i and a tool t_i for $i = 1, \dots, n$. Define for job j'_i the job set $S'_i = \{t_i\} \cup_{q \in P_i} S'_q$. Finally, we assign a weight w_i to j'_i . Denote the resulting instance of MLSC by I' . Since problem $1|prec; p_j = 1| \sum w_j C_j$ is NP-hard in the strong sense we can restrict ourselves to instances of $1|prec; p_j = 1| \sum w_j C_j$ in which the weights are polynomially bounded, and therefore I' has polynomial size even if the numbers are represented in unary. To prove the theorem, it suffices to show that given a solution of cost C to I there is a solution to I' of cost at most C , and vice versa.

First assume that π is a feasible schedule to I with cost C . Then, at time unit i we install tool $t_{\pi(i)}$. Since π satisfies the precedence constraints, we conclude that at time i the set $S'_{\pi(i)}$ has been installed, and therefore we gain a weight of $w_{\pi(i)}$. This is exactly the weight of $\pi(i)$ such that $C_{\pi(i)} = i$ (the completion time of job $\pi(i)$ is i and this term is multiplied by $w_{\pi(i)}$ in the objective function $\sum w_j C_j$). Therefore, the resulting solution costs C as well.

Consider now a solution π' to I' , i.e., at time i we install $t_{\pi'(i)}$. W.l.o.g. we assume that prior to the i -th time unit we have already installed the sets $S_{i'}$ for all i' such that $(i', \pi'(i)) \in G$. This assumption is w.l.o.g. because otherwise at time unit i we cannot complete the processing of any job, and we can exchange the positions of the tools in π' without additional cost. With this assumption, π' is a feasible solution to I , and as in the previous case, π' has a cost of at most C' . □

Remark 1. MLSC is NP-hard even for unweighted instances. The changes that are needed in the construction is to replace a job with weight w_j by a family of w_j identical jobs, each with unit weight. Since MLSC is NP-hard in the strong sense the resulting instance has a polynomial size.

3 The Basic Approximation Algorithm

In this section, we assume that there is a polynomial time algorithm for the densest k -sub-hyper-graph problem. In the next section we will show how to remove this assumption. We follow similar arguments as used by Goemans and Kleinberg [7] for the minimum latency problem.

Our algorithm will make use of the values of the densest k -sub-hyper-graph for $k = 2, 3, \dots, n$, in the following auxiliary hyper-graph. The vertex set is \mathcal{T} , for each job j we will have a hyper-edge $e_j = S_j$ that is its tool set with weight w_j . Denote by V_1, V_2, \dots, V_n the resulting vertex-sets and denote by W_i the weight of hyper-edges induced by V_i . In other words, W_i is the maximum weight of jobs that can be processed (covered) in i units of time (by i tools).

Given an increasing set of indices

$$j_0 = 0 < j_1 < j_2 < \dots < j_t = n,$$

we define the *concatenated solution* as follows: for all $i = 0, \dots, t - 1$, at time $\sum_{k=0}^i j_k$ we finished installation of the tools of the $V_{j_0} \cup V_{j_1} \cup \dots \cup V_{j_i}$ and in the

next j_{i+1} time units we install the yet uninstalled tools of $V_{j_{i+1}}$ in an arbitrary order (perhaps leaving idle time until the end of this time period). A job j is served at time no later than $\min \left\{ \sum_{k=0}^i j_k : S_j \subseteq \bigcup_{k=0}^i V_k \right\}$. Consider the following upper bound on the cost of the concatenated solution. Suppose that the weight of jobs that are completed during the time interval $\left[\sum_{k=0}^{i-1} j_k + 1, \sum_{k=0}^i j_k \right]$, is v_i . Let $q_i = \sum_{l=0}^i v_l$ be the weight of jobs completed until $\sum_{k=0}^i j_k$. Denote by W the total weight of all the jobs, i.e., $W = \sum_{j=1}^m w_j$.

The set V_{j_i} adds at most j_i to the waiting time of each of the $W - q_{i-1}$ units of weights of jobs that were not processed until time $\sum_{k=0}^{i-1} j_k$. Thus, the total cost of the concatenated solution is at most

$$\sum_{i=1}^t (W - q_{i-1}) \cdot j_i \leq \sum_{i=1}^t (W - W_{i-1}) \cdot j_i, \tag{1}$$

where the inequality follows since by definition $q_i \geq W_i$ for all i .

Our algorithm for approximating MLSC is as follows:

Algorithm A:

1. For $k = 0, 1, 2, \dots, n$, compute V_k , an optimal densest k -sub-hyper-graph solution and its value W_k .
2. Let G be the graph on the vertex set $\{0, 1, 2, \dots, n\}$, such that, for all $i \leq j$, G has an arc from i to j with length $(W - W_i) \cdot j$.
3. Compute a shortest $0 - n$ path in G . Denote its length by σ and suppose that it goes through $j_0 = 0 < j_1 < \dots < j_t = n$.
4. Output the concatenated solution .

The next lemma follows from (1):

Lemma 1. *The cost of the concatenated solution is at most σ .*

Let opt denote the optimal solution cost and let σ denote the length of a shortest path in G .

Theorem 2.

$$\sigma \leq e \cdot opt.$$

Proof. To prove the theorem, we replace each job j by w_j unit weight jobs each having the same tool set S_j . Thus, the number of jobs is now W . This change clearly has no effect on opt or σ . Let OPT be an optimal solution and denote by l_k^* the time it takes OPT to finish the first k jobs, $k = 1, \dots, W$. Note that $1 \leq l_1^* \leq \dots \leq l_W^*$. We construct a $1 - n$ path in G and compare its length to $opt = \sum_{k=1}^W l_k^*$.

Fix $c > 1$ and $1 \leq L_0 < c$. For $i = 1, \dots, t$ let $j_i = \lfloor L_0 c^{i-1} \rfloor$ where $t = \min\{i : L_0 c^i \geq n\}$. We may also assume w.l.o.g. that every tool is needed for some job so that the total time for the process is n , and therefore $j_t = n$. Consider the path $j_0 = 0, j_1, \dots, j_t = n$ in G . Its length is $\sum_{i=1}^t (W - W_{j_{i-1}}) \cdot j_i$. Since $W_t = W$,

$$\begin{aligned} \sum_{i=1}^t (W - W_{j_{i-1}})j_i &= \sum_{i=1}^t j_i \sum_{r=i}^t (W_{j_r} - W_{j_{r-1}}) \\ &= \sum_{i=1}^t \left[(W_{j_i} - W_{j_{i-1}}) \cdot \sum_{r=1}^i j_r \right] \\ &= \sum_{k=1}^W \delta_k, \end{aligned}$$

where $\delta_k = \sum_{l=1}^i j_l$ for $W_{j_{i-1}} < k \leq W_{j_i}$.

Let $L_k = \min\{L_0c^i : L_0c^i \geq l_k^*\}$.¹ By definition, $l_k^* \leq L_k$. Let s_k be such that $L_k = L_0c^{s_k}$. Therefore,

$$\delta_k = \sum_{l=1}^{s_k} j_l \leq \sum_{l=1}^{s_k} L_0c^{l-1} = \sum_{l=1}^{s_k} \frac{L_k}{c^{s_k-l+1}} \leq L_k + \frac{L_k}{c} + \frac{L_k}{c^2} + \dots = \frac{L_k c}{c-1},$$

where the first equation holds by definition of δ_k , the first inequality holds by definition of j_l , the second equation holds because $L_k = L_0c^{s_k}$.

Let $L_0 = c^U$ where U is a random variable uniformly distributed over $[0, 1]$. This defines a random path whose expected length is $\sum_{k=1}^n E[\delta_k]$. Moreover, $E[\delta_k] \leq \frac{c}{c-1}E[L_k]$. We now compute $E[L_k]$. First, assume that $l_k^* \geq L_0$. Observe that $\frac{L_k}{l_k^*}$, is a random variable of the form c^Y , where $Y = \left\lceil \log_c \left(\frac{l_k^*}{L_0} \right) \right\rceil - \log_c \left(\frac{l_k^*}{L_0} \right)$ is a uniform random variable over $[0, 1]$. Hence,

$$E[L_k] = l_k^* E[c^Y] = l_k^* \int_0^1 c^x dx = l_k^* \frac{c-1}{\ln c}.$$

Even if $l_k^* < L_0$ then $L_k = L_0 \leq c$ and $E[L_k] = E[L_0] = \frac{c-1}{\ln c} \leq l_k^* \frac{c-1}{\ln c}$, where the last inequality holds because $l_k^* \geq 1$.

Thus,

$$E[\delta_k] \leq \frac{c}{\ln c} l_k^*.$$

Therefore, the expected length of our random path is at most $\frac{c}{\ln c}$ times $\sum l_k^*$. Hence, the length of a shortest path is at most $\frac{c}{\ln c}$ times $\sum l_k^*$. This value is optimized by setting c to be the root of $\ln(c) - 1 = 0$, and hence $c = e \sim 2.71828$. Therefore, $\sigma \leq e \cdot opt$. □

Corollary 1. *Algorithm A is an e-approximation algorithm.*

4 The MLSC Approximation Algorithm

The results of Section 3 assumed that we are able to compute an optimal densest k -sub-hyper-graph for all values of k . In this section we remove this assumption

¹ δ_k is the minimum time in our logarithmic scale that the solution defined by the path j_0, \dots, j_t completes k jobs, whereas L_k is the time - in the same scale- it takes OPT to accomplish this task.

by following the framework carried by Archer, Levin and Williamson [1] for the minimum latency problem.

For $k = 1, 2, \dots, n$, we find either an optimal densest k -sub-hyper-graph or a pair of values $k_l < k < k_h$ with optimal solutions V_{k_l}, V_{k_h} for the densest k_l -sub-hyper-graph and the densest k_h -sub-hyper-graph problems with costs W_l, W_h (respectively) such that the following property holds: let $k = \alpha k_l + (1 - \alpha)k_h$, then $a_k = \alpha W_l + (1 - \alpha)W_h \geq W_k$.

We note that there are values of k such that it is possible to compute a densest k -sub-hyper-graph in polynomial time. To make this claim precise we will show that given a parameter $\lambda > 0$ defining the cost of buying a tool, and the gain w_j obtained by purchasing the subset S_j (thus completing job j), it is possible to compute a profit maximizing set of tools. This auxiliary problem can be solved by a polynomial time algorithm for the PROVISIONING PROBLEM: Given n items to choose from where item j costs c_j , and given m sets of items S_1, S_2, \dots, S_m that are known to confer special benefit; if all the items of S_i are chosen then a benefit b_i is gained. The goal is to maximize the net benefit, i.e., total benefit gained minus total cost of items chosen. The provisioning problem is known to be solvable in polynomial time (See [2,14] and also [11] pages 125-127).

In fact using a single parametric min-cut procedure it is possible to compute the entire upper-envelope of the points in the graph of W_k versus k ([15] and see also [9] for more related results). This piecewise linear graph gives the desired a_k values for all values of k .

For values of k that for which we can compute the optimal densest k -sub-hyper-graph we let $a_k = W_k$, and for other values of k we let $a_k = \alpha a_l + (1 - \alpha)a_h \geq W_k$ and say that they corresponds to *phantom solutions* (the notion of phantom solutions is inspired by [1]). These phantom solutions are not solutions as we are not able to compute a densest k -sub-hyper-graph in polynomial time, however these phantom solutions provide values of a_k . Lemma 2 shows that there exists a shortest path in G which does not use *phantom vertices*, i.e. vertices that correspond to phantom solutions, for which we are not able to compute a densest k -sub-hyper-graph. As a consequence, Algorithm A can be applied to the subgraph of G induced by the vertices that correspond to non-phantom vertices, without loss of optimality.

Lemma 2. *There exists a shortest path in G that does not use phantom vertices.*

Proof. We prove the lemma by showing that even when the true parameter W_k of a phantom vertex k is replaced by $a_k \geq W_k$, thus reducing the lengths $(W - W_k)j$ of all arcs (k, j) , there exists a shortest path which does not use phantom vertices.

Consider a shortest path that visits $i \rightarrow k \rightarrow j$, where k is a phantom vertex with corresponding k_l, k_h as defined above. Set $\gamma = \frac{a_h - a_l}{k_h - k_l}$. By definition $a_k = (1 - \alpha)a_l + \gamma(k - k_l)$. By the definition of the arc lengths, the sub-path $i \rightarrow k \rightarrow j$ costs

$$\begin{aligned} (W - a_i)k + (W - a_k)j &= (W - a_i)k + (W - [(1 - \alpha)a_l + \gamma(k - k_l)])j \\ &= k(W - a_i - \gamma j) + (W - (1 - \alpha)a_l + \gamma k_l)j. \end{aligned}$$

This is a linear function of k and it is valid for $\max\{i, k_l\} \leq k \leq \min\{j, k_h\}$. Therefore, it attains a minimum at one of the endpoints $\max\{i, k_l\}$ or $\min\{j, k_h\}$. We can either remove loops to reduce the length of the path and thus obtain a contradiction, or we reduce the number of vertices along the path that correspond to phantom vertices. Using an inductive argument we establish the lemma. \square

Therefore, we can apply Corollary 1 to get the main result of this paper:

Theorem 3. *There is an e -approximation algorithm for the MLSC problem.*

5 Bad Example for Our Analysis

Goemans and Kleinberg [7] proved that using the randomized path in their analysis does not hurt the approximation ratio (with respect to the shortest path). It follows that there are networks where the ratio between the shortest path and the optimal solution is arbitrary close to the provable approximation ratio.

In our analysis, we have different arc lengths. Therefore, the question whether our analysis is tight is open. So far we were able to construct networks (by solving large linear programs) where the ratio between the shortest 1- n path to the optimal cost is approximately 2.62 for $n = 70$. This bound although monotone increasing does not approach e as n goes to infinity. Therefore, it might be possible to improve our analysis by using a different randomized path in the proof of the approximation ratio.

References

1. A. Archer, A. Levin and D. P. Williamson. "Faster approximation algorithm for the minimum latency problem", *Cornell OR&IE Technical report 1362*, 2003.
2. M. L. Balinski, "On a selection problem," *Management Science*, **17**, 230-231, 1970.
3. A. Blum, P. Chalasani, D. Coppersmith, W. Pulleyblank, P. Raghavan, M. Sudan. "The minimum latency problem", *Proceeding of the 26th ACM Symposium on the Theory of Computing*, 163-171, 1994.
4. P. Brucker, "Scheduling algorithms," Springer-Verlag, Berlin, 2004.
5. U. Feige, D. Peleg and G. Kortsarz, "The dense k -subgraph Problem," *Algorithmica*, **29** 410-421, 2001.
6. U. Feige, L. Lovász and P. Tetali, "Approximating min sum set cover," *Algorithmica*, **40**, 219 - 234, 2004.
7. M. X. Goemans and J. Kleinberg. "An improved approximation ratio for the minimum latency problem", *Mathematical Programming*, 82:111-124, 1998.
8. L. A. Hall, A. S. Schulz, D. B. Shmoys and J. Wein, "Scheduling to minimize average completion time: off-line and on-line approximation algorithms," *Mathematics of Operations Research*, **22**, 513-544, 1997.
9. D. S. Hochbaum, "Economically preferred facilities locations with networking effect," manuscript, 2004.
10. B. J. Lageweg, J. K. Lenstra, E. L. Lawler and A. H. G. Rinnooy Kan, "Computer-aided complexity classification of combinatorial problems," *Communications of the ACM*, **25**, 817-822, 1982.

11. E. L. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart & Winston, New-York 1976.
12. E. L. Lawler, "Sequencing jobs to minimize total weighted completion time subject to precedence constraints", *Annals of Discrete Mathematics*, **2**, 75-90, 1978.
13. J.K. Lenstra and A. H. G. Rinnooy Kan, "Complexity of scheduling under precedence constraints," *Operations Research*, **26**, 22-35, 1978.
14. J. Rhys, "Shared fixed cost and network flows," *Management Science*, **17**, 200-207, 1970.
15. D. D. Witzgall and R. E. Saunders, "Electronic mail and the locator's dilemma," In *Applications of Discrete Mathematics*, R.D. Ringeisen and F.S. Roberts eds. SIAM 65-84, 1988.

Workload-Optimal Histograms on Streams

S. Muthukrishnan¹, M. Strauss², and X. Zheng³

¹ Supported by NSF ITR 0220280 and NSF 0354600, Rutgers University
muthu@cs.rutgers.edu

² Supported by NSF DMS 0354600, University of Michigan
martinjs@umich.edu

³ Supported by NSF DMS 0354600, University of Michigan
xuanzh@eecs.umich.edu

1 Introduction

A histogram is a piecewise-constant approximation of an observed data distribution. A histogram is used as a small-space, approximate synopsis of the underlying data distribution, which is often too large to be stored precisely. Histograms have found many applications in database management systems, perhaps most commonly for query selectivity estimation in query optimizers [1], but have also found applications in approximate query answering [2], load balancing in parallel join execution [3], mining time-series data [4], partition-based temporal join execution, query profiling for user feedback, etc. Ioannidis has a nice overview of the history of histograms, their applications, and their use in commercial DBMSs [5]. Also, Poosala's thesis provides a systematic treatment of different types of histograms [3]. Formally:

Definition 1. A B -bucket histogram \mathbf{H} of length N is a partition of $[0, N)$ into intervals $[b_0, b_1) \cup [b_1, b_2) \cup \dots \cup [b_{B-1}, b_B)$, where $b_0 = 0$ and $b_B = N$, together with a collection of B heights h_j , for $0 \leq j < B$, one for each bucket. A point query $\mathbf{A}[i]$ to \mathbf{H} returns the estimate h_j where $b_j \leq i < b_{j+1}$.

In building a B -bucket histogram, we want to choose $B - 1$ boundaries b_j and B heights h_j , dependent on \mathbf{A} . A number of different criteria are known [3] for choosing b_j 's and h_j 's; a popular and effective one is the *V-Opt histogram* [6], where b_j 's and h_j 's are chosen to minimize the total square error, taken *uniformly over the set of all point queries*, or, equivalently, $\|\mathbf{A} - \mathbf{H}\|^2 = \sum_i (\mathbf{A}[i] - h_{j(i)})^2$. (Once we have chosen the boundaries, the best bucket height on an interval I is the average of \mathbf{A} over I .)

In [7], the authors presented an $O(N^2B)$ time algorithm for determining the optimal histogram \mathbf{H}_{opt} that minimizes the total square error. This algorithm has two drawbacks:

- it is expensive—quadratic in N . In order to overcome this drawback, focus has been on $(1 + \epsilon)$ -approximations, that is, algorithms to find a histogram \mathbf{H} such that $\|\mathbf{A} - \mathbf{H}\|^2 \leq (1 + \epsilon)\|\mathbf{A} - \mathbf{H}_{\text{opt}}\|^2$.
- it needs \mathbf{A} to be stored explicitly which is prohibitive in space for large distributions where histograms are used as synopses. In order to overcome

the second drawback, the focus has been on the *data stream model* of computation where (a) the algorithm reads the signal left to right in one pass as $\mathbf{A}[1], \mathbf{A}[2], \dots$, using space *polylogarithmic* in the input length N ; this is the so-called *time-series model* [8], or, (b) \mathbf{A} is specified as a series of *updates* and the algorithm has to track the changes to \mathbf{A} in space and time per update *polylogarithmic* in input N ; this is the so-called *cash register model* if only additions are allowed, or more generally, the *dynamic maintenance model* if both additions and deletions are allowed [8]. Notice that with the polylogarithmic space requirement, the input can only be represented *lossily* since accurate representation of the signal or the workload will need at least linear space in the worst case.

Besides the parameters B, N , and ϵ , the algorithms' costs depend on the numerical precision involved; we let M be a parameter such that $\log(M)$ is roughly the number of bits of precision used (see below for a formal definition). A series of $(1 + \epsilon)$ -approximation algorithms have been proposed that work in time in $O(N + \text{poly}(B, \log N, \log M, 1/\epsilon))$ in the time series model using $\text{poly}(B, \log N, \log M, 1/\epsilon)$ space enroute [9–11]. In the dynamic maintenance model, the authors in [12] present an algorithm that uses time per update, space, and post-processing time $\text{poly}(B, \log N, \log M, 1/\epsilon)$. This solves the approximate V -Opt histogram computation problem from a theoretical point of view, modulo getting the constants involved to be as small as possible.

It has, however, long been an issue that the V -Opt histogram as defined above is limited in its applications because it does not take into account the *workload* of queries for which the histogram is optimized. In particular, when some of the point queries are more frequent than the others, the histogram needs to be better at approximating answers to the frequent queries rather than the infrequent ones. In other words, the metric to minimize is not the sum of squared errors *uniformly* over all point queries, but that obtained by weighting the error on each point query by the workload of how frequently each point query is posed. Formally:

Definition 2. *Given an input signal $\mathbf{A}[0 \dots N - 1]$ and workload $w[0 \dots N - 1]$, $0 \leq w_i$, the workload-optimal B -bucket histogram \mathbf{H}_{opt} is the choice of b_j 's and h_j 's that minimize $\|\mathbf{A} - \mathbf{H}\|_w^2 = \sum_i w_i (\mathbf{A}[i] - h_{j(i)})^2$.*

The problem of finding \mathbf{H}_{opt} is interesting on stored or streamed signals as well as stored or streamed workloads.

The database community has proposed methods not merely to synthesize data distributions but also to take the workload into account. Query feedback from the execution engine of a DBMS was used in [13] to modify the synopsis. Histogram boundaries are refined adaptively in [14–16] based on a dynamically evolving workload that is continuously updated based on feedback from the query engine; they differ in how they approximate values within buckets, how they weight the workload etc. Still, these methods do not give any provable results on approximating \mathbf{H}_{opt} . There has been some work on *other* synopses that are workload-aware. For example, [17] proposed sampling methods that

adapt to recent workload. IBM’s LEO optimizer [18] uses workload information for a variety of synopses. In [19], a $O(N^2B/\log B)$ time algorithm is presented for determining the optimal choice of B Haar wavelet terms; this has recently been improved to $O(N^2)$ time [20]. The Haar basis is modified in [21] with the knowledge of the workload and algorithms for obtaining B -term synopses are designed for this new basis; while this algorithm works in linear time, it does not provide a near-optimal B -term Haar wavelet synopsis. For special workloads, [19] presented a near-linear algorithm for finding the optimal B -term Haar wavelet synopsis. All of these results for Haar and related bases [19–21] work only when *both* the signal and workload are available in a stored form without any loss of information, and not streamed with polylogarithmic space. However, when both the signal and workload are stored explicitly without loss of information, the dynamic programming from [7] immediately gives an $O(N^2B)$ time algorithm for finding the optimal \mathbf{H}_{opt} , so the challenge in [19,20] arises from working with the Haar wavelet basis and does not reflect on the difficulty in constructing \mathbf{H}_{opt} . In [9], there are many results of the same flavor as our result—indeed, the expanded version of [9] contains many generalizations not considered here—but the results of [9] do not appear to address directly our time- and space- bounded, workload-aware problem with the bounds we give. To summarize, the *significant open problem with finding \mathbf{H}_{opt} is when either the signal or the workload is streamed or both are streamed, with space polylogarithmic in N .*

In this paper, we address the problem of computing \mathbf{H}_{opt} on data streams. Our primary question is, do the powerful theoretical results known for uniform histogram construction on data streams [12,11,9] hold for the workload-aware case as well? Is there a difference in streaming the signal versus the streaming the workload in polylogarithmic space? What is the information-content of the workload and how does it affect the complexity of histogram construction?

Our contributions are as follows. Suppose the data items are integers, and the weights are positive integers between the minimum weight, w_{\min} , and the maximum weight, w_{\max} . Let $M = \max\{\|A\|^2, \frac{w_{\max}}{w_{\min}}\}$ be a bound on the range of data and weights.

Workload w is Stored Without Loss of Information. We present an $O(N + \text{poly}(B, \log N, \log M, 1/\epsilon))$ -time algorithm to compute a B -bucket histogram \mathbf{H} with $\|\mathbf{A} - \mathbf{H}\|_w^2 \leq (1 + \epsilon)\|\mathbf{A} - \mathbf{H}_{\text{opt}}\|_w^2$ where \mathbf{H}_{opt} is the workload-optimal B -bucket histogram, with respect to arbitrary w . This is the first near-linear¹ time algorithm for approximating \mathbf{H}_{opt} under non-uniform workloads. The above algorithm can be run in the time series model taking only $O(1)$ time per new item and using $\text{poly}(B, \log N, \log M, 1/\epsilon)$ space and post-processing time to construct the $(1+\epsilon)$ -approximate histogram. Under the more general dynamic maintenance model, the above algorithm can be modified using previously known techniques so that the time per update, total space used, and postprocessing time are all $\text{poly}(B, \log N, \log M, 1/\epsilon)$. This is the first known set of algorithms that use

¹ Note that, for moderate values of the parameters other than N , the run time is dominated by $O(N)$. In this paper, we use the term “near-linear” for this type of cost.

sublinear—polynomial in B , $1/\epsilon$ and polylogarithmic in N , M —space for dealing with data stream signals and yet yields $(1 + \epsilon)$ approximate \mathbf{H}_{opt} histograms for any w . It matches the previously known bounds for the special case when the workload is uniform [12].

Workload w is Compressed. We consider two cases.

- Given a workload vector w , let w' denote the vector of weights in w , rounded to a power of $(1 + \epsilon)$; it is easy to see that w' can be substituted for w giving up no more than a $(1 + \epsilon)$ -approximation factor in histogram estimation. This is a trivial lossy compression of w .

A simple argument shows that, in general, if the *rounded* weight vector w' of worst-case space $|w'| = \Theta(\log(\log(M)/\epsilon)N)$ bits is compressed to $|w'| - \omega(\log(M)/\epsilon)$ bits, then no algorithmic result of the type is possible (even if the entire signal is stored without loss). That is, no significant lossy compression of w is possible beyond discarding low-order bits.

- We focus on the case when w' is compressed without loss. We show that if w' is losslessly compressible to a structure $C(w')$ of size $|C(w')|$ by, *e.g.*, the Ziv-Lempel method [22], then the preprocessing time and space can be reduced from linear in N to linear in $|C(w')|$. We present a $O(|C(w')|)$ -spaced data structure to answer “symbol-range-count” queries which enable all our workload-optimal histogram construction algorithms above to be implemented in space and time $|C(w')|\text{poly}(B, \log N, \log M, 1/\epsilon)$. This is an advantage for highly compressible workload w 's, where $|C(w')| \ll |w'|$. We also present an alternative algorithm with different tradeoffs among resources.

This work integrates aspects of streaming algorithms (where input is compressed lossily) with algorithms that work with losslessly compressed input. This is a novel direction suggested in [19].

Section 2 has preliminaries. Section 3 has our results for the case when w is uncompressed, and is one of the main results here. The lower bound on space when the workload is streamed is in Section 4 and is fairly simple. In Section 5, we describe methods for managing a compressed workload w . In this extended abstract, formal proofs are omitted.

2 Preliminaries

Definition 3. *Inner Product with Weight:* For any two signals \mathbf{A} and \mathbf{B} of length N and any weight vector w of length N , define $\langle \mathbf{A}, \mathbf{B} \rangle_w = \sum_{i=1}^N \mathbf{A}_i \mathbf{B}_i w_i$ and $\|\mathbf{A}\|_w = \sqrt{\langle \mathbf{A}, \mathbf{A} \rangle_w}$ where w_i is a non-negative weight at index i . We continue to write $\langle \mathbf{A}, \mathbf{B} \rangle$ and $\|\mathbf{A}\|$ for the dot product and norm under uniform workload where all w_i 's are equal.

Definition 4. *Robust Representation [12,11].* Fix a signal \mathbf{A} . A representation \mathbf{H}_r is called a (B, ϵ) -robust approximation to \mathbf{A} if (i) $\|\mathbf{H}_r - \mathbf{A}\| \leq \epsilon \|\mathbf{H}_{\text{opt}} - \mathbf{A}\|$, or, (ii) for any representation \mathbf{H} on the boundaries of \mathbf{H}_r and any other $B - 1$ boundaries, with optimal parameters, we have $(1 - \epsilon)\|\mathbf{A} - \mathbf{H}_r\|^2 \leq \|\mathbf{A} - \mathbf{H}\|^2$.

Here we call this property *bucket robustness*, to distinguish it from *linear robustness*, defined later.

3 Uncompressed Workload

We first give an algorithm for time series data stream model that takes time $O(1)$ per item, space and post-processing time $\text{poly}(B, \log N, \log M, 1/\epsilon)$, where the signal and workload have length N and bound M , and a $(1 + \epsilon)$ -approximate B -bucket histogram is desired. Our algorithm's overall structure follows closely the algorithm in [11] for the uniform workload, so we first sketch that algorithm and selected parts of its analysis, then describe in detail the changes needed for non-uniform workloads.

3.1 A Previous Uniform-Workload Algorithm

The algorithm in [11] proceeds as follows.

1. (Selection of large wavelet terms.) Read in a length- N stream \mathbf{A} of time-series data and output a list L of the $B' \leq \text{poly}(B, \log N, \log M, 1/\epsilon)$ wavelet terms with largest coefficients.²
2. (Construction of bucket-robust representation.) Select the largest $B'' = \text{poly}(B, \log N, \log M, 1/\epsilon)$ terms from L , greedily, using a particular 2-part stopping rule (described in detail below). Call the result \mathbf{H}_r , a *bucket-robust* histogram of $O(B'')$ buckets.
3. (Construction of output.) Find a best B -bucket histogram \mathbf{H} to \mathbf{H}_r , and output \mathbf{H} as a $(1 + \epsilon)$ -approximate histogram to \mathbf{A} .

Note that the first step is performed on the stream, but the last two steps are full-space, polynomial-time post-processing algorithms on small input, that is, input of polylogarithmic size. In [11], the authors showed that the computational cost of each step meets the claim.

We now consider in more detail the relevant parts of the [11] algorithm. In Step 2, we need an additional parameter, $\epsilon_r = \Theta(\epsilon)$. We take terms from L , from biggest to smallest, $4B \log(N)$ at a time, and add them to \mathbf{H}_r , which is initially the zero histogram. Let \mathbf{H}'_r denote the *next* value of \mathbf{H}_r , *i.e.*, \mathbf{H}_r plus the next $4B \log(N)$ terms to be taken. We stop when either of the following conditions is met:

- (No Progress.) $(1 - \epsilon_r) \|\mathbf{A} - \mathbf{H}_r\|_2^2 \geq \|\mathbf{A} - \mathbf{H}'_r\|_2^2$.
- (Many Terms.) We have accumulated T terms, for some T which is at most $O(\epsilon_r^2 \log(1/\epsilon_r) B \log(N))$.

Using a case analysis, the output \mathbf{H} is shown to be correct whichever stopping rule is used. The conditions in bucket-robustness (Definition 4) correspond to the Many Terms and No Progress stopping rules, respectively.

² We do not need the definition of wavelets in this paper.

In Step 3, dynamic programming similar to [7] is used. In particular, the dynamic programming algorithm accesses \mathbf{H}_r only by making the following query. Given interval $[\ell, r)$, what is the best height a of a 1-bucket histogram $a\chi_{[\ell, r)}$ and what is the resulting error $\sum_{\ell \leq i < r} (\mathbf{H}_r[i] - a)^2$ on that interval? This query must be answered in time to meet the post-processing bound.

3.2 Our Algorithm for Non-uniform Workloads

Our algorithm generalizes the algorithm in [11] to non-uniform workloads. We first give a high-level description of our algorithm. To state the algorithm clearly, it is convenient to give some abstract definitions along the way. Analysis will be given later.

1. (Weight-class splitting.) We regard each weight w_i as rounded to w'_i , a power of $(1 + \epsilon)$. There is a small number $p = \log_{1+\epsilon}(M)$ of these classes. We split the incoming time series into p new time series, according to the associated rounded weight.
2. (Conversion to wavelets, selection of large terms, and construction of bucket-robust representation.) For each substream, we create a bucket-robust representation, as in [11]. For each substream, record which of the two stopping conditions was used.
3. (Recombination of substreams.) Conceptually, recombine the bucket-robust representations with stopping rule No Progress, getting \mathbf{H}'_r , and recombine the bucket-robust representations with stopping rule Many Terms, getting \mathbf{H}''_r . Conceptually, combine \mathbf{H}'_r and \mathbf{H}''_r , getting \mathbf{H}_r . The algorithm does nothing; it represents \mathbf{H}'_r and \mathbf{H}''_r as the appropriate collection of bucket-robust histograms.
4. (Construction of output.) Find a best B -bucket histogram \mathbf{H} to \mathbf{H}_r , and output \mathbf{H} as a near-best histogram to \mathbf{A} .

In what follows, we will give details of each step where it significantly differs from [11]. The technical aspects are manifold, but the crux is \mathbf{H}_r is *not* bucket-robust and as a result [11] fails. We now proceed formally.

Step 1. Rounding Weights. We consider signals of length N , with weights w_1, \dots, w_N , and such that $\|\mathbf{A}\|^2 \leq M$. We will assume that data items are integers and that weights are positive integers in the range $w_{\min} = 1$ to some $w_{\max} \leq M$. Define $p = \log_{1+\epsilon} M + 1$, and define p different rounded weights w^1, w^2, \dots, w^p , where $w^i = (1 + \epsilon)^{i-1}$. Round all the original weights w_1, \dots, w_N down to rounded weights w'_1, \dots, w'_N respectively, *i.e.*, $w'_i = w^j$ where $w^j \leq w_i < w^{j+1}$. We use $w' = (w'_1 \cdots w'_N)$ to represent the length- N rounded weight vector.

Lemma 1. *Fix a signal \mathbf{A} of dimension N . Then $\|\mathbf{A} - \mathbf{H}'_{\text{opt}}\|_w^2 \leq (1 + \epsilon)\|\mathbf{A} - \mathbf{H}_{\text{opt}}\|_w^2$, where \mathbf{H}_{opt} is the optimal B bucket representation to \mathbf{A} under weight w , and \mathbf{H}'_{opt} is the optimal B bucket representation to \mathbf{A} under weight w' .*

Proof. We have $\|\mathbf{A} - \mathbf{H}'_{\text{opt}}\|_w^2 = \sum_{i=1}^N [\mathbf{A}_i - \mathbf{H}'_{\text{opt}}(i)]^2 w_i \leq (1 + \epsilon) \sum_{i=1}^N [\mathbf{A}_i - \mathbf{H}'_{\text{opt}}(i)]^2 w'_i$ from the relationship between w_i and w'_i . Further, by optimality

of \mathbf{H}'_{opt} , $\|\mathbf{A} - \mathbf{H}'_{\text{opt}}\|_w^2 \leq (1 + \epsilon) \sum_{i=1}^N [\mathbf{A}_i - \mathbf{H}_{\text{opt}}(i)]^2 w'_i \leq (1 + \epsilon) \sum_{i=1}^N [\mathbf{A}_i - \mathbf{H}_{\text{opt}}(i)]^2 w_i$. Hence, $\|\mathbf{A} - \mathbf{H}'_{\text{opt}}\|_w^2 \leq (1 + \epsilon) \|\mathbf{A} - \mathbf{H}_{\text{opt}}\|_w^2$.

Step 3. Here is the technical crux. Unfortunately, the old definition of bucket-robustness fails in our framework of separate substreams based on weight classes. We give a brief illustration why. Suppose there are just two weight classes, that partition $[0, N)$ precisely into the even and odd indices. Suppose \mathbf{H}' and \mathbf{H}'' are two bucket-robust histograms, defined on the even and odd indices, respectively, and each has a small number of buckets. Suppose that all heights represented in \mathbf{H}' and \mathbf{H}'' are distinct. Suppose that \mathbf{H}' and \mathbf{H}'' have about equal shares of the error. Finally, suppose that the reason for bucket-robustness is *only* that they are not much improved by refinement by B more buckets; that is, suppose $\|\mathbf{A} - \mathbf{H}'\|^2$ and $\|\mathbf{A} - \mathbf{H}''\|^2$ are each approximately $\frac{1}{2} \|\mathbf{A} - \mathbf{H}_{\text{opt}}\|^2 > 0$, which can happen if $\mathbf{A} - \mathbf{H}_{\text{opt}}$ is noisy. We claim that all of these assumptions are consistent.³ Now, let \mathbf{H} be the combination of \mathbf{H}' and \mathbf{H}'' ; we will show that \mathbf{H} is not bucket-robust. Note that, because of the even/odd partition induced by the given weights, \mathbf{H} has N buckets, each of size 1. Then one of the bucket-robust conditions says that if we further refine \mathbf{H} and then *optimize the heights*, we do not get much improvement, multiplicatively. Clearly, by optimizing the heights, we can get error zero! The other condition says that $\|\mathbf{A} - \mathbf{H}\|^2 \approx \epsilon^2 \|\mathbf{A} - \mathbf{H}_{\text{opt}}\|^2$. But $\|\mathbf{A} - \mathbf{H}\|^2 = \|\mathbf{A} - \mathbf{H}'\|^2 + \|\mathbf{A} - \mathbf{H}''\|^2 \approx \|\mathbf{A} - \mathbf{H}_{\text{opt}}\|^2 \gg \epsilon^2 \|\mathbf{A} - \mathbf{H}_{\text{opt}}\|^2$. It follows that neither condition of bucket-robustness is satisfied.

Instead, we introduce the following new notion.

Definition 5. Fix a signal \mathbf{A} and rounded weight vector w' . Given parameters B and ϵ , let \mathbf{H}_{opt} denote an optimal B -bucket histogram for \mathbf{A} under w' . A representation \mathbf{H}_r is called a (B, ϵ) -linearly-robust (or just (B, ϵ) -robust henceforth) approximation to \mathbf{A} under weight w' , if, for any B -bucket histogram H_B and any scalars a and b , either $\|\mathbf{A} - \mathbf{H}_r\|^2 \leq \epsilon^2 \|\mathbf{A} - \mathbf{H}_{\text{opt}}\|^2$ or $(1 - \epsilon) \|\mathbf{A} - \mathbf{H}_r\|_w^2 \leq \|\mathbf{A} - (a\mathbf{H}_r + b\mathbf{H}_B)\|_w^2$.

The first condition is similar to the corresponding condition in bucket-robustness: the error $\|\mathbf{A} - \mathbf{H}_r\|^2$ is already tiny compared with $\|\mathbf{A} - \mathbf{H}_{\text{opt}}\|^2$. The second condition is implied by bucket-robustness. Hence, linear-robustness is a weaker notion than bucket-robustness.

Returning to our overall algorithm, recall that \mathbf{H}'_r and \mathbf{H}''_r are the recombinations of bucket-robust histograms for substreams where the stopping rule is No Progress or Many Terms, respectively.

Lemma 2. Each of \mathbf{H}'_r and \mathbf{H}''_r is linearly robust. Further, the combination \mathbf{H}_r of \mathbf{H}'_r and \mathbf{H}''_r is linearly robust.

Step 4. This involves two aspects, first a proof that the output \mathbf{H} is correct, that is, a best B -bucket histogram \mathbf{H} to \mathbf{H}_r is a $(1 + \epsilon)$ -approximation of the optimal histogram to \mathbf{A} , and second, how to find \mathbf{H} .

³ In any case, we cannot provably rule out the existence of such \mathbf{H}' and \mathbf{H}'' .

For the uniform workload, the fact that \mathbf{H}_r is bucket-robust was shown in [11] to imply that the output \mathbf{H} is correct. This was done by showing that bucket-robustness implies linear-robustness and that linear-robustness implies correctness, though the concept of linear-robustness was not isolated as important in [11]. To extend the proof in [12,11] from the uniform workload to workload w' , we observe that the proof under the uniform workload in [11] uses only the triangle inequality and an approximate Pythagorean theorem, which hold also under weight w' . Therefore we conclude that \mathbf{H} is a correct (approximate) output. For the reader familiar with [11], we note that the approximate weighted Pythagorean Theorem says that if $\langle \mathbf{A} - \mathbf{C}, \mathbf{B} - \mathbf{C} \rangle_{w'} \leq \epsilon \|\mathbf{A} - \mathbf{C}\|_{w'} \|\mathbf{B} - \mathbf{C}\|_{w'}$ (a near-right angle at C), then $\|\mathbf{A} - \mathbf{B}\|_{w'}^2 = (1 \pm \epsilon) \left(\|\mathbf{A} - \mathbf{C}\|_{w'}^2 + \|\mathbf{B} - \mathbf{C}\|_{w'}^2 \right)$.

Second, we perform dynamic programming. As remarked earlier, it suffices to find the best 1-bucket histogram representation to \mathbf{H}_r on a given range to perform the dynamic programming. In order to do this, we need to be able to answer the following type of query, which we call a *symbol-range-count* query. Given a weight class label, i , and a position, j , how many times does the weight class i occur in $[0, j)$? We use the following:

Lemma 3. *Let histogram \mathbf{H}_r equal the conceptual combination of p histograms \mathbf{H}_r^i of length N and at most B' buckets with respect to a partition of $[0, N)$ corresponding to p weight classes in the rounded weight vector w' . In time (and space) $O(N)$, one can build from w' (independent of \mathbf{H}_r) a data structure that, on query $[j', j)$, gives the best one-bucket approximation to \mathbf{H}_r on $[j', j)$ and the associated error, in query time $O(pB' \log(N))$.*

We can now summarize:

Theorem 1. *There is an algorithm that, given parameters B, N, M, ϵ and weight vector w of length N and bound M , preprocesses w in time and space $O(N)$, reads data \mathbf{A} with $\|\mathbf{A}\|_w^2 \leq M$ in time series, then outputs a B -bucket histogram \mathbf{H} with $\|\mathbf{A} - \mathbf{H}\|_w^2 \leq (1 + \epsilon) \|\mathbf{A} - \mathbf{H}_{\text{opt}}\|_w^2$, where \mathbf{H}_{opt} is the best possible B -bucket histogram representation to \mathbf{A} under weight w . The algorithm uses space $O(\text{poly}(B, \log N, \log M, 1/\epsilon))$ in addition to the space associated with w independent of the input. The algorithm uses time $O(N)$ to read the stream of data and post-processing time $O(\text{poly}(B, \log N, \log M, 1/\epsilon))$ to build \mathbf{H} .*

This meets the complexity for the uniform workload case, and settles the problem modulo improving the polynomials. Using prior work on the general *dynamic maintenance* model for uniform workload case [12] we get the following corollary (which also meets the best-known complexity for the uniform workload case [12]):

Corollary 1. *For parameters N, M, B, ϵ , there is a randomized data structure for an array \mathbf{A} that preprocesses a workload w in time and space $O(N)$, requires additional space $O(\text{poly}(B, \log N, \log M, 1/\epsilon))$ and supports the following operations in time $O(\text{poly}(B, \log N, \log M, 1/\epsilon))$: update (Add v to $\mathbf{A}[i]$, where v may be positive or negative); build (Build a $(1 + \epsilon)$ -near optimal histogram with respect to the then-current dataset \mathbf{A} , under workload w).*

4 Lower Bounds

It is easy to see that a histogram algorithm that first reads the data and then is given a workload must store all the data, since the choice of workload and histogram approximation criterion can force the algorithm to recover any data item exactly. This immediately gives

Theorem 2. *Suppose data and workload values are interleaved arbitrarily. For any $B \geq 3$, any algorithm that outputs approximate B -bucket histogram uses space $\Omega(N \log(M))$ bits (enough to store all the data).*

While Theorem 2 is the strongest possible statement about interleaving data and workload values, it says nothing about compressing the workload. Above we showed that, to get a $(1 + \epsilon)$ -factor approximation, one can round weights to a power of $(1 + \epsilon)$ (i.e., discard low-order bits). We now show that, in a sense, this is the only kind of lossy compression that is possible.

Lemma 4. *Suppose an algorithm reads and processes a workload of length N and bound M into an object s of size $|s|$, then discards everything about the workload except s , then reads time series data. If, for any workload, any data, and any sufficiently small $\epsilon > 0$, the algorithm produces, with probability $\gg 1/2$, a $(1 + \epsilon)$ -approximation to the best B -bucket histogram, then the algorithm can be used as a subroutine to store any value from a vector of positive integer entries bounded by $M/4$, of length $\geq N - O(\log(M)/\epsilon)$, up to the factor $(1 + O(\epsilon))$.*

In particular, the lemma above implies:

Theorem 3. *For any $B \geq 3$ and any sufficiently small $\epsilon > 0$, if an algorithm represents in space $|s|$ a workload w of length N and bound M and finds $(1 + \epsilon)$ -near-best B -bucket histograms with respect to w , then $|s|$ is at least the space needed to store $(N - O(\log(M)/\epsilon))$ counters of $\Omega(\log(M)/\epsilon)$ states.*

5 Compressed Weights

In the previous section, we showed that lossy compression of the workload beyond rounding is not possible, even information-theoretically. In this section, we consider efficient algorithms for manipulating losslessly compressed workloads of rounded weights. We consider principally the famous Lempel-Ziv compression methods. Our goal in this section is to build a symbol-range-count structure R to match the given compression scheme, where, we recall, a symbol-range-count query is the pair (i, j) , for which the answer is the number of occurrences of i in $[0, j)$. That is, if the compressed text $C(w')$ has size $|C(w')|$, then, ideally, we want to build R with preprocessing time $O(|C(w')|)$, we want $|R| \leq O(|C(w')|)$, and we want symbol-range-count queries to be as quick as possible—plausible guarantees are $O(\text{poly}(B, \log N, \log M, 1/\epsilon))$ or some function of the compressed string. Thus, the challenge is to be *opportunistic* and design data structures bounded in size by $|C(w')|$. We also discuss building R' of size $|R'| < o(|C(w')|)$ such that R' and $C(w')$ together constitute a symbol-range-query structure, R .

This has the advantage that the total size $|R|$ is $(1 + o(1))|C(w')|$ rather than $O(1)|C(w')|$; this is useful if, say, $|C(w')| = |w'|/100$, so that the constant factor in $O(1)$ is significant. Opportunistic data structures are known for indexing a string for full-text substring queries [23], but no previous results are known for our problem of supporting the symbol-range-count query. The results of this section may of interest separately in database and string processing.

Formally, we are given a string $S[1, \dots, N]$, with each $S[i]$ in alphabet set of size p (the rounded weight classes in our histogram problem). We compress S using Lempel-Ziv algorithm, denoted LZ78, which works as follows. Say $S[1, \dots, i]$ has been compressed; a dictionary D of tuples (d_k, l_k) would have been constructed thus far with each $d_k = S[l_k, \dots, l_k + |d_k| - 1]$. The algorithm iteratively proceeds by finding the longest prefix $S[i + 1, \dots, j]$ that equals some d_k , compressing $S[i + 1, \dots, j + 1]$ as $(l_k, |d_k|, S[j + 1])$, adding $(S[i + 1, \dots, j + 1], i + 1)$ to D and continuing. Each such step is called a “parse” and the number of parses is directly related to the size of the compressed representation $C(S)$ of S upto constant factors. Hereafter, we will let $|C(S)|$ be the number of such parses, as is standard in the string compression area, without being specific about how to code each $(l_k, |d_k|, S[j + 1])$ in smallest number of bits.

There are many variants of this basic method, depending on whether windowing is used, whether $S[i + 1, \dots, j]$ and $S[l_k, \dots, l_k + |d_k| - 1]$ may overlap or not, how the parses are encoded using bits, etc. We will focus on the basic version above and our results will hold for these other variants as well. A significantly different variant is the LZ77 [24] algorithm in which we add all substrings of $S[i + 1, \dots, j]$ to D . This leads to larger D and hence, fewer parses and smaller $C(S)$. Our algorithm in this section will work with the LZ77 compression method as well (also for run-length encoding methods), but we omit the details in this extended abstract.

Lemma 5. *A string S given in its LZ78 compressed form $C(S)$ can be preprocessed in time and space $O(|C(S)|)$ such that a symbol-range-count query (i, j, α) can be answered in time $O(|C[i, j]| \log \log N)$, where $|C[i, j]|$ is the number of LZ78 parses overlapping $[i, j]$.*

We also present a different method that we call *decimated statistics*.

Lemma 6. *Suppose a rounded workload w' of length N , multiplicative increment $(1 + \epsilon)$, and bound M , is compressed to $C(w')$ by LZ78. One can construct from $C(w')$, in time $O(p|C(w')|)$, a structure R' , of size $o(|C(w')|)$, such that R' together with $C(w')$ constitute a symbol-range-count data structure of size $(1 + o(1))|C(w')|$, with query time $\text{poly}(\log N, \log M, 1/\epsilon)$.*

The decimated statistics approach is incomparable with Lemma 5, as the next theorem records. Combining the lemmas with our histogram algorithms we have, analogous to Theorem 1 (a similar result holds analogous to Corollary 1 as well):

Theorem 4. *There is an algorithm that, given parameters B, N, M, ϵ and LZ78-compressed text $C(w')$ of rounded weight vector w' with $p \leq O(\log(M)/\epsilon)$*

classes, preprocesses $C(w')$ in time and symbol-range-count space $O(|C(w')|)$, reads data \mathbf{A} with $\|\mathbf{A}\|_w^2 \leq M$ in time series, then outputs a B -bucket histogram \mathbf{H} with $\|\mathbf{A} - \mathbf{H}\|_w^2 \leq (1 + O(\epsilon))\|\mathbf{A} - \mathbf{H}_{\text{opt}}\|_w^2$, where \mathbf{H}_{opt} is the best possible B -bucket histogram representation to \mathbf{A} under weight w . The algorithm uses space $O(\text{poly}(B, \log N, \log M, 1/\epsilon))$ in addition to the space associated with w' independent of the input. The algorithm uses time $O(N)$ to read the stream of data and post-processing time $|C(w')|\text{poly}(B, \log N, \log M, 1/\epsilon)$ to build \mathbf{H} .

An alternative algorithm is as above except, for any parameter P (e.g., a P that grows slightly faster than $p \approx \log(M)/\epsilon$), the preprocessing time degrades to $O(P|C(w')|)$, the symbol-range-count space improves to $(1 + p/P)|C(w')|$, and the post-processing time changes to $\text{poly}(B, P, \log N, \log M, 1/\epsilon)$.

6 Concluding Remarks

We have shown, for the first time, given a data set of length N and bound M , how to build a $(1+\epsilon)$ -near optimal B -bucket histogram that is provably nearly optimal with respect to a non-uniform workload, where the algorithm runs in nearly linear time $O(N + \text{poly}(B, \log N, \log M, 1/\epsilon))$ and space $\text{poly}(B, \log N, \log M, 1/\epsilon)$ beyond what is needed to store the workload. This algorithm generalizes to the dynamic update streaming model. For both time-series and dynamic maintenance update models, our time and space costs are comparable to those for the uniform workload. We have also shown that lossy compression of the workload is not possible beyond rounding to within the factor $(1 + \epsilon)$. Finally, we show how to improve the space cost to essentially the space used to compress losslessly the rounded workload by the Lempel-Ziv algorithm.

Another way to solve the problem under the time-series model is based on the lockstep model, where we are given w_i *only* when $\mathbf{A}[i]$ arrives and we can not archive the entire workload w_i . This is what we call the *lockstep* model. In this model, we are able to give a different algorithm that can be run in time $O(N \log(U) + \text{poly}(B, \log U, \log M, 1/\epsilon))$ using space $\text{poly}(B, \log U, \log M, 1/\epsilon)$ to construct the $(1+\epsilon)$ -approximate histogram, where $U = \sum_i w_i$. This is somewhat weaker than our main result here, but the lockstep streaming model is weaker. So, this result may be of independent interest. We do not know whether near-linear time can be achieved in the lockstep model; this is a natural open question.

Full proofs, results in the lockstep model as well as other open problems are in the larger version of this paper [25].

References

1. Ioannidis, Y., Christodoulakis, S.: Optimal histograms for limiting worst-case error propagation in the size of join results. *ACM Trans. Database Syst.* **18** (1993) 709–748
2. Acharya, S., Gibbons, P., Poosala, V., Ramaswamy, S.: The aqua approximate query answering system. In: *SIGMOD Conference*. (1999) 574–576

3. Poosala, V.: Histogram-based estimation techniques in database systems. PhD thesis, Univ of Wisconsin (1997)
4. Keogh, E., Chakrabarti, K., Mehrotra, S., Pazzani, M.: Locally adaptive dimensionality reduction for indexing large time series databases. In: Proc. SIGMOD. (2001)
5. Ioannidis, Y.: The history of histograms (abridged). In: Proc. VLDB. (2003)
6. Ioannidis, Y., Poosala, V.: Balancing histogram optimality and practicality for query result size estimation. In: Proc. SIGMOD. (1995) 233–244
7. Jagadish, H.V., Koudas, N., Muthukrishnan, S., Poosala, V., Sevcik, K., Suel, T.: Optimal histograms with quality guarantees. In: Proc. VLDB. (1998) 275–286
8. Muthukrishnan, S.: Data stream algorithms and applications. <http://www.cs.rutgers.edu/~muthu/stream-1-1.ps> (2003)
9. Guha, S., Koudas, N., Shim, K.: Data-streams and histograms. In: Proc. ACM STOC. (2001) 471–475
10. Guha, S., Koudas, N.: Approximating a data stream for querying and estimation: Algorithms and performance evaluation. In: Proc. ICDE. (2002)
11. Guha, S., Indyk, P., Muthukrishnan, S., Strauss, M.: Histogramming data streams with fast per-item processing. In: Proc 29th ICALP. (2002) 681–692
12. Gilbert, A., Guha, S., Indyk, P., Kotidis, Y., Muthukrishnan, S., Strauss, M.: Fast, small-space algorithms for approximate histogram maintenance. In: Proc. ACM STOC. (2002) 389–398
13. Chen, C., Roussopoulos, N.: Adaptive selectivity estimation using query feedback. In: Proc. ACM SIGMOD. (1994)
14. Konig, A., Weikum, G.: Combining histograms and parametric curve fitting for feedback driven query result size estimation. In: Proc. VLDB. (1999)
15. Aboulnaga, A., Chaudhuri, S.: Self-tuning histograms: Building histograms without looking at data. In: Proc. ACM SIGMOD. (1999)
16. Qiao, L., Agrawal, D., Abbadi, A.E.: Rhist: adaptive summarization over continuous data streams. In: Proc. CIKM. (2002) 469–476
17. Ganti, V., Lee, M., Ramakrishnan, R.: Icicles—self-tuning samples for approximate query answering. In: Proc. VLDB. (2000)
18. Stillger, M., Lohman, G., Markl, V., Kandil, M.: Leo - db2's learning optimizer. In: Proc. VLDB. (2001) 19–28
19. Muthukrishnan, S.: Nonuniform sparse approximation theory with Haar wavelets. Technical report, DIMACS (2004)
20. Guha, S.: A note on wavelet optimization. <http://www.cis.upenn.edu/~sudipto/notes/wavelet.pdf.gz> (2004)
21. Matias, Y., Urieli, D.: Optimal workload-based wavelet synopses,. Technical report, TAU (2004)
22. Ziv, J., Lempel, A.: Compression of individual sequences via variable-rate coding. IEEE Transactions on Information Theory **24** (1978) 530–536
23. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: Proc. IEEE FOCS. (2000) 390–398
24. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on Information Theory **23** (1977) 337–343
25. Muthukrishnan, S., Strauss, M., Zheng, X.: Workload-optimal histograms on streams. Technical report, DIMACS (2005)

Finding Frequent Patterns in a String in Sublinear Time

Petra Berenbrink¹, Funda Ergun², and Tom Friedetzky³

¹ School of Computing Science, Simon Fraser University,
Burnaby, B.C., V5A 1S6, Canada
<http://www.cs.sfu.ca/~petra/>

² School of Computing Science, Simon Fraser University,
Burnaby, B.C., V5A 1S6, Canada
<http://www.cs.sfu.ca/~funda/>

³ Department of Computer Science, Durham University,
Durham, DH1 3LE, U.K.
<http://www.dur.ac.uk/tom.friedetzky/>

Abstract. We consider the problem of testing whether (a large part of) a given string X of length n over some finite alphabet is covered by multiple occurrences of some (unspecified) pattern Y of arbitrary length in the combinatorial property testing model. Our algorithms randomly query a sublinear number of positions of X , and run in sublinear time in n . We first focus on finding patterns of a given length, and then discuss finding patterns of unspecified length.

1 Introduction

The problem of finding frequent occurrences of patterns in a string comes up in many areas such as telecommunications, e-commerce, and databases, where the applications generate long data streams to be analyzed. An example from data mining is efficient handling of iceberg queries, that is, identifying those objects in a data stream which occur with frequency over a threshold. In property testing of strings, testing whether a string consists of back-to-back repetitions of the same patterns is called *periodicity testing*. Usually, the efforts to efficiently identify such trends in data are hampered by the large size of the data, which can be too large to fit into main memory and to be efficiently analyzable, even by a linear time algorithm.

In this work, we are interested in detecting frequent repetitions of a pattern (of any size) in a string of length n in time sublinear in n . In contrast to previous work, the pattern boundaries are unrestricted, which, while more realistic, complicates matters. We explore this problem in the *combinatorial property testing* model ([10,3]), and first obtain an algorithm which distinguishes between strings which are mostly covered with occurrences of one pattern of given size k and those that do not contain a large number of repeated patterns in $o(k)$ time. We then generalize our result to detecting repetitions of patterns of *unspecified* length, in $o(n)$ time. In both cases, if a frequent pattern exists, the algorithm can implicitly return a (likely approximate) copy of the pattern.

The fact that patterns can occur anywhere in the string means that there can be a linear number of possible patterns of linear size in a given string,

and comparing them to one another can easily lead to inefficient algorithms. To handle this, we represent each pattern as a short “sketch” or a “signature”. However, there can still be a linear number of signatures sharing one location. To deal with this, we use a sparse representation of our data which is based on the few locations sampled. We show that this small amount of information is sufficient to make conclusions about repeated trends in the input stream.

Our Results. We first present an algorithm which, given a string X and a length k , tests in $O(\sqrt{k}\text{polylog}k)$ time if there exists a pattern Y of length k that covers X (notice that Y is not given); allowing the occurrences of the pattern to overlap. We say that Y *approximately covers* an α -fraction of X if there exists a set of substrings $\mathcal{Z} = \{Z_1, \dots, Z_j\}$ of X (each of length k) where $h(Z_i, Y) \leq \epsilon k$ for all $Z_i \in \mathcal{Z}$ and at least αn locations of X are covered by some $Z_i \in \mathcal{Z}$. Here $h(Z_i, Y)$ is the *Hamming distance* between Y and Z_i . If a pattern of length k exists that covers all of X , with probability $1 - o(1)$ our algorithm outputs PASS. If there is no pattern Y of length k that covers an α -fraction of X , $\alpha, \epsilon \in (0, 1)$, it outputs FAIL with probability $1 - o(1)$.

Next, we give an algorithm which, given X and k , tests in time $O(\sqrt{k}\text{polylog}k)$ whether there is a string of a length $\ell \in [\delta k, \dots, k]$ covering an α -fraction of X ; it outputs FAIL if there is no pattern Y of length k that approximately covers an $(1 - \beta)\alpha$ -fraction of X , with probability $3/4$, with certain restrictions on α, β and ϵ . We use this variant to test if there is a pattern of *any* length that covers an α -fraction of X , or if no pattern exists that approximately covers a $(1 - \beta)\alpha$ -fraction of X .

Related Work. Combinatorial property testing was first defined by [10,3]. For an overview of results see [9] and the references therein. Two recent related results testing whether a string is close to periodic are [2,7]. They assume that the size of the period is fixed so that position in which the period (corresponding to “patterns”) appear is fixed, too. This means that it is more or less clear which positions should be sampled. In our case, since a pattern can be anywhere, we need to make sure that we have samples in all possible places that a pattern can be ¹. Another related result tests in sublinear time whether two strings have large edit distance ([1]).

There have also been sublinear space streaming results. Iceberg queries for identifying objects that appear in more than a fraction of a string are explored in [6]. Periodicity testing is investigated using sketches in [5] where the running time is considered in terms of memory accesses. These techniques differ from ours that they are not bound by sublinear time, but are expected to return more accurate answers.

2 Preliminaries

Given string X of length n , let $X[i]$ refer to the i th character of X . $r = [i : j]$ denotes $\{i, i + 1, \dots, j\}$, where i and j are called respectively the *left and right*

¹ To achieve this, we use an extra stage of sampling where one of the stages makes sure that two copies of the same pattern will be correctly aligned.

endpoints of r . $X[i : j]$ denotes the substring of X starting at location i and ending at j . $[n]$ is short for $[1 : n] = \{1, \dots, n\}$.

Given a location i in X , we say that a length k substring (or pattern) Y “covers”, “contains”, or “appears around” i if and only if $Y = X[j : j + k - 1]$ such that $i \in [j : j + k - 1]$. Here Y only refers to the contents of the substring, and thus, can be repeated elsewhere in X . We call each such repetition an *occurrence*. $h(X, Y)$ denotes the Hamming distance between X and Y .

3 Finding Frequent Patterns of Given Length k

We first consider finding patterns of length *exactly* k .

3.1 Length Exactly k

We now formally define the problem of testing for frequent patterns. Formally, given a string X of length n and $1 \leq k \leq \alpha n$, we would like to have an algorithm with the following behavior.

- If there is a pattern Y of length k which covers all locations of X , then the algorithm returns PASS with probability $1 - o(1)$.
- If there is no pattern Y of length k such that a set of substrings $\mathcal{Z} = \{Z_1, \dots, Z_j\}$ of X (of length k) exist where $h(Z_i, Y) \leq \epsilon k$ for all $Z_i \in \mathcal{Z}$ and at least αn locations of X are covered by some $Z_i \in \mathcal{Z}$, then the algorithm returns FAIL with probability $1 - o(1)$.

Note that the occurrences of the pattern can overlap. For a visual intuition on the FAIL condition, consider a scheme to mark an X with respect to a pattern Y of length k . For $j = 1, \dots, n - k + 1$, if $h(Y, X[j : j + k - 1]) \leq \epsilon k$ then mark locations $X[j], \dots, X[j + k - 1]$. In the end, if some $X[i]$ remains unmarked, then there exists no substring Z of X of length k that covers location i such that $h(Z, Y) \leq \epsilon k$. The FAIL condition holds if and only if the marking of X with respect to Y results in at least $(1 - \alpha)n$ unmarked locations for any Y of length k .

Consider $X = \text{abcabcaabcaabca}$. $Y = \text{abca}$ covers X fully, where the a in location 4 is covered by two overlapping copies of Y , thus the algorithm should return PASS. Now let $X' = \text{abcdbaddacdedbcbe}$. Substrings abcd , dbad , dacd , dbc b , cover all but two characters of X' , and each has Hamming distance 1 to $Y = \text{dbcd}$; thus the algorithm can return either PASS or FAIL. Later, we will show how our results translate into the non-overlapping case, where our definition of distance will be equivalent to the usual one.

The approach of randomly choosing a few locations in X and checking whether there is a pattern which covers all of these locations is not straightforward to implement in sublinear time, for two reasons. First, even if two random locations i and j lie in two occurrences of the same pattern, they are likely to be in different positions within the two occurrences, with k^2 possible location pairs. This hurdle is not present in periodicity testing where the pattern boundaries are fixed. Second, the fact that locations p_1 and p_2 , as well as p_1 and p_3 occur

within the same pattern does not imply that p_2 and p_3 do. Two patterns can share $\Theta(k)$ many patterns of length k ; thus, finding one shared by all of the sample points in $o(n)$ time is nontrivial.

A Three-Stage Sampling Approach. We now present our approach which tackles the above problems in time $o(k)$. To do that, we will use sampling and keep small summaries of our samples in “signatures”. Let $\ell_p = \Theta(\text{polylog}n)$, $\ell_s = \Theta(\sqrt{k} \log \log k)$, $\ell_t = \Theta(\text{polylog}n)$, with large enough constants hidden in the Θ . Our sampling has three stages, where Stages 1 and 2 obtain *primary* and *secondary locations* and Stage 3 the actual samples.

Stage 1: Construct set $P = \{p_0, p_1, \dots, p_{\ell_p}\}$, of *primary locations*, where each p_i is chosen independently and uniformly at random (i.u.r.) from $[n]$.

Stage 2: For each $p_i \in P$, construct set S_i of *secondary locations*, said to be *owned* by p_i , of the form $S_i = \{s_{i,0}, s_{i,1}, \dots, s_{i,\ell_s}\}$,² where each $s_{i,j} \in S_i$ is chosen i.u.r. from $[p_i - k : p_i + k]$.³

Stage 3: Construct a *sorted* list of locations $T = t_1 t_2 \dots t_{\ell_t}$ where the t_i are picked i.u.r. from $[-2k : 2k]$ and are in ascending order. Now consider any secondary location $s_{i,j} \in S_i$. Obtain samples $T_{i,j} = s_{i,j} + t_1, s_{i,j} + t_2, \dots, s_{i,j} + t_{\ell_t}$; these will be owned by $s_{i,j}$. The elements of $T_{i,j}$ are uniformly distributed in $[s_{i,j} - 2k : s_{i,j} + 2k]$; furthermore, the locations of the samples relative to any secondary location s that owns them is identical across all s (Fig. 1).

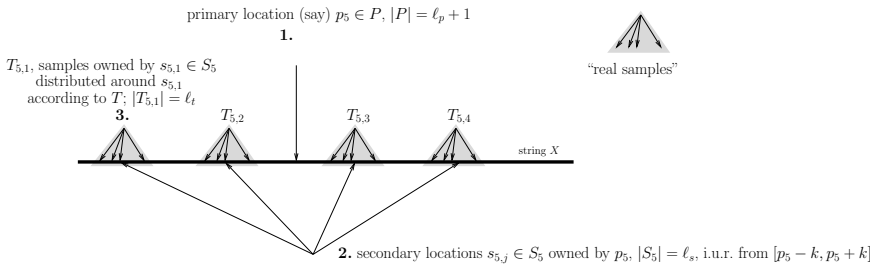


Fig. 1. Example for primary location p_5 ; steps (1,2,3) indicate order of selection

Templates and Signatures. We will represent each substring of length k of X with a short signature. To do this, decompose T into sublists which we call *templates*, each of which contains the offsets to obtain samples to form a signature.

Definition 1. A list τ is said to be a *template* (of T) if for some $-2k < i \leq k+1$, τ , τ is the maximal sublist of T whose elements are in the range $[i : i + k - 1]$.

The following lemma shows templates are large enough.

Lemma 1. Let $\ell_t = 24\bar{c} \log k$ for some large enough constant c . With probability at least $1 - o(1/k)$ every template consists of at least $\bar{c} \log k$ characters.

² We will drop subscripts later when they are obvious.

³ If $p_i < k$ ($p_i > n - k$) then the area $[1 : p_i + k]$ ($[p_i - k : n]$) is sampled.

Proof. Consider the probability that there exists a $j \in [-2k : 2k - j + 1]$ such that there are fewer than $\bar{c} \log k$ elements in T whose values are in $[j : j + k - 1]$. To do this, partition the interval $[-2k : 2k]$ into 12 subintervals of length $k/3$. Any interval $[j : j + k - 1]$ will fully contain such subinterval. The expected number of elements of T in a subinterval is $2\bar{c} \log k$, which is a lower bound on the expected length of a signature. Using Chernoff bounds (see e.g. [4]) the probability that a particular subinterval will have fewer than $\bar{c} \log k$ samples is at most $e^{-\bar{c} \log k} = o(1/k)$. Thus, the probability that at least one subinterval will have fewer than that many elements is also at most $o(1/k)$.

The next observation holds by symmetry, showing that the sample points are uniformly distributed over a template.

Observation 1. *For an interval $r = [i : i + k - 1]$ for $-2k \leq i \leq k + 1$, given that the template representing r contains p locations, the set consisting of these p locations is uniformly distributed in the subsets of size p of $\{i, \dots, i + k - 1\}$.*

Using the elements of a template as offsets with respect to a secondary location to obtain actual samples, we obtain a signature:

Definition 2. *Let $s = s_{l,m}$ be a secondary location and $\tau = t_i, t_{i+1}, \dots, t_j$ be a template representing some interval $[u : u + k - 1]$ for $-2k \leq u < k + 1$. The signature corresponding to τ with respect to $s_{l,m}$ is $sig_\tau(l, m) = X[s + t_i], X[s + t_{i+1}], \dots, X[s + t_j]$, representing the interval $[s + u : s + u + k - 1]$.*

Let $\mathcal{T} = \tau_1, \tau_2, \dots$ denote the list of all templates of T . Below we show that there are not too many distinct templates. This imposes an $O(\sqrt{k} \cdot \text{polylog} k)$ bound on the total number of signatures generated. The proof is omitted.

Lemma 2. *$|\mathcal{T}| \leq 2\ell_t$. Furthermore, the total number of signatures generated from X from the locations and samples obtained as above is at most $2\ell_t \ell_s (\ell_p + 1)$.*

Since there are many more intervals of length k than templates, we now build a succinct representation of their correspondance.

Definition 3. *Let τ be a template. Let $Q = \{i \mid -2k \leq i \leq k \text{ and interval } [i : i + k - 1] \text{ induces } \tau\}$. The range of τ , $r(\tau)$, is $[a : b]$, with a and b as the left and right endpoints of Q .*

The notion of the range of a template extends naturally to the range of a signature. Let $s_{i,j}$ be any secondary location and $[a : b]$ be the range of some template τ . Then the range of $sg = sig_\tau(i, j)$, denoted $r(sg)$, is $[a + s_{i,j} : b + s_{i,j}]$. We observe below how to compute the range of a template (the range of a signature is computed similarly). Let $t_0 = -2k - 1$ and $t_{i+1} = 2k + 1$.

Observation 2. *Let $\tau = t_i, t_{i+1}, \dots, t_j$ be a template. Then, $r(\tau) = [\max\{t_{i-1} + 1, t_j - k + 1\} : \min\{t_i, t_{j+1} - k\}]$.*

The Basic Sampling Algorithm. Our algorithm consists of two phases. In the initialization phase we construct data structure D with signatures related to the first primary location, p_0 . In the next phase we compare signatures of other primary locations, to those already considered. If we identify a pattern which occurs around all our primary locations, we return PASS. In what follows, let c be a sufficiently large constant.

Initialization Phase:

Obtain sets P, S of primary, secondary locations, T of offsets, and \mathcal{T} of templates
 If there exists a template $\tau \in \mathcal{T}$ with less than $c \log k$ sample points return FAIL
 Set $D = \phi; G = \phi;$
 For each secondary location $s_{0,i}$ for $1 \leq i \leq \ell_s$ and each template $\tau \in \mathcal{T}$
 $sg = sig_\tau(0, i)$
 let $r = r(\tau) \cap [p_0 - s_{0,i} - k + 1 : p_0 - s_{0,i}]$
 $R = \{r\}$
 if sg does not exist in D , insert $\langle sg, R \rangle$ in D ;
 otherwise let R' be the range of the entry found in D for sg
 change the range of the entry for sg in D to $R' \cup R$
 $G = G \cup R$

The operation taking intersection of the ranges ensures that substrings which do not intersect with p_0 are not considered⁴.

Iterative Phase:

For $m = 1$ to ℓ_p do
 $D' = \phi; G' = \phi;$
 Fill out D' with signatures around p_m as D was filled above for p_0
 For each signature sg in D' with range R
 If sg exists in D with range R' , $G' = G' \cup R$
 $G = G \cap G'$

Output: If $D \neq \phi$ return PASS, otherwise return FAIL.

Data Structures. Data structures D and D' store modified signatures and their ranges. A modified signature is obtained (from, say, $sg = sig_\tau(i, j)$) tagging sg with a prefix, namely the smallest index in τ . This ensures two matching (modified) signatures will come from the same template. Each node contains a signature and its current range set R , representing the (candidate) frequent substrings which have this signature. Both D, D' and R can be implemented by using any standard data structure that supports linear time construction and logarithmic time search and updates, as well as constant time *prev* and *next* operations. G and G' store ranges in a similar way. Inserting a range into R can take linear time due to the deletions of small ranges during merging. The deletion of a range can be charged to its insertion, maintaining logarithmic amortized insertion and deletion times. The union and intersection operations all are performed in logarithmic time per range.

Analysis of the Algorithm.

Theorem 3. *Let X be a string of length n and parameter k be such that $1 \leq k \leq \alpha n$. Let $\ell_p = c \cdot \log k$, $\ell_s = c' \sqrt{k \log k}$ and let $\ell_t = 24\bar{c} \log k$ with sufficiently large constants c, c', \bar{c} .*

(a) *If there is a pattern Y of length k that covers 100% of X , then the algorithm returns PASS with probability at least $1 - o(1)$.*

⁴ When we take a union of ranges, ranges which touch or overlap are merged.

(b) If there is no pattern Y of length k such that at least αn characters of X can be covered by substrings Z_1, \dots, Z_w (of length $|Y|$) where $h(Z_i, Y) \leq \epsilon k$ then the algorithm returns FAIL with probability at least $1 - o(1)$.

The algorithm runs in $O(\sqrt{k} \text{polylog} k)$ time and space.

Proof. We start with the proof of Part a. The runtime analysis is submitted in this short version.

(a): If the PASS condition is satisfied, we can get an outcome of FAIL if one the two following cases happens.

(i) For some p_i , we do not have a pair of “well aligned” secondary samples s, s' belonging to p_0 and p_i respectively. By assumption, we have a copy of a string Y covering p_0 and one covering p_i . To detect that these two copies are identical, we need to get identical signatures from them, for which we need to have secondary locations s, s' with identical relative locations w.r.t. the first and the second copy of Y respectively. By the birthday paradox (see [8], Page 45), the probability that we will not have such a “well aligned” pair of secondary locations for one particular p_i is at most $e^{-\ell_s(\ell_s-1)/2k} = e^{-(c'^2 k \log k - \sqrt{c'k \log k})/2k} \leq e^{-(c'^2 k \log k)/4k} \leq k^{-(c')^2/4}$ for $c' \geq 2\sqrt{c+1}$. Using the union bound, the probability that this situation might arise for some p_j is $1/k$.

(ii) We get a FAIL answer due to a signature which is smaller than the threshold. By Lemma 1 this can only happen with a probability of $o(1)$. Thus, the probability of an incorrect FAIL answer is at most $o(1)$.

(b): If the FAIL condition is satisfied, a PASS can be returned as as a result of two events, analyzed below.

(i) Choice of primary locations: Call two substrings of size k Z_1 and Z_2 *similar* if $h(Z_1, Z_2) \leq \epsilon k$. With the FAIL condition, a small number of the primary locations p_1, \dots, p_{ℓ_p} may be covered by substrings which are similar to one particular substring around p_0 . p_0 is covered by at most k different substrings of length k ; WLOG consider $Y = X[p_0 - k + 1 : p_0]$. Due to the FAIL assumption, marking X w.r.t. Y will leave at least $(1 - \alpha)n$ positions unmarked. The probability that a fixed primary location will fall on a marked position for a fixed string Y then is at most $1 - \alpha$.

(ii) Unlucky choice of templates: The signatures for two substrings Y and Y' can be identical even if Y and Y' differ in more that ϵk locations. This is a problem only if the signatures are at least $\bar{c} \log k$ characters long, since otherwise the algorithm automatically returns FAIL. (Note that for a match of signatures to be found, the two signatures must be generated from the same template, which guarantees that the two substrings are being compared at corresponding locations.)

Note that, due to how the signature of Y has been picked, the second statement of Lemma 1 and the bound on the size of a signature, the samples in the signature of Y correspond to a uniformly chosen subset of $\bar{c} \log k$ samples from Y . Assume that the signature for Y' has been obtained from the same template as that of Y (the opposite of this only helps us). For the two signatures to match, none of the samples in the signatures must be from locations where

Y and Y' differ. Since Y and Y' are not similar, the probability of this is at most $\epsilon^{\bar{c} \log k} \leq 1/k^3$. For any pair of primary locations p_0 and p_i we compare at most $\ell_s^2 \cdot 2\ell_t$ signatures with each other (see Lemma 2). The probability to find identical signatures for a pair of primary locations p_0 and p_i is at most $(\ell_s^2 \cdot 2\ell_t) \cdot \frac{1}{k^3} \leq \frac{1}{k}$. Since, given p_0 , there are at most k possible choices for Y , the probability of a false negative/false positive is at most $k \cdot ((1 - \alpha) + \frac{1}{k})^{\ell_p} \leq o(1)$.

We now present a lemma relating the result with overlapping patterns to non-overlapping patterns. The proof is omitted.

Lemma 3. *If αn characters of a string X of length n are covered by overlapping patterns of length k , then at least $\alpha n/2$ characters are covered by non-overlapping patterns.*

3.2 Length Approximately k

In this section we show how to test if any pattern of length in the range $[\delta k : k]$ for constant $\delta < 1$ occurs over a large fraction of a given string X . We first develop a high level algorithm similar to that in Section 3.1. Later, we will use this algorithm to find out if there is *any* pattern (of any size) which occurs frequently in X .

We define our modified algorithm in terms of its differences from the algorithm in Section 3.1. First, a template is now defined as the maximal sublist of T whose elements represent a range $[i : i + \delta k - 1]$ (see Definition 1). Consequently, a signature now spans an area of size δk .

The second change is in our data structures. In our previous algorithm, to identify a pattern as frequent, we confirmed that it occurred around all our primary locations. Here, we will check that a pattern occurs around a large number of primary locations. To count the occurrences of patterns around primary locations, we replace D with DR , described below. In the new algorithm, at the end of the iterative section, rather than taking an intersection of the ranges (along with signatures) found for the new p_m with the existing candidates ranges in D , we now simply add the new ranges found to DR . (Which keeps track of how many times a range has been added). At the end, if there is a particular pattern that occurs around many of the primary locations, it will be witnessed by DR that the signature and range representing the pattern have a large count (one for each occurrence around a primary location).

The algorithm outputs PASS if there is a signature and corresponding range (thus, a pattern) found around at least $\alpha \delta \ell_p$ primary locations, for some constants $\alpha, \delta < 1$, according to the count obtained from DR .

Data Structure for the Modified Algorithm. We use a data structure DR to store ranges in terms of their endpoints. DR is, like D , a standard data structure. Each node contains three fields: a *value* for an endpoint of a range, a *count* tracking the times that an endpoint has been encountered, and a one bit field containing the values *left* or *right* to qualify an endpoint. Here one can insert a range in logarithmic time, output how many times each (sub)range has been inserted in linear time for all of the ranges. For instance, if $[2 : 8]$ and $[6 : 14]$ have been inserted, DR has value 1 for $[2 : 5]$ and $[9 : 14]$, and 2 for the intersection, $[6 : 8]$.

To insert a range $[a, b]$, we first look for a in DR . If it is not found, we insert $(a, left, 1)$ into DR . If a exists and the endpoint bit shows *left*, we increment the count field for that entry; if the endpoint bit shows *right* we decrement the count. We treat b similarly: if the value is not found, we insert $(b, right, 1)$. If an entry exists and the endpoint is *right*, we increment the count; if the endpoint is *left*, we decrement the count. If at any point the count at a node reaches zero, we delete the node.

To obtain a count of the ranges, we use a range counter (initially set to 0), starting from the smallest value in DR and following the *next* pointers. For every node we see with endpoint *left* we increment the range counter by the count in that node; for every node with endpoint *right* we decrement by the count in that node. The value of the counter between two nodes in DR represents how many times the range delimited by the values in those two nodes has been inserted.

Analysis of the Algorithm. In this section we will prove that the algorithm works correctly. First we show that whenever there is a pattern of length k covering an α -fraction of the string, then there is a pattern of length δk covering an $\alpha\delta$ -fraction. The proof is omitted.

Lemma 4. *Let $\alpha \in (0, 1)$. Let X be a string of length n . Let $\delta \in (0, 1)$.*

- (a) *Whenever there exists a pattern of length ℓ with $\delta k \leq \ell \leq k$ that covers at least an α -fraction of C , then there also exists a pattern of length δk that covers at least an $(\alpha\delta)$ -fraction of the string.*
- (b) *Whenever there exists a pattern Y of length ℓ with $\delta k \leq \ell \leq k$ such that at least an α -fraction of X can be approximately covered by substrings Z_1, \dots, Z_j (of length $|Y|$) where $h(Z_i, Y) \leq \epsilon k$, then there also exist a pattern Y' of length δk and substrings Z'_1, \dots, Z'_j , such that least an $(\alpha\delta)$ -fraction of the string can be covered by the Z'_1, \dots, Z'_j with $h(Z'_i, Y') \leq \epsilon' \delta k$ where $\epsilon' = \epsilon/\delta$.*

Theorem 4. *Let $k \geq 100$. Let $\alpha \in [\frac{4}{5}, 1)$, $\beta \in (0, 1)$ with $\alpha(1 - \beta) \leq \frac{2}{3}$. Let $\delta = \frac{40}{41}$. Let X be a string of length n . Let $\ell_p = c \cdot \log k$, $\ell_s = c' \sqrt{k \log k}$ and let $\ell_t = \bar{c} \log k$ for large enough constants c, c', \bar{c} .*

- (a) *If there is a pattern Y of length ℓ with $\delta k \leq \ell \leq k$ that covers an α -fraction of X , then the algorithm returns PASS with probability at least $3/4$.*
- (b) *If there is no pattern Y of length ℓ with $\delta k \leq \ell \leq k$ such that at least an $\alpha(1 - \beta)$ -fraction of X can be covered by substrings Z_1, \dots, Z_j (of length $|Y|$) where $h(Z_i, Y) \leq \epsilon k$ then the algorithm returns FAIL with probability at least $3/4$.*

The algorithm runs in $O(\sqrt{k} \text{polylog } k)$ time and space.

Proof. (a) We can get a FAIL answer if one of the following three cases happen.

(i) For some p_i , we do not have a pair of “well aligned” secondary samples s and s' belonging to p_0 and p_i respectively. Using the birthday paradox we can show that the probability that there exists a primary location that we can not align to p_0 is $1 - o(1)$.

(ii) We get a FAIL due to a signature which is too small. Lemma 1 shows that this will only happen with a probability of $o(1)$.

(iii) We get a FAIL because not sufficiently many of our primary location positions fall into the pattern. Using Lemma 4, the probability that a fixed primary location does hit the occurrence of the pattern is at least $\alpha\delta$, thus p_0 will not be in the pattern with a probability of $1 - \alpha\delta$. From the remaining ℓ_p primary locations, expected $\alpha\delta\ell_p$ samples will fall into an occurrence of the pattern. Using Chernoff bounds from [4], we can show that the probability that fewer than $(1 - \gamma) \cdot \alpha\delta\ell_p$ of p_1, \dots, p_{ℓ_p} fall within an occurrence of the pattern is at most $1/k$. We can make γ a constant as close to zero as we wish by making the constant c (the coefficient of $\log k$ in ℓ_p) large enough.

Putting things together, the probability that the algorithm outputs FAIL in the PASS case is at most $o(1) + (1 - \alpha\delta) + 1/k = o(1) + (1 - (40/41)\alpha) + 1/k = o(1) + (1/41)\alpha + 1/k \leq 1/4$ for k a large enough constant.

(b) We now consider the probability of a PASS answer if the FAIL condition is satisfied. Notice that our algorithm now allows for finding patterns (of length between δk and k) that actually do not contain primary locations. As we choose secondary locations within a $\pm k$ radius around primary locations, a primary location may have a distance of up to $(1 - \delta)k$ from an endpoint of an occurrence of the pattern, and still be able to identify the pattern as such. We refer to these regions of size $(1 - \delta)k$ to the left and to the right of an occurrence as *extra regions*.

Consider the modification of the marking game from Section 3.1, that marks all locations that allow for identifying occurrences of the pattern Y , by marking both the occurrences of Y itself, as well as all the corresponding extra regions. It is easy to see that if there does *not* exist a pattern of length ℓ with $\delta k \leq \ell \leq k$ that covers an $\alpha(1 - \beta)$ -fraction of X , the modified marking scheme will mark at most $\alpha(1 - \beta)n + (\alpha(1 - \beta)n/\delta k) \cdot 2(1 - \delta)k = \alpha(1 - \beta)n \cdot (2/(40/41) - 1) = \alpha(1 - \beta)n \cdot (1 + 1/20)$ locations. The first term, $\alpha(1 - \beta)n$, is an upper bound on how much the actual pattern can cover, whereas the second term is an upper bound on the number of occurrences of the pattern, multiplied with the size of the extra regions (of which there are two for every occurrence). Let $\mu = \alpha(1 - \beta) (1 + \frac{1}{20})$, i.e., the coefficient of n in the above expression.

Fix an occurrence Y of length δk that is identifiable by p_0 , i.e., p_0 is contained in either Y itself, or in one of the two extra regions around Y . Notice that there are $k + 2(1 - \delta)k = (3 - 2\delta)k$ many choices for Y . There are two cases that let the algorithm find a pattern between p_0 and some p_i

(i) Unlucky choice of primary locations: too many of the primary locations p_1, \dots, p_{ℓ_p} may be covered by substrings which are similar to Y . Hence, the probability that a primary location is close enough to an occurrence of the pattern Y is at most μ (as defined above).

(ii) Unlucky choice of templates: The signatures for two substrings Y and Y' can be identical even if Y and Y' differ in more than $\epsilon'k$ locations (see Lemma 4, part (b)). Similar to Theorem 3, we can show that in this case the probability to find identical signatures for a pair of primary locations p_0 and p_i for $i \in \{1, \dots, \ell_p\}$ is at most $1/k$.

Similar to the proof of Theorem 3 we can argue that the probability to find identical signatures for a pair of primary locations p_0 and p_i is at most $\mu + 1/k$ for a fixed pattern Y . Hence, the expected value for the counter of Y is $(\mu + 1/k) \cdot \ell_p$. Using Chernoff bounds [4], it is easy to show that the probability that the algorithm finds more than $(1 + \gamma')(\mu + 1/k)\ell_p$ copies of Y , is at most $1/k^3$. Again, we can obtain a (constant) γ' as close to zero as we wish by choosing a sufficiently large value of c .

For the PASS and FAIL case to be distinguishable, we need $(1 + \gamma')(\mu + 1/k) = (1 + \gamma')(\alpha(1 - \beta)(1 + 1/20) + 1/k) + \lambda \leq (1 - \gamma) \cdot \alpha\delta$ for some (constant) $\lambda > 0$. Choose c large enough such that $(1 + \gamma') \leq \frac{100}{99}$. Since $k \geq 100$, $(1 + \gamma')(\alpha(1 - \beta)(1 + 1/20) + 1/k) \leq 71/99$. Furthermore, we can choose $(1 - \gamma) \geq \frac{41}{42}$, and thus $(1 - \gamma)\alpha\delta \geq \frac{41 \cdot 4 \cdot 40}{42 \cdot 5 \cdot 41} = \frac{32}{42} = \frac{16}{21}$. Therefore, we indeed have a gap of $\lambda = \frac{16}{21} - \frac{71}{99} = \frac{31}{693}$.

Since there are at most $k + (1 - \delta)k = (2 - \delta)k$ possible choices for Y , the probability of a false negative/false positive is at most $(2 - \delta)k \cdot \frac{1}{k^3} = o(1)$.

Using several runs of the algorithm together with simple majority vote it is easy to strengthen the results such that the algorithm gives the right answers with a polynomial small probability. In the following we refer to this algorithm as the reliable version of the algorithm that finds variable length patterns. The runtime is still $O(\sqrt{k}\text{polylog}k)$.

Note that the algorithm can be easily modified to answer PASS if, say, x percent of the string is covered with a pattern, and FAIL if less than $x - \alpha$ percent of the string are covered with a pattern.

4 Finding Frequent Patterns of Unspecified Length

In this part we will use the reliable version of the algorithm that finds variable length patterns in order to search for all patterns that cover most parts of the string. The new algorithm works in $\log_{1/\delta} n$ rounds. In round i ($1 \leq i \leq \log_{1/\delta} n$), we search for patterns of length ℓ with $\delta^i n \leq \ell \leq \delta^{i+1} n$.

The algorithm works on an *output table* which has an entry for every i with $1 \leq i \leq \log_{1/\delta} n$. It writes PASS (FAIL) in position i of the array if the algorithm outputs PASS (FAIL) in round i . We can prove the following Theorem.

Theorem 5. *Use the same definitions as in Theorem 4 and run the modified algorithm for $\Theta(\log n)$ times per round. Furthermore, fix $\delta < 1$ and choose r such that $\delta^r \geq 100$.*

- (a) *For every $i \leq r$ such there exists a pattern Y of length ℓ with $\delta^{i+1} n \leq \ell \leq \delta^i n$ that covers an α -fraction of X , the algorithm writes a PASS into position i of the output array with a probability of $1 - n^{-1}$.*

- (b) For every $i \leq r$ such there exists no pattern Y of length ℓ with $\delta^i n \leq \ell \leq \delta^{i+1} n$ such that at least an $\alpha(1-\beta)$ -fraction of X can be covered by substrings Z_1, \dots, Z_j (of length $|Y|$) where $h(Z_i, Y) \leq \epsilon k$ the algorithm writes a FAIL into position i of the output array with a probability of $1 - n^{-1}$.

The algorithm runs in $O(\sqrt{k} \text{polylog} k)$ time and space.

Proof. The proof follows directly from Theorem 4. The array has $o(n)$ entries and for every i the algorithm answers PASS (FAIL) correctly with a probability of $1 - n^{-2}$.

5 Conclusions

It is also possible to define an algorithm for which a constant number of primary locations is sufficient, rather than $O(\log k)$ as in the previous sections. However, since “nothing is for free” there is a bigger gap in the pattern length between the PASS and FAIL cases. Notice that for our algorithms a constant number of primary locations is not enough since we essentially search for the k possible patterns of length k that contain the primary location p_0 . This means that, for a fixed pattern Y which includes p_0 , the probability that all primary locations are contained in the same pattern has to be at most $1/k$ for the FAIL case. Since the probability that a fixed primary location is contained in a fixed pattern (*not* a fixed occurrence of a pattern) is constant, we need $\log k$ many primary locations. This algorithm will be presented in the full version. Unfortunately, it is in general not possible to determine the *longest* pattern occurring in the string, whilst guaranteeing a probability for correctness of the answer, using our model. See the full version of this paper for a more detailed discussion

References

1. T. Batu, F. Ergun, J. Kilian, A. Magen, S. Raskhodnikova, R. Rubinfeld and R. Sami. *A sublinear algorithm for weakly approximating edit distance*. *STOC 2003*, 316–324.
2. F. Ergun, S. Muthukrishnan, and C. Sahinalp, *Sublinear methods for detecting periodic trends in data streams*. *Latin American Symposium on Theoretical Informatics (LATIN), 2004*.
3. O. Goldreich, S. Goldwasser and D. Ron. *Property testing and its connection to learning and approximation*, *Journal of the ACM* 45(4):653–750, 1998.
4. T. Hagerup and C. Rüb. *A Guided Tour of Chernoff Bounds*. *Information Processing Letters* 33 (1989), pp. 305–308.
5. P. Indyk, N. Koudas and S. Muthukrishnan, *Identifying Representative Trends in Massive Time Series Data Sets Using Sketches*. *Proc. VLDB 2000*. 363–372.
6. R. Karp, S. Shenker, and C. Papadimitriou, *A simple algorithm for finding frequent elements in streams and bags*. *ACM Trans. Database Syst.* 28: 51-55 (2003)
7. O. Lachish and I. Newman, *Periodicity Testing*. *Proc. RANDOM 2005*, to appear.
8. R. Motwani and P. Raghavan, *Randomized Algorithms*. Cambridge University Press (1995)
9. D. Ron, *Property Testing (A Tutorial)*. *Handbook of Randomization, 2000*.
10. R. Rubinfeld and M. Sudan, *Robust Characterization of Polynomials with Applications to Program Testing*, *SIAM Journal of Computing* 25(2):252–271, 1996.

Online Occlusion Culling*

Gereon Frahling and Jens Krokowski

Heinz Nixdorf Institute, Computer Science Department,
University of Paderborn, D-33102 Paderborn, Germany
{frahling, kroko}@upb.de

Abstract. Modern computer graphics systems are able to render sophisticated 3D scenes consisting of millions of polygons. For most camera positions only a small collection of these polygons is visible. We address the problem of occlusion culling, i.e., determine hidden primitives. Aila, Miettinen, and Nordlund suggested to implement a FIFO buffer on graphics cards which is able to delay the polygons before drawing them [2]. When one of the polygons within the buffer is occluded or masked by another polygon arriving later from the application, the rendering engine can drop the occluded one without rendering, saving important rendering time.

We introduce a theoretical online model to analyse these problems in theory using competitive analysis. For different cost measures we invent the first competitive algorithms for online occlusion culling. Our implementation shows that these algorithms outperform the FIFO strategy for real 3D scenes as well.

1 Introduction

To visualize complex 3D scenes at interactive frame rates one needs efficient algorithms to determine the visible parts. Approximation and culling techniques are used to reduce the number of primitives that have to be rendered. Occlusion culling attempts to identify the parts of the scene hidden by objects in front of them. To identify all occlusions during the rendering it is necessary to render the primitives (or polygons) in front-to-back order in respect to the camera position. On the other hand, spatial sorting and the traversing of the used data structures are very expensive if done at all, especially for dynamic scenes. Therefore, most applications balance between updating, traversing, and rendering costs and perform only coarse spatial sorting. The rendering pipeline processes these polygons in causal order, i.e., in the order they arrive. Polygons occluded by polygons arriving later are unnecessarily rendered into the frame buffer and expensive pixel shader commands are executed. This increases the total rendering time.

Aila et al. [2] introduced the concept of a Delay Stream, i.e., a small cache, capable of storing (tiles of) polygons before rendering. If one of these polygons is culled by a polygon in the stream after it, it can be removed from the cache

* Research is partially supported by DFG grant 872/8-2, by the DFG Research Training Group GK-693 of the Paderborn Institute for Scientific Computation (PaSCo).

without rendering. The authors implemented the cache as a First-In-First-Out (FIFO) buffer and showed by practical measurements that on real scenes such a buffer can reduce the number of rendered pixels by a factor of four.

This paper addresses the question if FIFO is the best cache management method. The rendering process consists of different stages. The most complex and costliest stages address the rendering of single pixels, such that the rendering time depends nearly linear on the number of pixels drawn. We address this by a theoretical model called the *size model*. Other stages perform actions for each polygon which do not depend on the size. To cover these cases we look at the *uniform model*. We analyse online algorithms for both models using competitive analysis. In the uniform cost model we propose the algorithm MARK, a variant of a paging algorithm given in [10]. In the size model we analyse the algorithm BALANCE, a variant of a weighted caching algorithm given in [7].

The paper is organized as follows: Section 2 introduces the occlusion culling models and online analysis methods. In sections 3 and 4 we show how to adapt the algorithms MARK and BALANCE to the occlusion culling scenario. Practical measurements in section 5 examine the behaviour of MARK, BALANCE, and other strategies on real test scenes. We conclude in section 6.

1.1 Related Work

Since the early nineties many occlusion culling algorithms have been presented, a recent survey is given in [8]. The methods compute the visible primitives either during preprocessing and store them in Potential Visible Sets (PVS) for regions of the camera position or determine them online. Online methods normally store the scene in a hierarchical data structure, e.g., octree[12], kd-tree [9], or bounding box hierarchy [18], to compute the point-based visibility. Usually, the sub-division is stopped if the leafs of the data structure contain less than a threshold of some hundred polygons. PVS methods usually overestimate the exact visible set to create larger regions for the camera. The system of [13] computes the PVS during the walkthrough in less than a second but the calculated sets are about two times larger than the exact visible sets. All these methods send a stream of primitives to the graphics card for rendering, which includes more or less many hidden primitives which cannot be determined by a causal visibility test. Therefore, online occlusion culling methods can be used to improve such rendering systems.

Aila et al. [2] noticed that buffering parts of the stream can be used to detect primitives that are hidden by other ones presented later. Their system consists of a FIFO buffer and the visibility of each primitive is checked two times: when it is inserted into the buffer and again when leaving the buffer.

From the theoretical point of view no results are known addressing the online occlusion culling problem using competitive analysis. Sleator and Tarjan [15] analysed the competitive performance of paging algorithms (closely related to online occlusion culling algorithms, see Section 2). They proved that LRU and FIFO are k -competitive for paging problems with a cache size of k . They also proved a matching deterministic lower bound. In [16] Young showed for several

deterministic caching strategies that they are *loosely* $\log k$ -competitive, which is a weaker model of competitiveness we don't discuss here.

A first algorithm for the weighted caching problem was given in [7]. It is k -competitive and will be discussed in detail in Section 4. Young discussed the greedy-dual algorithm, a generalization of many well known paging strategies which is as well k -competitive in the weighted caching model [16]. A general k -competitive algorithm LANDLORD for weighted caching was given in [6] and was analysed in more detail in [17]. Fiat et al. [10] showed that there are randomized online algorithms for paging which are $O(\log k)$ -competitive. The authors presented an algorithm MARK, which achieves a competitive ratio of $2H_k$, where H_k denotes the k -th harmonic number. MARK is not the best algorithm known so far, since in [11] and in [1] randomized algorithms for paging were introduced achieving a competitive ratio of H_k matching the lower bound [10].

1.2 Our Contribution

Observing similarities with online paging and caching problems we first prove lower bounds on the competitive ratio of online occlusion culling problems. We extend the algorithms BALANCE [7] and MARK [10] to online occlusion culling. Particularly, the extension of BALANCE is not trivial since one needs methods to handle the case of multi-occlusions (one polygon occluding several other polygons). We introduce new analysis methods for BALANCE and MARK and show that our variants achieve the same competitive ratio for online occlusion culling as for paging and weighted caching, resp. No competitive algorithms for online occlusion culling were known before. Our theoretical results are summarized in Table 1. In Section 5 we test BALANCE and MARK in real world scenarios. The results show that they belong to the best occlusion culling algorithms known so far, in many cases outperforming all other algorithms.

2 Occlusion Culling Models and Online Analysis

In this section we introduce two models for occlusion culling having different cost measures. In each model our algorithm must handle a stream of polygons. Each polygon p has a screen size $w(p)$ denoting the number of screen pixels showing the polygon. A polygon can occlude other polygons on the screen. In our theoretical model we assume that a polygon is either completely occluded or not occluded at all by another polygon. By using a z-buffer [5], our algorithm can identify polygons occluded by polygons seen earlier within the stream (not paying any costs).

An online occlusion culling algorithm maintains a cache (or buffer) in which it can store k polygons. Each polygon p seen in the data stream has to enter the cache at a cache position chosen by the algorithm. If a polygon q in the cache is occluded by p the algorithm can drop q and replace it by p in the cache without paying any rendering costs. Notice that it is possible that p occludes several polygons within the cache simultaneously. If there are any free cache positions, the algorithm can store p at a free cache position. In case the cache is

full and no polygon within the cache is occluded the algorithm has to choose one polygon within the cache to be drawn on screen, paying the rendering costs for this polygon. After the drawing p can enter the free cache position. The online algorithms differ in the strategies of choosing the polygons to be drawn when the cache is full. They must decide *online*, i.e., without knowing future polygons.

We will analyse two different cost measures since rendering applications in practice can be limited by different bottlenecks [14]: in "vertex (or polygon) limited" situations all rendered polygons create uniform costs. This is addressed by the *uniform cost model*, in which an online algorithm has to pay one cost unit for each polygon it renders on the screen. Please note, if in practice the overall performance is bounded by geometry processing, our method can only speed up the rendering about a constant factor, since the time of the online occlusion culling step is dominated by the z-buffer test which is already linear in the number of tested polygons.

Many applications are "fill limited", i.e., the rendering time is dominated by processing all pixels. This is modeled in the more sophisticated *size model*, in which each polygon p contributes with its size $w(p)$ to the rendering costs.

We will analyse both models using *competitive analysis* [15]. We compare an online algorithm ALG (which must decide without knowing future polygons) with an optimal offline algorithm OPT which knows the whole instance in advance and computes the best strategy for the instance.

Definition 1. Let c be a constant and ALG be a (randomized) online algorithm for occlusion culling. For an instance I let $C_{\text{ALG}}(I)$ denote the rendering costs of ALG on the instance and $C_{\text{OPT}}(I)$ the minimum rendering costs achievable by any algorithm on the instance. For a randomized algorithm $C_{\text{ALG}}(I)$ is a random variable dependent on the coin flips during the run of the algorithm. We call a deterministic online algorithm ALG c -competitive, if for each instance I : $C_{\text{ALG}}(I)/C_{\text{OPT}}(I) \leq c$. A randomized online algorithm ALG is c -competitive, if for each instance I : $\mathbf{E}[C_{\text{ALG}}(I)]/C_{\text{OPT}}(I) \leq c$.

The uniform and size models have many similarities to paging [15] respectively weighted caching [7]. In the weighted caching online problem an algorithm has to maintain a cache of k pages. It sees a stream of requests of pages. When a requested page p is not in the cache, this is called a cache fault and the page must be brought to the cache by paying costs of $w(p)$. If the requested page is within the cache, an algorithm does not need to do anything, and pays no costs. In the paging online problem all page weights are one.

When we don't allow multi-occlusions (one polygon occluding several other polygons simultaneously) and we assume that each occluding polygon has the same size as the occluded polygon, the occlusion problem in the uniform model is equivalent to paging and the occlusion problem in the size model is equivalent to weighted caching. Each repeated page in a paging instance corresponds to an occluded polygon in a occlusion culling instance. Using this coherence we can transfer lower bounds from paging [10] and weighted caching [15] to online occlusion culling:

Table 1. Bounds on the competitive ratio shown in this paper for the analysed occlusion culling problems (see Section 2 for model descriptions). Notice $\ln k < H_k \leq 1 + \ln k$.

Problem	Lower bound	Upper bound
Deterministic algorithm / Uniform model	k	k (BALANCE)
Randomized algorithm / Uniform model	H_k	$2 + 2H_k$ (MARK)
Deterministic algorithm / Size model	k	k (BALANCE)
Randomized algorithm / Size model	H_k	k (BALANCE)

Theorem 1. *Let H_k be the k -th harmonic number. No deterministic (randomized) online algorithm for occlusion culling in the uniform model can be better than k -competitive (H_k -competitive). No deterministic online algorithm for occlusion culling in the size model can be better than k -competitive.*

3 The Algorithm MARK

We alter the paging algorithm MARK and his analysis given by Fiat et al. [10] to handle the case of one polygon occluding several other polygons. MARK maintains for each cache position a marking bit.

Algorithm MARK

Unmark all marking bits.

for each polygon p within the stream **do**

If p is not occluded by a polygon already seen (test using z-buffer) **then**

 Drop all polygons within the cache occluded by p and unmark their positions (which are empty now)

If there is an empty cache position **then**

 Bring polygon p into that position and mark the position.

else

If there is no unmarked cache position **then** unmark all positions

 Choose one of the unmarked positions uniformly at random

 Render the polygon of this position

 Bring polygon p into this position and mark the position

end if

end if

end for each

Theorem 2. *The algorithm MARK is $2 + 2 \cdot H_k$ -competitive for the uniform online occlusion culling problem, where H_k denotes the k -th harmonic number.*

The proof will be provided in the full version of the paper.

Since this occlusion culling algorithm MARK is equivalent with the paging algorithm MARK for paging on "paginglike instances" (see Section 2), we can apply the lower bound of Achlioptas et al. [1] for the paging case.

Lemma 1. *Let H_k denote the k -th harmonic number. The occlusion culling algorithm MARK is not c -competitive for $c < 2 \cdot H_k - 1$.*

4 The Algorithm BALANCE

In this section we look at the more practical *size model*, in which each polygon contributes with its size on the screen to the drawing costs. From Section 2 we conclude that the weighted occlusion culling problem is at least as hard as weighted caching and no deterministic occlusion culling algorithm for the size model can have a better competitive ratio than k .

We first give evidence why all known algorithms for online occlusion culling are not competitive in the size model.

Lemma 2. *No (randomized) algorithm, which does not consider the sizes of the polygons can be c -competitive with a constant c .*

The Lemma shows that a competitive algorithm must prefer to render small polygons when the buffer is full. However, always preferring the smallest one within the buffer is also not a good strategy.

Lemma 3. *The algorithm LEASTVISIBLEFIRST which always renders the smallest polygon, is not c -competitive for any constant c .*

The proofs will be provided in the full version of the paper.

In [7] Chrobak, Karloff, Payne, and Vishwanathan showed that an algorithm called BALANCE is k -competitive for the weighted caching problem. We adapt BALANCE to the problem of weighted occlusion culling and show that it is still k -competitive. In contrast to Chrobak et al. we must deal with the difficult case of multi-occlusions.

The algorithm BALANCE maintains counters S_1, S_2, \dots, S_k for each of the k positions in the cache. Let w_i denote the size of the polygon currently within the i -th cache position.

Algorithm BALANCE

```

for each polygon  $p$  within the stream do
  If  $p$  is not occluded by a polygon already seen (test using z-buffer) then
    If  $p$  does not occlude any polygon within the cache then
      Find the index  $i \in \{1, \dots, k\}$  such that  $S_i = \min\{S_1, \dots, S_k\}$ 
      Render the polygon at the  $i$ -th cache position (if there is any)
      Store  $p$  at the  $i$ -th cache position
      Set  $S_i \leftarrow S_i + w(p)$ 
    end if
  If  $p$  occludes the polygons  $q_1, \dots, q_l$  at cache positions  $i_1, \dots, i_l$  then
    Replace the occluded polygon  $q_1$  by  $p$ 
    Set  $S_{i_1} \leftarrow S_{i_1} + w(p) - w(q_1)$ 
    For  $j = 2, \dots, l$  set  $S_{i_j} \leftarrow \max_{h \in \{1, \dots, l\}} \{S_h - w_h\}$ 
    Remove the occluded polygons  $q_2, \dots, q_l$  from their positions
  end if
end if
end for each

```

4.1 Analysis of Algorithm BALANCE

We show that BALANCE is k -competitive for the weighted occlusion culling problem. Consider an optimal offline algorithm OPT. Consider a worst adversarial input sequence of polygons.

Lemma 4. *If BALANCE is not k -competitive, there is an input sequence such that*

1. BALANCE produces more than k times the costs of OPT.
2. No polygon is occluded by polygons already seen in the sequence.
3. At the end of the sequence OPT and BALANCE have the same set of polygons within their caches.
4. No polygon in the sequence occludes another one already rendered by OPT.
5. No polygon in the sequence occludes exactly one within the cache of BALANCE.
6. No counter S_i decreases during the run of BALANCE.

Proof: If BALANCE is not k -competitive, there is an input sequence, such that BALANCE produces more than k times the costs of OPT. We will alter this input sequence into an input sequence having all the remaining properties, such that the costs of BALANCE do not decrease and the costs of OPT do not increase.

A polygon which is occluded by polygons already seen in the sequence will be detected by the z-buffer and therefore has no impact on BALANCE or OPT. If at the end of the sequence OPT and BALANCE do not have the same set of polygons within their cache we alter the sequence by appending polygons occluding polygons within the cache of OPT. This does not increase the costs of OPT. At some point of time BALANCE will have rendered all polygons not in the cache of OPT and the caches have the same content.

Consider a polygon p which occludes polygons q_1, \dots, q_l already rendered by OPT. If no q_i is within the cache of BALANCE, p could be replaced by a polygon which does not occlude any polygon seen so far. This would not alter the costs of OPT or BALANCE. If q_1, \dots, q_m are within the cache of BALANCE p could be replaced by a polygon \tilde{p} having size $w(p) - w(q_1)$, and presented at a point of time, such that it will follow q_1 within the cache of BALANCE. Furthermore, we replace the polygons q_2, \dots, q_m by polygons not occluded by p , but of a smaller size, such that they remain in the buffer of BALANCE exactly until p is presented. Notice that the behavior of BALANCE is exactly the same as before. The cost of Balance therefore can only increase (by the rendering of q_2, \dots, q_m). The costs of OPT can only decrease because instead of q_2, \dots, q_m OPT now has to render smaller polygons.

We can furthermore avoid the case of a polygon p which occludes exactly one polygon q within the cache of BALANCE. If q is not within the cache of OPT, case 4 applies. If q is within the cache of OPT, we can replace q by p and delete the second occurrence of p without altering the costs of BALANCE or OPT.

We will now alter the sequence such that no counters of BALANCE decrease. A counter can only decrease if more than one polygon is occluded by a new polygon

p . W.l.o.g. this happens at cache positions $1, \dots, l$, such that $i_j = j$. Only for the occluded polygons q_2, \dots, q_l the counter values decrease, since p is stored within the first cache position. Let $\tilde{S}_j = S_j - w(q_j)$ denote the counter value of cache position j before the element q_j came into it. Let $\tilde{S} = \max_{h \in \{1, \dots, l\}} \{S_h - w_h\}$ denote the corresponding counter value after BALANCE dropped q_2, \dots, q_l .

We alter the sequence to one without decreasing counters in the following way: We replace the polygons $q_j, j = 2, \dots, l$ by polygons \tilde{q}_j of size $w(\tilde{q}_j) = \tilde{S} - \tilde{S}_j$ and p by a polygon \tilde{p} of size $w(\tilde{p})$ occluding $q_1, \tilde{q}_2, \dots, \tilde{q}_l$. This way the counters S_2, \dots, S_l are exactly at value \tilde{S} when p is presented within the stream and do not decrease. Notice that all counters are at least of value $\min\{S_1, \dots, S_k\}$ until p is presented. This guarantees that no \tilde{q}_i is rendered before the occurrence of p . Therefore, BALANCE behaves the same way on the altered input sequence and the costs of BALANCE are the same as before the alteration (since q_2, \dots, q_l don't need to be rendered either way). The costs for OPT can only decrease since the sizes of the polygons are smaller after this instance change. \square

We will now concentrate on sequences fulfilling the requirements of Lemma 4. For these instances the proof follows the ideas of the proof from [7] for the paging case. We just have to deal with the fact that one page can occlude several pages in the cache. For instances fulfilling the requirements of Lemma 4 this can be analysed in the same way than single occlusions.

Theorem 3. *BALANCE is k -competitive for the weighted occlusion culling problem.*

The proof will be provided in the full version of the paper.


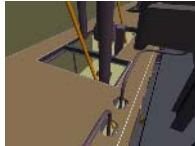
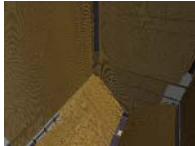
5 Experiments

We have implemented our proposed approach as a prototypical rendering system in C++ using OpenGL routines for the rendering. All experiments are made on a windows based system with a 1.6GHz Pentium M and a NVIDIA Quadro FX graphics card. For all benchmarks we choose a resolution of 1024×768 pixels.

We perform two common culling methods called view frustum culling and backface culling in front of the online occlusion culling step. Additionally, causal occluded polygons detected at the beginning of our online occlusion unit by the z-buffer test are dropped without counting. Therefore, remaining polygons after these standard tests will pass all stages of a common rendering pipeline, (temporally) change pixel values and cause rendering costs.

The z-buffer of the occlusion test was realized using an OpenGL feedback buffer to guarantee that the depth values used for the occlusion test are exactly the same as the depth values used later in the rendering process. Hence, for each rendered polygon we have to read out the z-buffer of the graphics card, which is a time consuming process. Therefore, we cannot state any realistic results for the improvement of the rendering time. Like Aila et al.[2], we assume that the savings in pixel processing should directly result in a increased frame rate.

Table 2. Statistical overview for different test scenes. Reduction of polygons and drawn hidden pixels for buffer size $k = 40, 200, 1000$. Best and close to the best (within 5%) results are highlighted in bold. *PowerPlant Corridor & Tubes* courtesy of the Walkthrough Group at the University of North Carolina at Chapel Hill (www.cs.unc.edu/walk/).

Scene									
# Polygons in view frustum	44563			380639			329168		
# Polygons after standard tests	14512			163053			53041		
buffer size k	40	200	1000	40	200	1000	40	200	1000
Reduction of polygons in per cent									
MARK	24.4%	33.4%	44.5%	25.4%	34.7%	46.1%	32.4%	38.8%	44.9%
BALANCE	15.8%	24.6%	38.5%	16.6%	25.8%	39.9%	21.7%	26.2%	35.5%
LRU	24.7%	34.1%	45.7%	25.3%	32.9%	45.3%	35.9%	40.5%	45.9%
NF	3.0%	19.5%	44.1%	3.4%	20.2%	30.3%	3.4%	9.4%	29.0%
MVF	4.1%	19.5%	42.2%	4.8%	20.7%	39.8%	2.3%	7.6%	28.9%
LVF	0.2%	2.5%	15.7%	0.3%	3.4%	16.4%	0.4%	2.1%	8.4%
Depth complexity	1.91			3.84			6.46		
reduction of drawn hidden pixels in per cent									
MARK	6.2%	13.7%	23.5%	8.8%	14.4%	24.6%	16.3%	21.9%	28.1%
BALANCE	7.9%	21.0%	34.9%	9.1%	23.9%	36.4%	22.7%	41.9%	60.2%
LRU	6.5%	14.7%	23.8%	7.4%	15.7%	25.3%	18.5%	21.5%	28.3%
NF	1.0%	7.8%	18.5%	1.3%	9.2%	20.1%	2.9%	6.0%	19.6%
MVF	0.7%	4.6%	12.2%	0.6%	5.3%	14.3%	2.2%	3.9%	13.1%
LVF	3.4%	15.8%	32.7%	4.1%	16.7%	33.3%	22.7%	38.9%	55.9%

5.1 Strategies

We investigated the empirical performance of the strategies MARK and BALANCE of Sections 3 and 4, respectively. Since our occlusion culling scenario is related to caching, we also consider the standard caching strategy *Least-Recently-Used* (LRU) as well as fairly natural strategies in this scenario, depending on the distance to the view point and the projected screen size:

Selection by Distance. If the polygons of a scene are processed strictly front-to-back, all completely occluded polygons will be determined by a causal occlusion test and the rendering costs will be minimized. In order to get a partially spatial sorting we have implemented the strategy *Nearest-First* (NF), which always renders the polygon nearest to the viewer among all polygons currently in the buffer.

Visible Pixels. The *Least-Visible-First* (LVF) strategy records the number of visible pixels of a polygon at the time it is inserted into the buffer. If a buffer overflow occurs, it selects the polygon with the smallest pixel counter. The opposite strategy *Most-Visible-First* (MVF) selects the polygon with the greatest pixel counter.

Depending on the cost model used for the investigation, there are arguments for both strategies: on the one hand, one expects that as the size of a polygon increases, the "probability" for its occlusion will decrease. Additionally, it is more likely that small polygons will be far away from the view point, since the

projected screen size of a polygon decreases approximately with the square of its distance to the view point [9]. Therefore, the selection strategy should prefer small polygons to be kept in the buffer, as MVF does. On the other hand, if the *size model* is observed, the rendering of a small polygon increase the total cost insignificantly. But, if it is kept in the buffer, it wastes valuable storage capacity that could be used for efficient buffering otherwise.

Note, the strategies First-In-First-Out (FIFO) and Least-Recently-Used (LRU) are identical in the online occlusion culling model because each polygon appears in the stream exactly once.

5.2 Results

Our test scenes and their characteristics are summarized in Table 2. We made experiments for each combination of strategy, scene and buffer size. The polygons are roughly spatial sorted by using an octree which is traversed in a front-to-back order. Since there is some redundancy in the results we do not give figures for all possible combinations. Instead we try to focus and explain the behavior on selected examples. We discuss the results for varying buffer sizes in detail for the *Town* scene, but the characteristics of the curves are similar for the other scenes. Therefore, we summarized the results for buffer sizes 40, 200 and 1000 in Table 2 and the results for the test scene *Town* are shown in detail in Figure 1 at the top row.

Uniform Cost Model. First we compare the strategies w.r.t. the uniform cost model. In the top-left diagram of Figure 1 we see the number of polygons recognized to be occluded for different buffer sizes and strategies. After all standard visibility tests 14512 polygons have to be rendered without our online occlusion culling unit, whereas just 48.7% of them are (partial) visible. The highest reduction is achieved by strategies MARK and LRU for all buffer sizes. With buffer size 1000 LRU recognizes 45.7% of the remaining polygons as occluded (MARK: 44.5%). The second best group of strategies is composed of NF and MVF followed by BALANCE with 38.5% – 44.1% identified occlusions. Finally, the performance of LVF is very poor, because many small polygons are occluded by large polygons later arriving the cache, but the strategy LVF already selected these polygons for rendering.

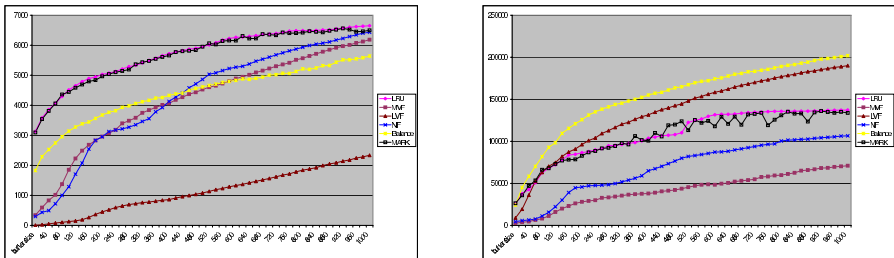


Fig. 1. Comparison of the strategies for the test scene *town*. On the left side the results for the uniform cost model are shown and the results for the size model are on the right.

Size Model. After the standard visibility tests 1,212,416 pixels are changed during the rendering, but only 637,225 different pixels are visible (the remaining pixels are filled with the background color without creating any costs). Therefore, the depth of complexity of the *Town* scene is 1.91, i.e., 91% of the pixels of the screen are rendered twice on average.

The top-right diagram of Figure 1 illustrates the number of "economized" pixels for different cache sizes, i.e., the number of pixels that were saved due to the fact that the corresponding hidden polygon was not rendered. In this cost model the strategy BALANCE achieved best reduction results. With cache size 1000 it excludes 34.9% of the hidden pixels from rendering. The second best strategy is LVF (which is the worst strategy in the polygon cost model) reducing the number of changed pixel values about 32.7%. The strategies MARK and LRU behave nearly the same and exclude 23.5% and 23.8% of the hidden pixels from rendering, resp., followed by NF and MVF. The good performance of BALANCE can be explained the following way: Strategies which do not prefer rendering small polygons must have higher drawing costs. However, holding the biggest polygons within the cache forever occupies valuable cache positions. BALANCE is in theory *and* practice a good compromise.

Note, our online occlusion culling system is not able to reach the optimal depth complexity of 1.0 because many polygons are only partial visible and we do not split these polygons.

To summarize briefly, considering the uniform cost model MARK and LRU achieve the best reduction results for all of our tests and BALANCE and LVF do so for the size model.

6 Conclusion and Future Work

Interesting extensions of these results would be randomized algorithms which achieve a better competitive ratio than k for the size model. Since these algorithms would directly lead to improved weighted caching algorithms this is a challenging problem to look at.

It would be also interesting to translate other caching algorithms to online occlusion culling. Results about lookahead could probably be transferred to occlusion culling. Results about loose competitiveness would imply good results in practice.

Another challenge could be not to look at the worst case behavior of algorithms. A first step could be to develop reasonable models to represent input distributions of scenes. Different online occlusion culling algorithms could be analysed according to these input distributions and lead to even better algorithms in practice (compare [3] for average case paging analysis).

Acknowledgements

We would like to thank Christian Sohler for many fruitful discussions about this topic.

References

1. D. Achlioptas, M. Chrobak, and J. Noga. Competitive analysis of randomized paging algorithms. *Proc. 4th Annual European Symposium on Algorithms (ESA)*, pp. 419–430, 1996.
2. T. Aila, V. Miettinen, and P. Nordlund. Delay streams for graphics hardware. In *ACM Transactions on Graphics*, 22(3), pages 792–800. ACM, ACM Press, 2003.
3. L. Becchetti. Modeling Locality: A Probabilistic Analysis of LRU and FWF. *Proc. 12th Annual European Symposium on Algorithms (ESA)*, pp. 98–109, 2004.
4. A. Borodin and R. El-Yaniv. Online computation and competitive analysis. *Cambridge University Press*, 1998.
5. E. Catmull. A Subdivision Algorithm for Computer Display of Curved Surfaces. *PhD thesis, University of Utah*, 1974.
6. P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. *USENIX Symposium on Internet Technologies and Systems*, 1997.
7. M. Chrobak, H. Karloff, T. H. Payne, and S. Vishwanathan. New results on server problems. *SIAM Journal on Discrete Mathematics*, 4(2), pp. 172–181, 1991.
8. D. Cohen-Or, Y. Chrysanthou, C. Silva, and F. Durand. A survey of visibility for walkthrough applications. *Transactions on Visualization and Computer Graphics*, 9(3):412–431, 2003.
9. S. R. Coorg and S. J. Teller. Real-time occlusion culling for models with large occluders. In *Symposium on Interactive 3D Graphics*, pages 83–90, 189, 1997.
10. A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young. On competitive paging algorithms. *Journal of Algorithms*, 12, pp.685–699, 1991.
11. L. A. McGeoch and D. D. Sleator. A strongly competitive randomized paging algorithm. *Algorithmica* 6, pp.816–825, 1991.
12. N. Greene, M. Kass, and G. Miller. Hierarchical z-buffer visibility. In *Proc. of ACM SIGGRAPH 93*, pages 231–238, 1993.
13. T. Leyvand, O. Sorkine, and D. Cohen-Or. Ray Space Factorization for From-Region Visibility. In *ACM Transactions on Graphics*, 22(3), pages 595–604. ACM, ACM Press, 2003.
14. D. Shreiner. Performance opengl: Platform independent techniques. In *ACM SIGGRAPH 2001 course notes*, 2001.
15. D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2), pp.202–208, 1985.
16. N. E. Young. The k-server dual and loose competitiveness for paging. *Algorithmica* 11(6), pp.525–541, 1994.
17. N. E. Young. Online file caching. *Proc. 9th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp.82–86, 1998.
18. H. Zhang, D. Manocha, T. Hudson, and K. Hoff. Visibility culling using hierarchical occlusion maps. In *Proc. of ACM SIGGRAPH 97*, pages 77–88, 1997.

Shortest Paths in Matrix Multiplication Time*

[Extended Abstract]

Piotr Sankowski

Institute of Informatics, Warsaw University,
Banacha 2, 02-097, Warsaw, Poland
sank@mimuw.edu.pl

Abstract. In this paper we present an $\tilde{O}(Wn^\omega)$ time algorithm solving single source shortest path problem in graphs with integer weights from the set $\{-W, \dots, 0, \dots, W\}$, where $\omega < 2.376$ is the matrix multiplication exponent. For dense graphs with small edge weights, this result improves upon the algorithm of Goldberg that works in $\tilde{O}(mn^{0.5} \log W)$ time, and the Bellman-Ford algorithm that works in $O(nm)$ time.

1 Introduction

The Single Source Shortest Paths (SSSP) problem is one of the most fundamental problems in combinatorial optimization. In this paper we consider this problem in the case when negative weights are allowed but no negative weight directed cycles. In this case the first algorithm for SSSP problem was proposed by Shimbel in 1955 [12]. Some years later, the so called, Bellman-Ford method was developed in the papers [7,1,8]. The Bellman-Ford algorithm is strongly polynomial, i.e., its time complexity does not depend on the weights in the graph. Thirty years later three scaling algorithms were developed [4–6]. The fastest of them is the algorithm of Goldberg. It works only in the case of integer edge weights from the set $\{-W, \dots, 0, \dots, +\infty\}$, and its time complexity depends on $\log W$. This algorithm works faster than the Bellman-Ford method under the similarity assumption, i.e., when $W = O(\text{poly}(n))$. The complexity results for the SSSP problem with negative edge weights are summarized in Table 1.

In this paper we show how the matrix multiplication can be used to obtain algorithm for the SSSP problem in the case of integer edge weights from the set $\{-W, \dots, 0, \dots, +W\}$. Our algorithm works in time $\tilde{O}(Wn^\omega)$, where ω is the matrix multiplication exponent. The best known bound on ω is $\omega < 2.376$ given by Coppersmith and Winograd [2]. Our result improves upon the previous fastest algorithms in the case of dense graphs with small integer weights. The same complexity result for the SSSP problem has been obtained independently by Yuster and Zwick [14]. Their result is based on a distance oracle that after $\tilde{O}(Wn^\omega)$ preprocessing time can answer distance queries in $O(n)$ time.

The rest of the paper is organized as follows. In the remainder of this introductory section, we summarize the results in linear algebra algorithms and recall

* Research supported by KBN grant 4T11C04425.

Table 1. The complexity results for the SSSP problem with negative weights. The bold font indicates an asymptotically best bound in the table.

Complexity	Author
$O(n^4)$	Shimbel (1955) [12]
$O(n^2mW)$	Ford (1956) [7]
$O(nm)$	Bellman (1958) [1], Moore (1959) [8]
$O(n^{\frac{3}{4}}m \log W)$	Gabow (1983) [4]
$O(\sqrt{nm} \log(nW))$	Gabow and Tarjan (1989) [5]
$O(\sqrt{nm} \log(W))$	Goldberg (1993) [6]
$\tilde{O}(n^\omega W)$	this paper and Yuster and Zwick [14]

the randomization technique used later in this paper. In Section 2 we present our algorithm for computing the single source distances. In Section 3 we show how our algorithm can be used to detect negative weight directed cycles. Finally in Section 4 we give the algorithm for computing the lightest paths tree.

1.1 Linear Algebra Algorithms

The interaction of the matrix multiplication and linear algebra is well understood. The best known algorithms for many problems in linear algebra work in matrix multiplication time, i.e., the determinant of a $n \times n$ matrix A , or the solution to the linear system of equations, can be computed in $O(n^\omega)$ arithmetic operations. Very recently Storjohann [13] has shown that these problems for polynomial matrices can be solved with the same exponent.

Theorem 1 (Storjohann '03). *Let $A \in K[x]^{n \times n}$ be a polynomial matrix of degree d and $b \in K[x]^{n \times 1}$ be a polynomial vector of the same degree, then*

- *rational system solution $A^{-1}b$ (Algorithm 5 [13]),*
- *determinant $\det(A)$ (Algorithm 10 [13]),*

can be computed in $\tilde{O}(n^\omega d)$ operations in K , with high probability.

1.2 Zippel-Schwartz Lemma

Almost always when we apply algebraic methods to construct graph algorithms, the problem is reduced to testing if some polynomial is non-zero. For example, if we want to check if a given vertex is reachable in the graph from another one, then we can test if appropriately defined adjacency matrix is non-singular [9]. In order to verify that the matrix is non-singular, we compute its determinant, which is a polynomial in the entries of the matrix. In this case we cannot use symbolic computation because the polynomial may have exponentially many terms and it would give an exponential time algorithm. The following lemma due to Zippel [15] and Schwartz [11] can be used to overcome this obstacle.

Lemma 2. *If $p(x_1, \dots, x_m)$ is a non-zero polynomial of degree d with coefficients in a field and S is a subset of the field, then the probability that p evaluates to 0 on a random element $(s_1, s_2, \dots, s_m) \in S^m$ is at most $d/|S|$. We call such event false zero.*

Corollary 3. *If a polynomial of degree n is evaluated on random values modulo prime number p of length $(1 + c) \log n$, then the probability of false zero is at most $\frac{1}{n^c}$, for any $c > 0$.*

Note that in the standard computation model with word size $O(\log n)$, the finite field arithmetic modulo p , except division, can be realized in constant time. To realize division we need $O(\log n)$ time, but divisions in our algorithms are not the dominating operations.

2 Shortest Paths

A *weighted directed n -vertex graph* G is a tuple $G = (V, E, w, W)$, where the vertex set is given by $V = \{1, \dots, n\}$, $E \subseteq V \times V$ denotes the edge set, and the function $w : E \rightarrow \{-W, \dots, 0, \dots, W\}$ ascribes weights to the edges.

Consider a path $p = v_1, v_2, \dots, v_k$ of length k . The weight of this path is given by $w(p) = \sum_{i=1}^{k-1} w((v_i, v_{i+1}))$. The distance from v to u in G , denoted by $\text{dist}_G(v, u)$, is equal to the minimal weight of the paths starting at v and ending in u .

In the *single source shortest paths (SSSP) problem* we are given a weighted directed graph G and a single vertex v in G , and we want to compute the distances from v to all other vertices in G . If there is a negative weight cycle in G reachable from v then the distances are undefined. The algorithm for SSSP problem should detect and report such a case.

In order to compute the distances in the graph we extend the method for computing the transitive closure introduced by Sankowski [9] and then used in [10] for dynamically computing distances in graphs with positive integer weights. Let us define a *symbolic adjacency matrix* of the weighted directed graph $G = (V, E, w, W)$ to be the $n \times n$ matrix $\tilde{A}(G)$ such that

$$\tilde{A}(G)_{i,j} = \begin{cases} y^W & \text{if } i = j \\ x_{i,j} y^{w((i,j))+W} & \text{if } (i, j) \in E, \\ 0 & \text{otherwise,} \end{cases}$$

where $x_{i,j}$ are unique variables corresponding to the edges of G . Note that $\tilde{A}(G)$ is a matrix polynomial of degree $2W$. In the following we denote by $\text{deg}_y^*(q)$ the smallest degree term of y in the multi-variable polynomial q .

Lemma 4. *Let G be a directed weighted graph without negative weight cycles and let $\tilde{A}(G)$ be the symbolic adjacency matrix of G . The weight of the lightest path in G from i to j is given by*

$$\text{dist}_G(i, j) = \text{deg}_y^* \left(\text{adj} \left(\tilde{A}(G) \right)_{i,j} \right) - (n - 1)W.$$

Moreover, all non-zero terms in $\text{adj} \left(\tilde{A}(G) \right)_{i,j}$ are non-zero in a finite field \mathbb{Z}_p .

Proof. For a given matrix A , we denote by $A^{i,j}$ the matrix A with elements in i -th row and j -th column set to zero except that $A_{i,j} = 1$. From the definition of the adjoint we have that $\text{adj}(A)_{i,j} = \det(A^{j,i})$. Applying this formula to $\tilde{A}(G)$ we obtain

$$\text{adj}\left(\tilde{A}(G)\right)_{i,j} = \det\left(\tilde{A}(G)^{j,i}\right) = \sum_{p \in \Gamma_n} \text{sgn}(p) \prod_{k=1}^n \tilde{A}(G)_{k,p_k}^{j,i}, \tag{1}$$

where Γ_n is the set of n element permutations, and $\text{sgn}(p)$ is the sign of the permutation p .

A permutation p defines a set of directed edges $E_p = \{(i, p_i) : 1 \leq i \leq n \text{ and } i \neq p_i\}$ and a set of vertices $V_p = \{i : 1 \leq i \leq n \text{ and } i = p_i\}$. The edge set E_p is a set of cycles given by the non-zero length cycles of p and the set V_p is the set of vertices covered by zero length cycles. Note that we have

$$\prod_{k=1}^n \tilde{A}(G)_{k,p_k}^{j,i} = \prod_{(k,l) \in E_p} \tilde{A}(G)_{k,l}^{j,i} \prod_{k \in V_p} \tilde{A}(G)_{k,k}^{j,i} =$$

From the definition of $\tilde{A}(G)^{j,i}$, we see that this product is non-zero if the edge set E_p contains the edge (j, i) and the other edges in E_p are in G , i.e., $E_p - (j, i) \subseteq E$. Let us denote by \mathcal{C}_G the set of all sets of disjoint directed cycles in G . Now by using the definition of $\tilde{A}(G)$ we can write

$$\begin{aligned} &= \prod_{(k,l) \in E_p - (j,i)} \tilde{A}(G)_{k,l}^{j,i} \prod_{k \in V_p} \tilde{A}(G)_{k,k}^{j,i} = \\ &= \prod_{(k,l) \in E_p - (j,i)} x_{k,l} y^{w((k,l)) + W} \prod_{k \in V_p} y^W = \\ &= y^{W|E_p - (j,i)|} y^{W|V_p|} \prod_{(k,l) \in E_p - (j,i)} x_{k,l} y^{w((k,l))} = \end{aligned}$$

We have $|E_p| + |V_p| = n$ and so,

$$= y^{W(n-1)} \prod_{(k,l) \in E_p - (j,i)} x_{k,l} y^{w((k,l))}.$$

By plugging the above equality into (1) we get

$$\begin{aligned} &\text{deg}_y^* \left(\text{adj}\left(\tilde{A}(G)\right)_{i,j} \right) - (n-1)W = \\ &= \text{deg}_y^* \left(\sum_{p \in \Gamma_n} \text{sgn}(p) \prod_{k=1}^n \tilde{A}(G)_{k,p_k}^{j,i} \right) - (n-1)W = \\ &= \min_{p \in \Gamma_n} \left\{ \text{deg}_y^* \left(\prod_{(k,l) \in E_p - (j,i)} x_{k,l} y^{w((k,l))} \right) \right\} = \end{aligned}$$

$$\begin{aligned}
 &= \min_{p \in \Gamma_n} \left\{ \deg_y^* \left(y^{\sum_{(k,l) \in E_p - (j,i)} w((k,l))} \prod_{(k,l) \in E_p - (j,i)} x_{k,l} \right) \right\} = \\
 &= \min_{E_p \in \mathcal{C}_G, (j,i) \in E_p} \left\{ \sum_{(k,l) \in E_p - (j,i)} w((k,l)) \right\} = \\
 &= \min_{E_p \in \mathcal{C}_G, (j,i) \in E_p} \left\{ \sum_{\substack{c \subseteq E_p, \\ c \text{ is a cycle}}} w(c) \right\} - w((j,i)) = \tag{2}
 \end{aligned}$$

In G there are no negative weight cycles, so the minimum is achieved for E_p containing only one cycle, because the other cycles can only increase the value (2). Thus we get

$$= \min_{\substack{c \subseteq E, (j,i) \in c, \\ c \text{ is a cycle}}} \{w(c) - w((j,i))\}.$$

where we take the minimum over the cycles c containing the edge (j,i) . Note that the rest of the cycle c forms a path from i to j with weight $w(c) - w((j,i))$. Hence the smallest term in the minimum corresponds to the lightest path, because we sum over all cycles.

Notice that each monomial in (1) has coefficient ± 1 , so each non-zero term in $\text{adj}(\tilde{A}(G))_{i,j}$ is also non-zero over \mathcal{Z}_p , for any prime $p \geq 2$.

The above theorem shows that the adjoint matrix encodes the distances in the graph. In connection with Theorem 1 we can formulate the following

Theorem 5. *Let $G = (V, E, w, W)$ be a weighted directed graph with integer edge weights and let v be a vertex in G . Assuming that G has no negative weight cycles, the distances from the vertex v can be computed in $\tilde{O}(Wn^\omega)$ time, with high probability.*

Proof. The algorithm works as follows

1. choose a prime number p of length $3\lceil \log n \rceil$,
2. substitute all variables $x_{i,j}$ in the matrix $\tilde{A}(G)$ for random numbers from the set $\{1, \dots, p - 1\}$ and let the resulting matrix be A ,
3. using Theorem 1 compute $\det(A^T)$ and $(A^T)^{-1}e_v$, where e_v is the v -th versor, i.e., n dimensional zero vector except 1 on the i -th place.
4. with high probability

$$\text{dist}_G(v, u) = \deg_y^* \left(\left(\det(A^T) (A^T)^{-1} e_v \right)_u \right) - (n - 1)W,$$

for all $u \in V$.

We have $(\det(A^T) (A^T)^{-1} e_v)_u = (\text{adj}(A^T)e_v)_u = \text{adj}(A^T)_{u,v} = \text{adj}(A)_{v,u}$. Note that the coefficient before the smallest degree term of y in $\tilde{A}(G)_{v,u}$ is a non-zero polynomial of degree n (see Theorem 4). The smallest degree term in $\text{adj}(A)_{v,u}$ is its evaluation over \mathcal{Z}_p . From Corollary 3 we get that the probability of a false zero is $O(n^{-2})$, and hence the probability that any of the $O(n)$ distances is computed incorrectly is $O(n^{-1})$.

Note that the above algorithm computes also the value $\text{dist}_G(v, v)$, and $\text{dist}_G(v, v) = 0$. However, when there are some negative weight cycles in G then $\text{dist}_G(v, v) \neq 0$. This observation will be used in the next section for computing distances in the presence of negative weight cycles.

3 Detecting Negative Weight Cycles

In the previous section we have shown an algorithm for computing distances in the graph only when there are no negative weight cycles. In this section we show how the above method can be used for detecting negative weight cycles.

Theorem 6. *Let G be a weighted directed graph and let $\tilde{A}(G)$ be the symbolic adjacency matrix of G . The value $\text{deg}_y^*(\det(\tilde{A}(G))) - nW$ is equal to the minimum weight of a disjoint set of cycles in G . Moreover, all non-zero terms in $\det(\tilde{A}(G))$ are non-zero in a finite field \mathbb{Z}_p .*

Proof. Similarly as in Theorem 4 we write

$$\det(\tilde{A}(G)) = \sum_{p \in \Gamma_n} \text{sgn}(p) \prod_{k=1}^n \tilde{A}(G)_{k,p_k},$$

and after the same transformations we obtain at the end,

$$\text{deg}_y^*(\det(\tilde{A}(G))) - nW = \min_{E_p \in \mathcal{C}_G} \left\{ \sum_{c \subseteq E_p, c \text{ is a cycle}} w(c) \right\},$$

and the theorem follows.

The above theorem can be very simply turned into an algorithm for detecting negative weight cycles as the following theorem states.

Theorem 7. *Let $G = (V, E, w, W)$ be a weighted directed graph with integer edge weights. A negative weight cycle in G can be detected in $\tilde{O}(Wn^\omega)$ time, with high probability.*

Proof. Any negative cycle in the graph G can be detected in the following way

1. choose a prime number p of length $2\lceil \log n \rceil$,
2. substitute all variables $x_{i,j}$ in the matrix $\tilde{A}(G)$ for random numbers from the set $\{1, \dots, p - 1\}$ and let the resulting matrix be A ,
3. using Theorem 1 compute $\det(A)$,
4. if $\text{deg}_y^*(\det(A)) - nW < 0$ then, with high probability, G has a negative weight cycle.

If there are negative weight cycles in G then the minimum weight of the disjoint set of cycles is also negative, so the correctness of the above algorithm follows from Theorem 6 and Corollary 3.

In order to detect a negative weight cycle reachable from a given vertex v we have to be a bit more careful than in the above theorem.

Theorem 8. *Let $G = (V, E, w, W)$ be a weighted directed graph with integer edge weights and let v be a vertex in G . There is an $\tilde{O}(Wn^\omega)$ time algorithm that, with high probability,*

- detects if there is a negative weight cycle reachable from v in G ,
- and if there is no such cycle it computes the distances from v .

Proof. Note that if there are negative weight cycles in G then the algorithm from Theorem 5 returns wrong, too short distances, because negative weight cycles can contribute to the sum in (2). However, when there are no negative weight cycles the distances are computed correctly.

In order to detect negative weight cycles we use the standard Bellman-Ford method [3]. The idea is as follows

1. run the algorithm from Theorem 5 to compute the distances from v , and let the computed values be $d_{v,u}$, for all $u \in G$,
2. define new distances $d'_{v,u}$ to be

$$d'_{v,u} := d_{v,u} - d_{v,v}.$$

3. run the single iteration of the Bellman-Ford method on the distances d' ,
4. if d' remained unchanged in the Bellman-Ford step then return d' , else there is a negative length cycle reachable from v .

There are three possibilities

- **There are no negative length cycles in G** — We have $d_{v,v} = 0$ and from Theorem 5 we get that the distances are computed correctly.
- **No negative length cycle in G is reachable from v** — In (2) the distance $\text{dist}_G(v, u)$ is given by the sum over the sets of cycles in G . Note that the path from v to u is disjoint from any negative weight cycle in G , and so this sum is reduced by the minimum weight of the disjoint set of cycles in G . Note that $\text{adj}(\tilde{A}(G))_{v,v} = \det(\tilde{A}(G - \{v\}))$, where $G - \{v\}$ is G with removed vertex v . Hence from Theorem 6 we get that $\text{dist}_G(v, v) = \text{deg}_y^*(\det(\tilde{A}(G - \{v\}))) - (n - 1)W$ is equal to the minimum weight of the disjoint set of cycles in $G - \{v\}$. Because the negative weight cycles are disjoint from v this gives also the minimum weight of the disjoint set of cycles in G .
- **There is a negative length cycle in G reachable from v** — In this case the algorithm from Theorem 5 computes wrong distances for the vertices reachable from v . However, whichever the distances are, they will be changed by the Bellman-Ford step, because there is a negative weight cycle.

The above idea can be used for computing distances from the vertex v to all other vertices which are reachable only by paths that do not touch any negative weight cycle. The details will be given in the full version of this paper.

4 Constructing the Lightest Paths Tree

In this section we show how the lightest paths tree in the weighted directed graph can be constructed.

Theorem 9. *Let $G = (V, E, w, W)$ be a weighted directed graph with integer edge weights and let v be a vertex in G . A lightest paths tree rooted at v can be constructed in $\tilde{O}(Wn^\omega)$ time, with high probability.*

Proof. The algorithm for computing the lightest paths tree works as follows

1. run the algorithm from Theorem 8 to compute the distances from v , and let the computed values be $d_{v,u}$, for all $u \in G$,
2. let w' be a new weight function in G defined by

$$w'((i, j)) := w((i, j)) - d_{v,j} + d_{v,i}, \quad (3)$$

3. compute the lightest paths tree in $G' = (V, E, w')$ using the Dijkstra's algorithm.

The distances $\text{dist}_G(v, u)$ form a potential function with the following properties [3],

- (i) the weights defined in (3) are positive,
- (ii) each path p from v to u is lightest with respect to w iff it is lightest with respect to w' .

Because of (i) we can use the Dijkstra algorithm. By (ii) the lightest paths tree returned by the Dijkstra's algorithm for w' is also the lightest paths tree for w .

Acknowledgments

I would like to thank my favorite supervisor Krzysztof Diks for his unwavering support and Marcin Mucha for many helpful discussions.

References

1. R. Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.
2. D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *Proceedings of the nineteenth annual ACM conference on Theory of computing*, pages 1–6. ACM Press, 1987.
3. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge Mass., 1990.
4. H.N. Gabow. Scaling Algorithms for Network Problems. *J. Comput. Syst. Sci.*, 31(2):148–168, 1985.
5. H.N. Gabow and R.E. Tarjan. Faster Scaling Algorithms for Network Problems. *SIAM J. Comput.*, 18(5):1013–1036, 1989.

6. A. V. Goldberg. Scaling Algorithms for the Shortest Paths Problem. *SIAM J. Comput.*, 24(3):494–504, 1995.
7. L.R. Ford Jr. Network Flow Theory. Paper P-923, The RAND Corporation, Santa Monica, California, August 1956.
8. E. F. Moore. The Shortest Path Through a Maze. In *Proceedings of the International Symposium on the Theory of Switching*, pages 285–292. Harvard University Press, 1959.
9. P. Sankowski. Dynamic Transitive Closure via Dynamic Matrix Inverse. In *Proceedings of the 45th annual IEEE Symposium on Foundations of Computer Science*, pages 248–255, 2004.
10. P. Sankowski. Subquadratic Algorithm for Dynamic Shortest Distances. In *Proceedings of the 11th International Computing and Combinatorics Conference (COCON'05)*, LNCS 3595, 2005.
11. J. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM*, 27:701–717, 1980.
12. A. Shimbel. Structure in Communication Nets. In *In Proceedings of the Symposium on Information Networks*, pages 199–203. Polytechnic Press of the Polytechnic Institute of Brooklyn, Brooklyn, 1955.
13. A. Storjohann. High-order lifting and integrality certification. *J. Symb. Comput.*, 36(3-4):613–648, 2003.
14. R. Yuster and U. Zwick. Answering distance queries in directed graphs using fast matrix multiplication. In *The 46th Annual Symposium on Foundations of Computer Science (FOCS'05)*, 2005.
15. R. Zippel. Probabilistic algorithms for sparse polynomials. In *International Symposium on Symbolic and Algebraic Computation*, volume 72 of *Lecture Notes in Computer Science*, pages 216–226, Berlin, 1979. Springer-Verlag.

Computing Common Intervals of K Permutations, with Applications to Modular Decomposition of Graphs

Anne Bergeron¹, Cedric Chauve¹, Fabien de Montgolfier²,
and Mathieu Raffinot³

¹ Département d'informatique, Université du Québec à Montréal, Canada
{bergeron.anne, chauve.cedric}@uqam.ca

² LIAFA, Université Denis Diderot - Case 7014, 2 place Jussieu,
F-75251 Paris Cedex 05, France
fm@liafa.jussieu.fr

³ CNRS - Laboratoire Génome et Informatique, Tour Evry 2, 523,
Place des Terrasses de l'Agora, 91034 Evry, France
raffinot@genopole.cnrs.fr

Abstract. We introduce a new way to compute common intervals of K permutations based on a very simple and general notion of generators of common intervals. This formalism leads to simple and efficient algorithms to compute the set of all common intervals of K permutations, that can contain a quadratic number of intervals, as well as a linear space basis of this set of common intervals. Finally, we show how our results on permutations can be used for computing the modular decomposition of graphs in linear time.

1 Introduction

The notion of *common interval* was introduced in [16] in order to model the fact that, when comparing genomes, a group of genes can be rearranged but still remain connected. In [16], Uno and Yagiura proposed a first algorithm that computes the set of common intervals of a permutation P with the identity permutation in time $O(n + N)$, where n is the length of P , and N is the number of common intervals. However, N can be of size $O(n^2)$, thus the algorithm of Uno and Yagiura has an $O(n^2)$ time complexity. Heber and Stoye [10] defined a subset of size $O(n)$ of the common intervals of K permutations, called *irreducible intervals*, that forms a basis of the set of all common intervals: every common interval is a chain overlapping irreducible intervals. They proposed an $O(Kn)$ time algorithm to compute the set of irreducible intervals of K permutations, based on Uno and Yagiura's algorithm.

One of the drawbacks of these algorithms is that properties of Uno and Yagiura's algorithm are difficult to understand [4]. Even the authors describe their $O(n + N)$ algorithm as "*quite complicated*", and, in practice, simpler $O(n^2)$ algorithms run faster on randomly generated permutations [16]. On the other

hand, Heber and Stoye's algorithms rely on a complex data structure that mimics what is known, in the theory of modular decomposition of graphs, as the PQ -trees of *strong intervals*. An incentive to revisit this problem is the central role that these PQ -trees seem to play in the field of comparative genomics. Strong intervals can be used to identify significant groups of genes that are conserved between genomes [11], or as guides to reconstruct evolution scenarios [1,8].

In order to design alternative efficient algorithms to compute common intervals, we propose a theoretical framework for common intervals based on generating families of intervals. For two permutations, these families can be computed by straightforward $O(n)$ algorithms that use only tables and stacks as data structures, and that upgrade trivially to the case of K permutations. Using these families, we compute common intervals with simple $O(n + N)$ and $O(n)$ algorithms whose properties can be readily verified. We then link this work to previous studies on common interval, and we propose a new canonical representation of the family of common intervals that is simpler than the PQ -trees.

Finally, we extend our approach to the classical graph problem of *modular decomposition* that aims to efficiently compute a compact representation of the modules of a graph. The first linear time algorithms that were developed [6,13] are rather complex and many efforts have been made in the design of decomposition algorithms that are efficient in practice, even if they do not run in linear time but in quasi-linear time [7,14].

The article is structured as follows. In Section 2, we describe the notion of generators of common intervals and how to compute generators of K permutations of size n in $O(Kn)$ time. The third section explains how to generate the set of all N common intervals in $O(n + N)$ using a generator. Section 4 describes a new linear space basis of common intervals, called the canonical generator, and describes the relations between this new basis and the classical basis of strong intervals. Section 5 contains a simple linear-time algorithm to construct the strong intervals basis, given the canonical generator. Finally, in Section 6, we extend our results to the modular decomposition of graphs. Some proofs are omitted, however they can be found in [2].

2 Common Intervals and Generators

A *permutation* P on n elements is a complete linear order on the set of integers $\{1, 2, \dots, n\}$. We denote Id_n the identity permutation $(1, 2, \dots, n)$. An *interval* of a permutation $P = (p_1, p_2, \dots, p_n)$ is a set of consecutive elements of permutation P . An interval of a permutation will be denoted by either giving its left and right bounds, such as $[i, j]$, or by giving the list of its elements, such as $(p_i, p_{i+1}, \dots, p_j)$. An interval $[i, j] = (i, i + 1, \dots, j)$ of the identity permutation will be simply denoted by $(i..j)$.

Definition 1. Let $\mathcal{P} = \{P_1, P_2, \dots, P_K\}$ be a set of K permutations on n elements. A *common interval* of \mathcal{P} is a set of integers that is an interval in each permutation of \mathcal{P} .

The set $\{1, 2, \dots, n\}$ and all singletons are always common intervals of any non empty set of permutations, they are called *trivial* intervals. In the sequel, we assume, without loss of generality, that the set \mathcal{P} contains the identity permutation Id_n . A common interval of \mathcal{P} can thus be denoted as an interval $(i..j)$ of the identity permutation.

Definition 2. Let $\mathcal{P} = \{Id_n, P_2, \dots, P_K\}$ be a set of K permutations on n elements. A *generator* for the common intervals of \mathcal{P} is a pair (R, L) of vectors of size n such that:

1. $R[i] \geq i$ and $L[j] \leq j$ for all $i, j \in \{1, 2, \dots, n\}$,
2. $(i..j)$ is a common interval of \mathcal{P} if and only if $(i..j) = (i..R[i]) \cap (L[j]..j)$.

The following proposition shows how to construct a generator for a union of sets of permutations, given generators for each set. If X and Y are two vectors, we denote by $\min(X, Y)$ the vector $\min(X[1], Y[1]), \dots, \min(X[n], Y[n])$.

Proposition 1. Let (R_1, L_1) and (R_2, L_2) be generators for the common intervals of two sets \mathcal{P}_1 and \mathcal{P}_2 of permutations, both containing the identity permutation. The pair $(\min(R_1, R_2), \max(L_1, L_2))$ is a generator for the common intervals of $\mathcal{P}_1 \cup \mathcal{P}_2$.

Proof. First note that $(i..j) = (i..R[i]) \cap (L[j]..j)$ if and only if $L[j] \leq i \leq j \leq R[i]$. Interval $(i..j)$ is a common interval of $\mathcal{P}_1 \cup \mathcal{P}_2$ if and only if it is a common interval of both \mathcal{P}_1 and \mathcal{P}_2 , which is equivalent to $L_1[j] \leq i \leq j \leq R_1[i]$ and $L_2[j] \leq i \leq j \leq R_2[i]$, and finally to $\max(L_1[j], L_2[j]) \leq i \leq j \leq \min(R_1[i], R_2[i])$ □

Proposition 1 implies that, given an $O(n)$ algorithm for computing generators for the common intervals of two permutations, we can easily deduce an $O(Kn)$ algorithm for computing a generator for the common intervals of K permutations.

Generators are far from unique, but some are easier to compute than others. Identifying good generators is a crucial step in the design of efficient algorithms to compute common intervals. The remaining of this section focuses on particular classes of generators that turn out to have interesting properties with respect to computations.

Definition 3. Let $P = (p_1, \dots, p_n)$ be a permutation on n elements. For each element p_i , we define two intervals containing p_i :

- $IMax[p_i]$ is the largest interval of P whose elements are all $\geq p_i$,
- $IMin[p_i]$ is the largest interval of P whose elements are all $\leq p_i$.

And the following two integers:

- $Sup[p_i]$ is the largest integer such that $(p_i..Sup[p_i]) \subseteq IMax[p_i]$,
- $Inf[p_i]$ is the smallest integer such that $(Inf[p_i]..p_i) \subseteq IMin[p_i]$.

Remark that $(p_i..Sup[p_i])$ and $(Inf[p_i]..p_i)$ are intervals of the identity permutation, but not necessarily intervals of permutation P . For example, if $P = (1\ 4\ 7\ 5\ 9\ 6\ 2\ 3\ 8)$, we have: $IMax[5] = (7\ 5\ 9\ 6)$ and $Sup[5] = 7$, and $IMin[8] = (6\ 2\ 3\ 8)$ and $Inf[8] = 8$.

Proposition 2. *The pair of vectors (Sup, Inf) is a generator for the common intervals of P and Id_n .*

Proof. Suppose that $(i..j)$ is a common interval of P and Id_n , then $Sup[i] \geq j$ and $Inf[j] \leq i$ since all elements in the set $(i..j)$ are consecutive in permutation P . Thus $(i..j) = (i..Sup[i]) \cap (Inf[j]..j)$. On the other hand, suppose that $Sup[i] \geq j$ and $Inf[j] \leq i$, then $IMax[i]$ contains j and $IMin[j]$ contains i . Since both $IMax[i]$ and $IMin[j]$ are intervals of P , their intersection is an interval and is equal to $(i..j)$. □

Algorithm 1. Computing the generator (Sup, Inf)

```

Inf[1] ← 1, Sup[n] ← n.
For k from 1 to n, m[k] ← k, M[k] ← k.
For k from 2 to n
    While m[k] - 1 is in IMin[k], m[k] ← m[m[k] - 1]
    Inf[k] ← m[k]
For k from n - 1 to 1
    While M[k] + 1 is in IMax[k], M[k] ← M[M[k] + 1]
    Sup[k] ← M[k]
    
```

Proposition 3. *Let P be a permutation on n elements. If the bounds of intervals $IMax[k]$ and $IMin[k]$ are known for all k , then Algorithm 1 computes (Sup, Inf) in $O(n)$ time.*

Proof. We first show that Algorithm 1 is correct. Suppose that, at the beginning of the k -th iteration of the second *For* loop, $Inf[k'] = m[k']$ for all $k' < k$, and $m[k] \in IMin[k]$. This is the case at the beginning of iteration $k = 2$, since $Inf[1] = 1$. By definition, $Inf[k] \leq k$, thus before entering the *While* loop, we have $Inf[k] \leq m[k]$. If the test $m[k] - 1 \in IMin[k]$ of the *While* loop is true, then $Inf[k] \leq m[k] - 1$, implying $Inf[k] \leq Inf[m[k] - 1]$. Since $Inf[m[k] - 1] = m[m[k] - 1]$ by hypothesis, the instruction in the *While* loop preserves the invariant $Inf[k] \leq m[k]$. When the test of the *While* loop becomes false, then $Inf[k]$ is greater than $m[k] - 1$, thus $Inf[k] = m[k]$. The proof of correctness for *Sup* is similar.

Suppose that $IMin[k] = [i, j]$, then the tests in the *While* loops can be done in constant time using the inverse of permutation P . The total time complexity follows from the fact that the instruction within the *While* loop is executed exactly $n - 1$ times. Indeed, consider, at any point of the execution of the algorithm, the collection of intervals $(m[k]..k)$ of the identity permutation that are not contained in any other interval of this type. After the initialization loop, we have n such intervals, and at the completion of the algorithm, there is only one, namely $(1..n)$, since $Inf[n] = 1$. The instruction in the *While* loop merges two consecutive intervals into one and there can be at most $n - 1$ of these merges. □

The computation of the bounds of intervals $IMax[k]$ and $IMin[k]$, as well as the computation of the inverse of permutation P , are quite straightforward. As an example, Algorithm 2 shows how to compute the left bound of $IMax[p_i]$.

Proposition 4. *Let $P = (p_1, \dots, p_n)$ be a permutation on n elements, Algorithm 2 computes the left bound $IMax[p_i]$ in $O(n)$ time.*

Proof. The time complexity of Algorithm 2 is immediate since each position is stacked once. The correctness of $LMax$ relies on the fact that, at the beginning of the i -th iteration, the position j of the nearest left element such that $p_j < p_i$ must be in the stack. If it was not the case, then an element smaller than p_j was found between the positions j and i , contradicting the definition of position j \square

Algorithm 2. Computing the left bound $LMax[p_i]$ of $IMax[p_i]$ for all p_i

```

 $S$  is a stack of positions;  $s$  denotes the top of  $S$ .
Push 0 on  $S$ 
 $p_0 \leftarrow 0$ 
For  $i$  from 1 to  $n$ 
    While  $p_i < p_s$  Pop the top of  $S$ 
     $LMax[p_i] \leftarrow s + 1$ 
    Push  $i$  on  $S$ 
    
```

To summarize the results of this section, we have:

Theorem 1. *Let $\mathcal{P} = \{Id_n, P_2, \dots, P_K\}$ be a set of K permutations on n elements. A generator for the common intervals of \mathcal{P} can be computed in $O(Kn)$ time.*

3 Common Intervals of K Permutations in Optimal Time

We now turn to the problem of generating all common intervals of K permutations in $O(N)$ time, where N is the number of such common intervals, given a generator satisfying the following property.

Definition 4. Two sets A and B *commute* if either $A \subseteq B$, or $B \subseteq A$, or A and B are disjoint, and otherwise they *overlap*. A collection \mathcal{C} of sets is *commuting* if, for any pair of sets A and B in \mathcal{C} , A and B commute. A generator (R, L) for the common intervals of $\mathcal{P} = \{Id_n, P_2, \dots, P_K\}$ is *commuting* if both the collections $\{(i..R[i])\}_{i \in (1..n)}$ and $\{(L[i]..i)\}_{i \in (1..n)}$ are commuting. If (R, L) is a commuting generator, we define $Support[i]$, for $i > 1$, to be the greatest integer $j < i$ such that $R[i] \leq R[j]$.

It turns out that generators defined in Section 2 are commuting. Indeed, generators defined in Proposition 1 are commuting if they are constructed with generators (R_1, L_1) and (R_2, L_2) that are commuting. This is a consequence of the fact that if $a < b$ and $a' < b'$ then $\min(a, a') < \min(b, b')$ and $\max(b, b') > \max(a, a')$. For the generator (Sup, Inf) , we have:

Proposition 5. *The generator (Sup, Inf) for the common intervals of permutations P and Id_n is commuting.*

Proposition 6. *Given a commuting generator (R, L) , Algorithm 3 computes the values $Support[i]$, for all $i > 1$, in linear time.*

Theorem 2. *Given a commuting generator (R, L) , Algorithm 4 outputs all common intervals of a set \mathcal{P} of K permutations on n elements, in $O(n + N)$ time, where N is the number of common intervals of the set \mathcal{P} .*

Proof. The time complexity of Algorithm 4 is immediate. Suppose that interval $(i..j)$ is identified by the algorithm. At the start of the j -th iteration of the *For* loop, $i = j$, thus $j \leq R[i]$. If the test of the *While* loop is true, then $i \geq L[j]$, and $(i..j)$ is a common interval. If $i' = Support[i]$, then $R[i'] \geq R[i]$, thus $j \leq R[i']$ at the end of the *While* loop.

On the other hand, if $(i..j)$ is a common interval of \mathcal{P} , with $i < j$, then $Support[j] \geq i$, since $R[i] \geq R[j]$. Let i' be the smallest integer such that $i < i'$ and $(i'..j)$ is identified by Algorithm 4 as a common interval. Such an interval exists, since $(j..j)$ is a common interval. Finally, $Support[i'] = i$ since $Support[i']$ must be greater than or equal to i . If it is greater, then $(Support[i']..j)$ is a common interval, contradicting the definition of i' . □

Algorithm 3. Computing $Support[i]$ for a commuting generator (R, L)

```

S is an empty stack; s denotes the top of S
Push 1 on S
For i from 2 to n
    While  $R[s] < i$  Pop the top of S
     $Support[i] \leftarrow s$ 
    Push i on S
    
```

Algorithm 4. Common intervals of a set \mathcal{P} given a generator (R, L)

```

For j from n to 1
     $i \leftarrow j$ 
    While  $i \geq L[j]$ 
        Output  $(i..j)$  (* Interval  $(i..j)$  is a common interval of the set  $\mathcal{P}$  *)
         $i \leftarrow Support[i]$ 
    
```

4 Canonical Representations of Closed Families

The common intervals of a set of permutations is an example of a more general families of intervals, the *closed* families. In this section, we develop a new canonical representation for such families, based on the generators of the previous section.

A *closed* family \mathcal{F} of intervals of the identity permutation on n elements is a family that contains all singletons, the interval $(1..n)$ and that has the following property: if $(i..k)$ and $(j..l)$ are in \mathcal{F} , and $i \leq j \leq k \leq l$, then $(i..j)$, $(j..k)$, $(k..l)$

and $(i..l)$ belong to \mathcal{F} . It is easy to extend Definition 2 of generators to the more general case of closed families.

A classical result [3] establishes a bijection between the PQ -trees with n leaves and closed families of Id_n , thus allowing a representation of size $O(n)$ for any closed family. Among all possible generators, the following ones will also provide a representation of size $O(n)$ for any closed family:

Definition 5. A generator (R, L) for a closed family \mathcal{F} is canonical if, for all $i \in (1..n)$, intervals $(i..R[i])$ and $(L[i]..i)$ belong to \mathcal{F} .

Proposition 7. Let \mathcal{F} be a closed family. The canonical generator of \mathcal{F} always exists, and it is unique and commuting.

Theorem 3. Given a commuting generator (R', L') , Algorithm 5 computes the canonical generator (R, L) of a closed family \mathcal{F} in $O(n)$ time.

Algorithm 5. Canonical generator (R, L) given a commuting generator (R', L')

```

The vector Support is obtained from  $R'$  using Algorithm 3
 $R[1] \leftarrow n$ 
For  $k$  from 2 to  $n$ 
     $R[k] \leftarrow k$ 
For  $k$  from  $n$  to 2
    If  $(Support[k]..R[k]) \in \mathcal{F}$ 
         $R[Support[k]] \leftarrow \max(R[k], R[Support[k]])$ 
(* Computation of  $L$  is similar, by defining the vector Support with respect
to  $L'$  *)
    
```

Example 1. Let $\mathcal{P} = \{Id_8, P_2\}$ and $\mathcal{Q} = \{Id_8, P_3\}$ with

$$Id_8 = (1, 2, 3, 4, 5, 6, 7, 8) \quad P_2 = (1, 3, 2, 4, 5, 7, 6, 8) \quad P_3 = (2, 8, 3, 4, 5, 6, 1, 7)$$

Figure 1 shows the generators (Sup, Inf) for the common intervals of \mathcal{P} and the common intervals of \mathcal{Q} ; a generator for the common intervals of the union $\mathcal{P} \cup \mathcal{Q}$ using the two generators (Sup, Inf) ; and the canonical generator for the common intervals of $\mathcal{P} \cup \mathcal{Q}$.

Compared to PQ -trees, this canonical representation of a closed family \mathcal{F} is much simpler since it uses only two arrays. Moreover, some operations, for example testing whether an interval $(i..j)$ belongs to the family \mathcal{F} , are also simpler using this representation. However, PQ -trees have the advantage of being recursive structures. Thus, in order to be complete, we next show how to transform one representation into the other using the key notion of *strong intervals*.

5 From Canonical Generators to Strong Intervals

A *strong interval* of \mathcal{F} is an interval that commutes with each interval of \mathcal{F} . In this section, we show how to compute the strong intervals, given a canonical

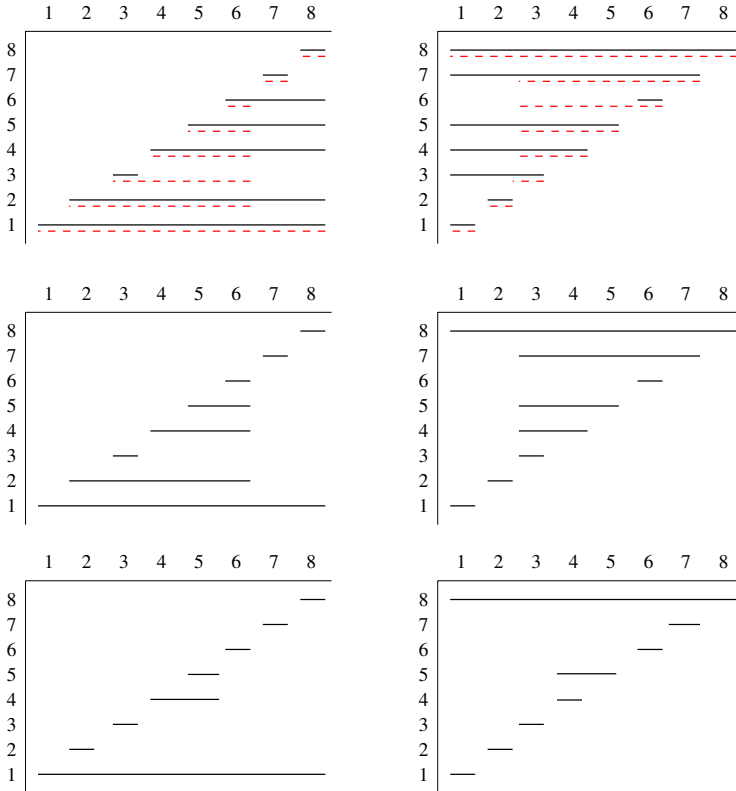


Fig. 1. The top two diagrams show the generators (Sup, Inf) of the common intervals of the set \mathcal{P} in solid lines, and of set \mathcal{Q} in dashed lines. A line in row i of the left diagram extends from column i to column $Sup(i)$, and a line in row i of the right diagram extends from column $Inf(i)$ to column i . The middle diagrams shows a generator for the common intervals of $\mathcal{P} \cup \mathcal{Q}$ constructed using Proposition 1. Finally, the bottom diagrams shows the canonical generator constructed by Algorithm 5.

generator. The structure of PQ -tree, the inclusion tree of the strong intervals, is another classical representation of an interval family [3]. This concept is investigated in [12], where it is explained how to compute the PQ -tree from the strong intervals.

Let (R, L) be the canonical generator. Consider the $4n$ bounds of intervals of the families $(i..R[i])$ and $(L[j]..j)$ for $i, j \in (1..n)$. Let (a_1, \dots, a_{4n}) be the list of these $4n$ bounds sorted in increasing order, with the left bounds placed before the right bounds when they are equal. For the example of Figure 1 this list is

$$(1, 1, 1, \bar{1}, 2, 2, \bar{2}, \bar{2}, 3, 3, \bar{3}, \bar{3}, 4, 4, 4, \bar{4}, 5, 5, \bar{5}, \bar{5}, 6, 6, \bar{6}, \bar{6}, 7, 7, \bar{7}, \bar{7}, 8, \bar{8}, \bar{8}, \bar{8})$$

where \bar{i} denotes a right bound. Such a list can be constructed easily by scanning the two vectors R and L , and by noting that each $i \in (1..n)$ is a left bound at least once, and a right bound at least once.

Proposition 8. *Given the ordered list (a_1, \dots, a_{4n}) of the $4n$ bounds of a canonical generator (R, L) , Algorithm 6 outputs the strong intervals of a closed family in $O(n)$ time.*

Algorithm 6. Computation of the strong intervals

```

 $S$  is a stack of bounds,  $s$  denotes the top of  $S$ 
For  $i$  from 1 to  $4n$ 
  If  $a_i$  is a left bound
    Push  $a_i$  on  $S$ 
  Else
    Output  $(s..a_i)$  (* Interval  $(s..a_i)$  is strong *)
    Pop the top of  $S$ 
    
```

6 Modular Decomposition

Let $G = (V, E)$ be a directed, finite, loopless graph, with $|V| = n$ and $|E| = m$. Undirected graphs may be seen as symmetrical directed graphs in this context. A *module* is a subset M of V that behaves like a single vertex: for $x \notin M$ either there are $|M|$ arcs that join x to all vertices of M , or no arc joins x to M , and conversely either there are $|M|$ arcs that join all vertices of M to x , or no arc joins M to x . A *strong* module does not overlap any other module. There may be up to 2^n modules in a graph (in the complete graph for instance) but there are at most $O(n)$ strong modules, and the *modular decomposition tree* based on the strong modules inclusion tree is sufficient to represent all modules [15]. The modular decomposition tree is indeed the *PQ*-tree of the family of modules.

Linear-time decomposition algorithms have been discovered [6,13] but remain rather complex. Simpler algorithms work in two steps: computing a factorizing permutation, and then building a tree representation on it. The first step was published in [9]. In this paper, we simplify the second step.

A *factorizing permutation* of a graph [5] is a permutation of the vertices of the graph in which every strong module of the graph is a *factor*, that is an interval of the permutation. Since the strong modules are a commuting family, every graph admits a factorizing permutation. A factorizing permutation of a graph can be computed in linear time [9]. In the following we assume, without loss of generality, that the vertex-set V is the set $\{1, ..n\}$ and that the identity permutation is a factorizing permutation of the graph.

Given an interval $(u..v)$ of the factorizing permutation, a vertex $x \notin (u..v)$ is a *splitter* of the interval if there are between 1 and $v - u$ arcs going from x to $(u..v)$, or if there are between 1 and $v - u$ arcs going from $(u..v)$ to x . A *right-module* is an interval $(u..v)$ with no splitters greater than v . A *left-module* is an interval $(u..v)$ with no splitters smaller than u . An *interval-module* is an interval $(u..v)$ with no splitters. Clearly interval-modules are modules. However, some modules are not interval-modules, but, according to the definition of a factorizing permutation, the strong modules of the graph are interval-modules.

It is well known that modules behave like intervals: unions, intersections or differences of two overlapping modules are modules. Thus:

Proposition 9. [15] *The interval-modules of a factorizing permutation of a graph G are a closed family. The strong intervals of this family are exactly the strong modules of the graph G .*

Definition 6. For a vertex v let $R[v]$ be the greatest integer such that $(v..R[v])$ is a left-module and $L[v]$ the smallest integer such that $(L[v]..v)$ is a right-module.

It can be proved that for every $w \in (L[v]..v)$, $(w..v)$ is a right-module, and for every $w < L[v]$, $(w..v)$ is not a right-module. For this reason $(L[v]..v)$ is called *the* maximal right-module ending at v . In a similar way, we can define the maximal left-module beginning at v . We have:

Proposition 10. *The pair (R, L) is a commuting generator of the interval-modules family.*

Proof. Interval $(u..v)$ is an interval-module if and only if $R[u] \geq v$ and $L[v] \leq u$, thus (R, L) is a generator. The family defined by R is commuting because if $(u..R[u])$ overlaps $(v..R[v])$, and if, without loss of generality, $u < v$, then $(u..R[v])$ is a left-module starting at u greater than the maximal left-module $(u..R[u])$, which is a contradiction. A similar argument shows that L also is commuting. □

In order to compute the maximal right-strong modules, we use a simplified version of an algorithm due to Capelle and Habib [5]. The algorithm to compute the maximal left-modules is similar.

Let us consider the maximal right-module $(L[v]..v)$ ending at v . If $L[v] > 1$, then there exists an $x > v$ that splits $(L[v] - 1..v)$, otherwise this right-module would not be maximal, and x therefore splits $(L[v] - 1..L[v])$, but does not split $(y - 1, y)$ for all $L[v] < y \leq v$. Based on this observation, Capelle and Habib algorithm proceeds in two steps. First, for every vertex v the *rightmost splitter* $s[v]$ is computed. It is the greatest vertex, if any, that splits the pair $(v - 1..v)$. Then a loop for v from n to 2 computes all the maximal right-modules $(L[x]..x)$ such that $v = L[x]$. Computing $s[v]$ can be done by a simultaneous scan of the adjacency lists of v and $v - 1$: the greatest element occurring in only one adjacency list is kept. This can be done in time proportional to the size of the adjacency lists. The computation of $s[v]$ for all v can therefore be done in $O(n + m)$ time, that is linear in the size of the graph. The second step is Algorithm 7. It clearly runs in $O(n)$ time, and its correctness relies on the following invariant:

Invariant. *At step v , for all vertices x in the stack, $(v..x)$ is a right-module, and for all $x > v$ not in the stack, $L[v] > v$.*

Proof. The invariant is initially true. Every step maintains it: if $s[v]$ does not exist then for all x in the stack $(v - 1..x)$ is a right-module, and $(v - 1..v)$ also is a right-module. And if $s[v]$ exists, (v) is the maximal right-module ending at v . For all $x < s[v]$ $(v - 1..x)$ is not a right-module and $(v..x)$ is therefore the maximal

right-module ending at x . For all $x \geq s[v]$ $(v - 1..x)$ is still a right-module, because $s[v]$ is the greatest of the splitters of $(v - 1, v)$. \square

We thus have:

Theorem 4. *Given a graph G , and a factorizing permutation of G , it is possible to compute the modular decomposition tree of G in time $O(n + m)$.*

Algorithm 7. Computing all maximal right-modules given $s[v]$

```

S is a stack of vertices; t denotes the top of S.
for v from n to 2
  if s[v] exists
    L[v] ← v
    While t < s[v]
      L[t] ← v
      Pop the top of S
  else
    Push v on S
    
```

7 Conclusion

In the present work, we formalized two concepts about common intervals, namely generators and canonical representation, that proved to have important algorithmic implications. Indeed, the combinatorial properties of these objects, and in particular the different links between them, are central in the design and the analysis of the simple optimal algorithms for computing common intervals of permutations we presented. It is important to highlight that our algorithms are really “optimal” since they are based on very elementary manipulations of stacks and arrays. This is, we believe, a significant improvement over the existing algorithms that are based on intricate data structures, both in terms of ease of implementation and time efficiency, and in terms of understanding the underlying concepts [10,16].

Moreover, we showed how, transposed in the more general context of modular decomposition of graphs, our results have a similar impact and lead to a significant simplification of some existing algorithms. Indeed, modular decomposition algorithms are quite complex algorithms, but using the simple factorizing permutation algorithm of [9] and the right-modules identification algorithm of Section 6, a generator of the interval-modules can easily be computed in linear time; tools from Section 5 can then be used to compute the strong interval-modules, that also are the strong modules, and the PQ -tree, called *modular decomposition tree* in this context.

References

1. S. Bérard, A. Bergeron, and C. Chauve. Conserved structures in evolution scenarios. In *Comparative Genomics, RECOMB 2004 International Workshop*, vol. 3388 of *Lecture Notes in Comput. Sci.*, p. 1–15. Springer-Verlag, 2005.

2. A. Bergeron, C. Chauve, F. de Montgolfier, and M. Raffinot. Computing common intervals of K permutations, with applications to modular decomposition of graphs. LIAFA technical report 2005-006 available at http://www.liafa.jussieu.fr/web9/rapportrech/listrapport_fr.php?anscol=2005
3. S. Booth and G. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-trees algorithms. *J. Comput. Syst. Sci.*, 13:335–379. 1976.
4. B. M. Bui Xuan, M. Habib, and C. Paul, From Permutations to Graph Algorithms, LIRMM technical report RR-05021, 2005.
5. C. Capelle and M. Habib. Graph decompositions and factorizing permutations. in *Fifth Israel Symposium on Theory of Computing and Systems, ISTCS 1997*, p. 132–143. IEEE Computer Society, 1997.
6. A. Cournier and M. Habib. A new linear algorithm for modular decomposition. In *Trees in algebra and programming – CAAP’94, 19th International Colloquium*, vol. 787 of *Lecture Notes in Comput. Sci.*, p. 68–84. Springer-Verlag, 1994.
7. E. Dahlhaus, J. Gustedt, and R. M. McConnell. Efficient and practical algorithms for sequential modular decomposition. *J. Algorithms*, 41(2):360–387. 2001.
8. M. Figeac and J.-S. Varré. Sorting by reversals with common intervals. In *Algorithms in Bioinformatics, 4th International Workshop, WABI 2004*, vol. 3240 of *Lecture Notes in Comput. Sci.*, p. 26–37. Springer-Verlag, 2004.
9. M. Habib, F. de Montgolfier and C. Paul. A Simple Linear-Time Modular Decomposition Algorithm for Graphs, Using Order Extension In *Algorithm Theory – SWAT 2004, 9th Scandinavian Workshop on Algorithm Theory*, vol. 3111 of *Lecture Notes in Comput. Sci.*, p. 187–198. Springer-Verlag, 2004.
10. S. Heber and J. Stoye. Finding all common intervals of k permutations. In *Combinatorial Pattern Matching, 12th Annual Symposium, CPM 2001*, vol. 2089 of *Lecture Notes in Comput. Sci.*, p. 207–218. Springer-Verlag, 2001.
11. G.M. Landau, L. Parida and O. Weimann. Gene Proximity Analysis Across Whole Genomes via PQ Trees. 6th Combinatorial Pattern Matching Conference (CPM), 2005.
12. R. M. McConnell and F. de Montgolfier. Algebraic Operations on PQ-trees and Modular Decomposition Trees. WG’05, 31st International Workshop on Graph-Theoretic Concepts in Computer Science, 2005.
13. R. M. McConnell and J. Spinrad. Linear-time modular decomposition and efficient transitive orientation of comparability graphs. In *Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, p. 536–545. ACM/SIAM, 1994.
14. R. M. McConnell and J. Spinrad. Ordered vertex partitioning. *Discrete Mathematics & Theoretical Computer Science*, 4:45–60. 2000.
15. R. H. Möhring and F. J. Radermacher. Substitution decomposition for discrete structures and connections with combinatorial optimization. *Annals of Discrete Mathematics*, 19:257–356. 1984.
16. T. Uno and M. Yagiura. Fast algorithms to enumerate all common intervals of two permutations. *Algorithmica*, 26(2):290–309. 2000.

Greedy Routing in Tree-Decomposed Graphs

Pierre Fraigniaud*

CNRS, University of Paris-Sud

Abstract. We propose a new perspective on the small world phenomenon by considering arbitrary graphs augmented according to probabilistic distributions guided by tree-decompositions of the graphs. We show that, for any n -node graph G of treewidth $\leq k$, there exists a tree-decomposition-based distribution \mathcal{D} such that greedy routing in the augmented graph (G, \mathcal{D}) performs in $O(k \log^2 n)$ expected number of steps. We also prove that if G has chordality $\leq k$, then the tree-decomposition-based distribution \mathcal{D} insures that greedy routing in (G, \mathcal{D}) performs in $O((k + \log n) \log n)$ expected number of steps. In particular, for any n -node graph G of chordality $O(\log n)$ (e.g., chordal graphs), greedy routing in the augmented graph (G, \mathcal{D}) performs in $O(\log^2 n)$ expected number of steps.

1 Introduction

In his seminal work [23], Kleinberg gave a formal support to the “six degrees of separation” phenomenon, defined after the Milgram’s experiment [31], recently reproduced by Dodds, Muhamad, and Watts [13] (see also [1]). This experiment demonstrated that there are short chains of acquaintances between individuals, and that these chains can be discovered in a greedy manner. More precisely, given an arbitrary source person s (e.g., living in Wichita, KA), and an arbitrary target person t (e.g., living in Cambridge, MA), a letter can be transmitted from s to t via a chain of individuals related on a personal basis. The target is identified by its name, its professional occupation, and by the US state of its home town. The transmission rule is that the letter held by an intermediate person x is passed to the next person y who, as judged by x , is most likely to know the target among all persons x knows on a first-name basis. Milgram’s experiment conclusion is often summarized as the six degrees of separation phenomenon because, for chains that reached the target, the number of intermediate persons between the source and the target ranged from 2 to 10, with a median of 5.

Expanding on [35], Kleinberg modeled Milgram’s experiment as follows (cf. [23,24]). Let \mathcal{M} be the set of all 2-dimensional square meshes (i.e., the $n \times n$ grids, for $n \geq 1$). For $M \in \mathcal{M}$, every node x of M is given an additional directed link pointing to some node y . The head y of the added link (x, y) ,

* The author received additional supports from the project “PairAPair” of the ACI Masses de Données, from the project “Fragile” of the ACI Sécurité Informatique, from the project “Grand Large” of INRIA, and from ShmoogLe.

called the *long-range contact* of x , is chosen according to the 2-harmonic distribution \mathcal{H} , i.e., the probability that x chooses y , $y \neq x$, as long-range contact, is $\text{Prob}_x(y) = 1/(H_x \cdot \text{dist}^2(x, y))$ where $\text{dist}(x, y)$ denotes the Manhattan distance between x and y in M , and $H_x = \sum_{y \neq x} 1/\text{dist}^2(x, y)$ is a normalizing coefficient. The resulting graph is called an *augmented mesh*, and the set of graphs M augmented by \mathcal{H} is denoted by (M, \mathcal{H}) . Then, Kleinberg defined *greedy routing* in any graph of (M, \mathcal{H}) as the following process: Given a target node t , and a current node x , x selects among all its neighbors (including its long-range contact) the one that is closest to t in the mesh M (i.e., according to the Manhattan distance), and forwards to this neighbor. Kleinberg proved that greedy routing in the n -node mesh augmented with long-range links set according to the 2-harmonic distribution performs in $O(\log^2 n)$ expected number of steps.

In Kleinberg's model, the choice of the 2-harmonic distribution for the 2-dimensional meshes is crucial. Indeed, Kleinberg also proved that greedy routing in 2-dimensional meshes augmented with the k -harmonic distribution $\text{Prob}_x(y) = 1/(H_x^{(k)} \cdot \text{dist}^k(x, y))$ where $H_x^{(k)} = \sum_{y \neq x} 1/\text{dist}^k(x, y)$, performs poorly if $k \neq 2$, i.e., in $\Omega(n^\alpha)$ expected number of steps, for some $\alpha > 0$ that depends on k . Therefore, finding the right distribution for 2-dimensional meshes was far from being obvious, and there is no distribution \mathcal{D} for which greedy routing in (M, \mathcal{D}) is known to perform in $O(\text{polylog}(n))$ expected number of steps, but (distributions structurally equivalent to) the 2-harmonic distribution. More generally, the design of an appropriate distribution \mathcal{D} for an arbitrary given graph G seems to be uneasy. Formally, we raise the following question:

Problem 1. *For any n -node graph $G = (V, E)$, is there a distribution \mathcal{D} such that greedy routing in the augmented graph (G, \mathcal{D}) performs in $O(\text{polylog}(n))$ expected number of steps?*

By “greedy routing” in (G, \mathcal{D}) , it is meant the following process:

Definition 1. (*Greedy Routing.*) *For any target node $t \in V$, the current node $x \in V$ selects among all its neighbors (including its long-range contact chosen according to \mathcal{D}) the neighbor y that is closest to t in the underlying graph G , and forwards to y .*

Note that it is not sufficient to place a graph with small diameter on top of the underlying graph G for greedy routing to perform in a small number of steps. Indeed, greedy routing optimizes the choice of the current node's neighbor according to a distance measured in G , and not in the graph including the long-range links.

Beside its own theoretical interest as a natural generalization of Kleinberg's work on the mesh, solving Problem 1 would have a significant impact on our understanding of routing in social networks, as illustrated by Milgram's experiment. Indeed, although Kleinberg's model is a powerful tool for analyzing greedy routing strategies, there is no evidence that the network formed by social acquaintances looks like an augmented mesh. There are however some evidences that the social entities share a common knowledge about their relative distances,

based on their geographical positions, on their professional occupations, on their hobbies, or on any criteria available to the entities. This common knowledge could be modeled by a graph G . Then, the random events of life create connections between individuals who have, a priori, very little in common. This could be modeled by random links added on top of the underlying graph G . Therefore there is some support to the hypothesis that a social network can reasonably be modeled by an augmented graph (G, \mathcal{D}) . Still, this gives rise to several questions: what is the graph G ? What is the distribution \mathcal{D} ? Why the long-range links are structured according to some specific distribution rather than to another? By considering Problem 1, this paper is an attempt to solve these questions.

1.1 Our Results

First, we address Problem 1 in tree-decomposed graphs. Informally, the *treewidth* of a graph measures how far the graph is from a tree. Graphs of bounded treewidth form a large class of graphs, including trees, outer-planar graphs, series-parallel graphs, etc. In addition to their connection to the graph-minor theory (cf, e.g., [33]), they have a wide range of applications in graph searching [32] and routing [18,19]. They also play a central role in complexity and logic. In particular, it is known that several NP-hard problems can be solved in polynomial time if instances are restricted to graphs of bounded treewidth [3,6]. Actually, on graphs of treewidth at most k , where k is fixed, every decision or optimization problem expressible in monadic second-order logic has a linear algorithm [12]. We show that, for any n -node graph G of treewidth $\text{tw}(G)$, there exists a *tree-decomposition-based* distribution \mathcal{D} such that greedy routing in the augmented graph (G, \mathcal{D}) performs in

$$O(\text{tw}(G) \log^2 n) \tag{1}$$

expected number of steps. In particular, for graphs of bounded treewidth, there exists a tree-decomposition-based distribution such that greedy routing in the augmented graph performs in $O(\log^2 n)$ expected number of steps. This latter bound is close to optimal as it is known that, in the n -node directed ring, no distribution enables greedy routing to perform better than $\Omega(\log^2 n / \log \log n)$ expected number of steps [4]. We also give a constructive variant of our result. More precisely, given any n -node graph G , we show how to construct (in polynomial time) a distribution \mathcal{D} such that greedy routing in the augmented graph (G, \mathcal{D}) performs in $O(\text{tw}(G) \cdot \sqrt{\log \text{tw}(G)} \cdot \log^2 n)$ expected number of steps.

Social networks possess specific topological properties which strongly impact the performances of greedy routing. In particular nodes of social networks are often grouped in communities. This property motivated us to investigate greedy routing in graphs of bounded *chordality* (the chordality is the length of the longest chordless cycle). We prove that if G has chordality γ , then the tree-decomposition-based distribution \mathcal{D} insures that greedy routing in (G, \mathcal{D}) performs in

$$O((\gamma + \log n) \log n) \tag{2}$$

Table 1. Performances of (pure) greedy routing (with 1 long-range contact per node)

Underlying graph	Distribution	Expected #steps	References
d -dimensional meshes	d -harmonic	$O(\log^2 n)$	[23]
d -dimensional meshes ring	k -harmonic, $k \neq d$	$\Omega(n^\alpha)$, $\alpha > 0$	[23]
directed ring	1-harmonic	$\Omega(\log^2 n)$	[5]
d -dimensional meshes, $d > 1$	any	$\Omega(\log^2 n / \log \log n)$	[4]
moderate growth graphs	d -harmonic	$\Omega(\log^2 n)$	[20]
graphs of treewidth $\leq k$	1/ball-size	$O(\text{polylog}(n))$	[15]
graphs of chordality $\leq \gamma$	tree-decomposition-based	$O(k \log^2 n)$	[this paper]
	tree-decomposition-based	$O((\gamma + \log n) \log n)$	[this paper]

expected number of steps. In particular, for any n -node graph G of chordality $O(\log n)$ (e.g., chordal graphs), greedy routing in the augmented graph (G, \mathcal{D}) performs in $O(\log^2 n)$ expected number of steps, where \mathcal{D} is the tree-decomposition-based distribution. It is important to note that, as opposed to Equation 1, the performance of greedy routing in graphs of bounded chordality is independent from the treewidth of these graphs, although the treewidth of n -node chordal graphs can take any value between 1 and $n - 1$.

All known complexity results (including ours) relative to the performances of greedy routing in graphs augmented with one long-range contact per node are summarized in Table 1. This table does not list results related to variants of greedy routing, such as the ones mentioned in Section 1.2.

Finally, in the last part of this paper, we revisit our setting of the long-range links and argue (somewhat informally) that our tree-decomposition-based distribution is plausible in the context of social networks, i.e., a social network is well modeled by a graph G augmented with long-range links chosen according to a tree-decomposition-based distribution \mathcal{D} . In particular, as opposed to hierarchical models which define the hierarchical structure *a priori* (cf., e.g., [25,34]), the hierarchy of our model is inherited from the structure of the social network.

1.2 Related Works

Several authors expanded on [23,24]. In [5], it is shown that the $O(\log^2 n)$ upper bound of [23] is tight in the ring augmented with the 1-harmonic distribution, i.e., greedy routing performs in $\Omega(\log^2 n)$ expected number of steps. More generally, [4] shows that in the directed ring augmented with *any* distribution, greedy routing performs in $\Omega(\log^2 n / \log \log n)$ expected number of steps. In [26], a decentralized routing algorithm for augmented meshes is described. The routing visits $O(\log^2 n)$ nodes, and distributively discovers routes of expected length $O(\log n (\log \log n)^2)$ links using headers of size $O(\log^2 n)$ bits. Neighbor-of-neighbor greedy routing defined in [11,27] performs in $O(\frac{1}{c \log c} \log^2 n)$ expected number of steps, with c long-range contacts per node. The non-oblivious routing protocol described in [28] performs in $O(\log^{1+1/d} n)$ expected number of steps in the d -dimensional mesh. The oblivious Indirect-greedy routing protocol described in [20] performs in $O(\log^{1+1/d} n)$ expected number of steps in the

d -dimensional mesh. [20] also shows that the $O(\log^2 n)$ upper bound of [23] is tight in the d -dimensional mesh augmented with the d -harmonic distribution, for any $d \geq 1$. [15] generalizes Kleinberg's result to the family of "moderate growth graphs", namely the graphs such that, roughly speaking, the size of the ball of radius r centered at any node x is equal to $r^{d_x(r)}$ where d_x is a function that is \mathcal{C}^1 and whose derivative is in $O(1/(r \log r))$. Finally, [29] recently proposed several constructions of small worlds, based on adding links with probability proportional to the inverse distance. These constructions generalize both [23] and [25]. Finally, in [25,30,34], the authors consider a hierarchical model that will be discussed in more detail in Section 5.

2 Definitions and Notations

Performances of Greedy Routing. Let $G = (V, E)$ be a connected graph with n nodes, and let $\mathcal{D} = \{\text{Prob}_x, x \in V\}$, where, for any $x \in V$, Prob_x is a probability distribution on $V \setminus \{x\}$. An augmentation of G according to \mathcal{D} is a graph obtained from G by adding at every node $x \in V$ one directed edge (x, y) where y is chosen with probability $\text{Prob}_x(y)$. For every ordered pair $(s, t) \in V \times V$, let $\mathbf{X}_{s,t}$ be the random variable specifying the number of steps required by greedy routing to go from s to t in the augmented graph. Let $\text{EX}_{s,t}$ be the expected value of $\mathbf{X}_{s,t}$. Kleinberg proved that, if G is a 2-dimensional square mesh, and \mathcal{D} is the 2-harmonic distribution, then $\text{EX}_{s,t} = O(\log^2 n)$. Greedy routing insures that, at every step, one gets closer in G to the target, i.e.:

Fact 1. *If x is the current node, and greedy routing forwards to y , then $\text{dist}_G(y, t) < \text{dist}_G(x, t)$ where $\text{dist}_G(\cdot)$ is the distance function in the underlying graph G .*

As a consequence, greedy routing has no loop, and it requires at most $\text{dist}_G(s, t)$ steps to go from s to t . In particular, in graphs with polylogarithmic diameter, there is no need to add long-range contacts for greedy routing to perform in polylogarithmic number of steps.

Treewidth. A *tree-decomposition* of graph G is a pair (T, X) where T is a tree, and $X = \{X_v, v \in V(T)\}$ is a collection of subsets of $V(G)$ satisfying the following three conditions:

- **C1:** $V(G) = \cup_{v \in V(T)} X_v$;
- **C2:** For any edge e of G , there is a set X_v such that both end-points of e are in X_v ;
- **C3:** For any triple u, v, w of nodes in $V(T)$, if v is on the path from u to w in T , then $X_u \cap X_w \subseteq X_v$.

Condition **C3** can be rephrased as: for any node x of G , $\{v \in V(T) \mid x \in X_v\}$ is a subtree of T . The sets X_v s are called *bags*. The *width*, $\omega(T, X)$, of a tree-decomposition (T, X) is defined as $\max_{v \in V(T)} |X_v| - 1$, i.e., the width of (T, X) is roughly the maximum size of its bags. The *treewidth* $\text{tw}(G)$ is defined as

$\min \omega(T, X)$ where the minimum is taken over all tree-decompositions (T, X) of G . For instance, trees have treewidth 1, cycles have treewidth 2, and n -node cliques have treewidth $n - 1$.

Any bag of a tree-decomposition is a *separator* of the graph, that is a vertex set whose removal disconnects the graph. In fact, a tree-decomposition can be obtained by recursively separating the graph. (This is essentially the way treewidth is $O(\log n)$ -approximated in [7,9].) We will intensively use this fact throughout all the paper. Let (T, X) be a tree-decomposition of a graph G . Let x and y be two nodes of G , and let b be a bag of T containing neither x nor y , i.e., $b \cap \{x, y\} = \emptyset$. Removing b from T results in a forest of $k \geq 1$ trees T_1, \dots, T_k . Since $b \cap \{x, y\} = \emptyset$, C1 and C3 imply that there is a unique i (resp., j) in $\{1, \dots, k\}$ such that x (resp., y) belongs to some bag(s) of T_i (resp., T_j). Assume that $i \neq j$, then the following is folklore:

Fact 2. *The bag b is an (x, y) -separator in G (i.e., all paths from x to y in G go through some node(s) in b).*

3 Tree-Decomposition-Based Distribution

This section is dedicated to the definition of the tree-decomposition-based distribution of the long-range contacts, and to the proof of the following result:

Theorem 1. *For any connected n -node graph G of treewidth $\leq k$, there is a distribution \mathcal{D} such that, for any source-destination pair (s, t) , $\mathbf{EX}_{s,t} = O(k \log^2 n)$.*

Corollary 1. *For any connected n -node graph G of bounded treewidth, there is a distribution \mathcal{D} such that greedy routing in the augmented graph (G, \mathcal{D}) performs in $O(\log^2 n)$ expected number of steps.*

Corollary 1 is close to optimal since [4] shows that greedy routing performs in $\Omega(\log^2 n / \log \log n)$ expected number of steps in the directed ring augmented with *any* distribution. In both Theorem 1 and Corollary 1, the distribution \mathcal{D} is a tree-decomposition-based distribution, as defined in the proof bellow.

Proof of Theorem 1. For any $k \geq 2$, let \mathcal{G}_k be the class of connected graphs of treewidth $< k$. Let $G \in \mathcal{G}_k$ be a graph of n nodes, and let T be a tree-decomposition of G , of width $< k$. We can choose T with at most n bags (cf., e.g., Theorem 4.8 and Proposition 4.16 in [22]). In order to describe the distribution \mathcal{D} , we describe the \mathcal{D}_x s, i.e., we describe the setting of the long-range contact of every node x in G . Recall that a *centroid* of an r -node tree is a node whose removal from the tree results in a forest with at most $r/2$ nodes in each subtree. A tree has either one or two centroids, and if a tree has two centroids, then they are neighbors.

Let c be a centroid of T . For every node $x \in V$, let us denote by \hat{x} the bag containing x that is closest to c in T . Note that, by C3 of the treewidth definition, \hat{x} is uniquely defined. We set $c_x^{(0)} = c$, and define $T_x^{(1)}$ as the subtree of $T \setminus \{c_x^{(0)}\}$ containing \hat{x} . Then, let $c_x^{(1)}$ be a centroid of $T_x^{(1)}$, and let $T_x^{(2)}$ be the subtree of $T_x^{(1)} \setminus \{c_x^{(1)}\}$ containing \hat{x} . And so on. One constructs in this way two sequences

$$(T_x^{(0)}, T_x^{(1)}, \dots, T_x^{(q_x)}) \text{ and } (c_x^{(0)}, c_x^{(1)}, \dots, c_x^{(q_x)})$$

where:

1. $T_x^{(0)} = T$;
2. $c_x^{(i)}$ is the centroid of $T_x^{(i)}$ closest to c in T ;
3. $T_x^{(i+1)}$ is the subtree of $T_x^{(i)} \setminus \{c_x^{(i)}\}$ containing \hat{x} ;
4. $c_x^{(q_x)} = \hat{x}$.

Note that since $|T| \leq n$, and $|T_x^{(i+1)}| \leq |T_x^{(i)}|/2$, we get that both sequences are of length $q_x + 1 \leq \log n$. The result hereafter directly follows from the definition of these two sequences.

Lemma 1. *For any two nodes u and v , and for any index i , if $\hat{v} \in T_u^{(i)}$, then $c_v^{(j)} = c_u^{(j)}$ for $j = 0, \dots, i$, and $c_v^{(j)} \in T_u^{(i)}$ for $j = i, \dots, q_v$.*

Tree-Decomposition-Based Distribution \mathcal{D} . Node x picks its long-range contact as follows:

- First x selects an index $i \in \{0, \dots, q_x\}$ with $\text{Prob}_x(i) = 1/(q_x + 1)$;
- Next, x selects a node y chosen uniformly at random in the bag $c_x^{(i)}$.

Node y is the long-range contact of x .

We show that with this setting of the long-range contacts, for any source node s , and any target node t , $\text{EX}_{s,t} = O(k \log^2 n)$. Let G^+ be an instance of the graph G augmented with the long-range contacts set as above. Note that G^+ is directed since the edge from a node to its long-range contact is directed (edges of the underlying graph G remain undirected). Let $t \in V(G)$, and let $i \in \{1, \dots, q_t\}$. Let

$$U_i = \{v \in V(G) \mid \hat{v} \in T_t^{(i)}\}.$$

Lemma 2. *The node-set $\cup_{j=0}^{i-1} c_t^{(j)} \subseteq V(G)$ separates U_i and $V(G) \setminus U_i$ in G^+ , i.e., any path in G^+ from a node in U_i to a node in $V(G) \setminus U_i$ goes through a node in $\cup_{j=0}^{i-1} c_t^{(j)}$.*

Proof. Let P be a path from a node in U_i to a node in $V(G) \setminus U_i$. Let $e = (v, w)$ be an edge of P from $v \in U_i$ to $w \in V(G) \setminus U_i$. If $v \in c_t^{(j)}$ for some $j \in \{0, \dots, i-1\}$, then we are done. Thus assume $v \notin \cup_{j=0}^{i-1} c_t^{(j)}$. Since $w \notin U_i$, we have $\hat{w} \notin T_t^{(i)}$. We consider separately the case where e is an edge of G , from the case where e is a long-range link.

If $e \in E(G)$, then let b be a bag containing both v and w (this bag exists from C2). On the one hand, by C3, w belongs to all bags on the path in T from b to \hat{w} . On the other hand, we have b further from c than \hat{v} , i.e., \hat{v} is on the path from b to c in T . Now, by construction of the sequence $\{c_t^{(j)}, 0 \leq j \leq q_t\}$, the neighborhood of $T_t^{(i)}$ in T (i.e., the set of bags not in $T_t^{(i)}$ but adjacent to some bag in $T_t^{(i)}$) is included in $\cup_{j=0}^{i-1} c_t^{(j)}$. Hence $b \in T_t^{(i)}$ since otherwise v would belong to some bag

of the neighborhood of $T_t^{(i)}$ (by C3), which would imply $v \in \cup_{j=0}^{i-1} c_t^{(j)}$. Since \widehat{w} is closer to c than b , but $\widehat{w} \notin T_t^{(i)}$, there is some $c_t^{(j)}$, $j \in \{0, \dots, i-1\}$ on the path from b to \widehat{w} in T . Therefore, $w \in \cup_{j=0}^{i-1} c_t^{(j)}$, proving Lemma 2.

If $e \notin E(G)$, then w is the long-range contact of v . By the setting of the long-range contacts, $w \in \cup_{j=0}^{qv} c_v^{(j)}$. By Lemma 1, all bags $c_v^{(j)}$ for $j \geq i$ are nodes of $T_t^{(i)}$. If node w belongs to some bag b of $T_t^{(i)}$, then, as in the case $e \in E(G)$, combining C3 with the fact that $\widehat{w} \notin T_t^{(i)}$ yields $w \in \cup_{j=0}^{i-1} c_t^{(j)}$, and we are done. Thus assume that w does not belong to any bag of $T_t^{(i)}$. Therefore $w \in \cup_{j=0}^{i-1} c_v^{(j)}$. From Lemma 1, since $\widehat{v} \in T_t^{(i)}$, $c_v^{(j)} = c_t^{(j)}$ for $j = 0, \dots, i$. Therefore $w \in \cup_{j=0}^{i-1} c_t^{(j)}$, which completes the proof of Lemma 2. \diamond

Let $(T_t^{(0)}, T_t^{(1)}, \dots, T_t^{(qt)})$ and $(c_t^{(0)}, c_t^{(1)}, \dots, c_t^{(qt)})$ be the sequences of subtrees and centroids corresponding to the target t . Let x be the current node. (Initially, x is the source node s .) Let i be the largest index such that $\widehat{x} \in T_t^{(i)}$. Let x_0, x_1, \dots, x_r be the sequence of nodes visited by greedy routing from $x = x_0$ until either it reaches a node x_r with $\widehat{x}_r \notin T_t^{(i)}$, or it reaches t .

Lemma 3. *Let $y \in \cup_{j=0}^i c_t^{(j)}$. For every $\ell = 0, \dots, r-1$, the probability that y is the long-range contact of x_ℓ is at least $1/(k \log n)$. The probability that the long-range contact of x is in $c_t^{(j)}$ is at least $1/\log n$ for every $j = 0, \dots, i$.*

Proof. We have $\widehat{x}_\ell \in T_t^{(i)}$ for every $\ell < r$. Thus, from Lemma 1, for any $\ell < r$, $c_{x_\ell}^{(j)} = c_t^{(j)}$ for $j = 0, \dots, i$. Therefore, every node x_ℓ , $\ell < r$, has its long-range contact in a specific bag $c_t^{(j)}$, $0 \leq j \leq i$, with probability $1/(1 + q_{x_\ell})$. A node $y \in c_t^{(j)}$ for some $j \leq i$ is the long-range contact of x_ℓ with probability at least $1/(|c_t^{(j)}|(1 + q_{x_\ell}))$. Since $|c_t^{(j)}| \leq k$, and $1 + q_{x_\ell} \leq \log n$, Lemma 3 follows. \diamond

Lemma 4. *The path from s to t constructed by greedy routing does not visit any bag $c_t^{(0)}, \dots, c_t^{(qt)}$ more than k times.*

Proof. Since T has width $< k$, no bag contains more than k nodes. Thus, from Fact 1, no bag can be visited by greedy routing more than k times on the way from s to t . This is true in particular for bags $c_t^{(0)}, \dots, c_t^{(qt)}$. \diamond

Finally, we will make use of the following simple result. Let $(X_i)_{i \geq 1}$ be a sequence of independent random variables in $\{0, 1, \dots, N\}$ with

$$\begin{aligned} \text{Prob}(\{X_i = j\}) &= p/N \quad \text{if } j \in \{1, \dots, N\}; \\ \text{Prob}(\{X_i = 0\}) &= 1 - p; \end{aligned}$$

for some $0 < p < 1$. We consider the following iterative process. Let $S_0 = \{b_1, \dots, b_N\}$ be a set of N non negative integers. After the i th trial, if $X_i > 0$, then all integers $b_j \geq b_{X_i}$ in the current set are removed, i.e., $S_i = S_{i-1} \setminus \{b_j \mid b_j \geq b_{X_i}\}$. Let Y be the random variable specifying the number of trials i until S_i becomes empty.

Lemma 5. $EY \leq N/p$.

Proof. The set becomes empty after the first trial i such that $X_i = j$, where $b_j = \min_{\ell} b_{\ell}$. This occurs with probability at least p/N . \diamond

Let P be the path followed by greedy routing from s to t in G^+ . We decompose P into a sequence of subpaths $P_0 P_1 P_2 \dots P_{q_t}$ where the first node of P_0 is s , the last node of P_{q_t} is t , and, for every $i = 0, 1, \dots, q_t$, $P_i \subseteq T_t^{(i)}$ and is minimal for that property. More explicitly, let $P = x_0, \dots, x_r$ with $x_0 = s$ and $x_r = t$. For $i = 0, \dots, q_t$, let a_i be the smallest index such that, for every $j \geq a_i$, $\hat{x}_j \in T_t^{(i)}$. In particular, $a_0 = 0$ since $\hat{s} \in T = T_t^{(0)}$ and every node of G belongs to some bag of T . Similarly, $a_{q_t} \leq r$ since greedy routing eventually reaches $t \in c_t^{(q_t)} \in T_t^{(q_t)}$. We define P_i as the path in G which starts at x_{a_i} , and ends at $x_{a_{i+1}-1}$, but P_{q_t} which ends at t . (If $a_{i+1} = a_i$, then P_i is the empty path.) We have:

$$|P| = \sum_{i=0}^{q_t} |P_i|. \tag{3}$$

Let $i \in \{0, 1, \dots, q_t\}$, and consider P_i . By definition, while traveling along P_i , greedy routing never goes out of $T_t^{(i)}$. Thus, from Lemma 2, it does not visit nodes x such that $\hat{x} \in \cup_{j=0}^{i-1} c_t^{(j)}$. P_i may however go in and out of $T_t^{(i+1)}$. From Lemma 2, the only way P_i goes in and out of $T_t^{(i+1)}$ is through $c_t^{(i)}$. From Lemma 3, for each node x of P_i , the long-range contact y of x is in $c_t^{(i)}$ with probability at least $1/\log n$. Assume success, i.e., $y \in c_t^{(i)}$. From Fact 1, no node z with $\text{dist}_G(z, t) > \text{dist}_G(y, t)$ will be ever visited by greedy routing after x . In particular, no node $z \in c_t^{(i)}$ with $\text{dist}_G(z, t) > \text{dist}_G(y, t)$ will be ever visited by greedy routing after x . In the same spirit as for Lemma 5, we just say that those nodes y and $z \in c_t^{(i)}$ are “removed”. We are in the situation of Lemma 5 with $p \geq 1/\log n$, and $N = |c_t^{(i)}| \leq k$. Thus, after an expected number of at most $O(k \log n)$ trials, all nodes in $c_t^{(i)}$ are removed. Therefore, from Lemma 4, after this expected amount of trials, no nodes of $c_t^{(i)}$ will be ever visited by greedy routing. Hence, once in P_i , the path P enters P_{i+1} after at most $O(k \log n)$ expected number of steps. In other words, the expected length of P_i is $O(k \log n)$. Therefore, from Eq. 3, the expected length of the path P is at most $O(q_t k \log n) \leq O(k \log^2 n)$ which completes the proof of Theorem 1. \square

Theorem 1 is an existential result. Nevertheless, a combination of this theorem with known results from the literature allows us to explicitly construct a long-range contact distribution for any graph G . This is however to the price of a $\log \text{tw}(G)$ factor in the performances of greedy routing. More precisely, a tree-decomposition T of any graph G , with width $\leq O(\text{tw}(G) \sqrt{\log \text{tw}(G)})$, can be computed in polynomial time (see [17]). Once this is done, since the distribution \mathcal{D} in Theorem 1 can obviously be computed in polynomial time, we get:

Corollary 2. *There is an polynomial time algorithm that, for any connected n -node graph G of treewidth $\leq k$, computes a distribution \mathcal{D} such that greedy*

routing in the augmented graph (G, \mathcal{D}) performs in $O(k\sqrt{\log k} \log^2 n)$ expected number of steps.

4 Greedy Routing in Augmented Chordal Graphs

Entities in social networks are known to be often grouped in communities [35]. This motivated us to study greedy routing using tree-decomposition-based long-range contact distributions in graphs of bounded *chordality*. Formally, the chordality of a graph G is the maximum length of a chordless cycle in G . In particular, a graph of chordality 3 is a *chordal* graph.

Theorem 2. *For any connected graph G of n nodes and chordality γ , there is a tree-decomposition-based distribution \mathcal{D} enabling greedy routing in (G, \mathcal{D}) to perform in $O((\gamma + \log n) \log n)$ expected number of steps, for any source-destination pair.*

The proof uses the same arguments as in the proof of Theorem 1, combined with the following two additional facts.

1. For any n -node connected graph of chordality γ , there is a tree-decomposition with at most n bags such that two nodes in the same bag are at distance at most $\gamma/2$ [21] (see also [8]).
2. Let $x_0 = s, x_1, x_2, \dots, x_r = t$ be the path followed by greedy routing from s to t ; If $\text{dist}_G(x_i, x_j) \leq d$ then $|i - j| \leq d$.

Note that the result of Theorem 2 is independent from the treewidth of the graph. It has an important consequence:

Corollary 3. *For any n -node graph G of chordality $O(\log n)$ (in particular for any chordal graph), there is a tree-decomposition-based distribution \mathcal{D} such that greedy routing in the augmented graph (G, \mathcal{D}) performs in $O(\log^2 n)$ expected number of steps.*

5 Discussion

Individuals can be grouped in large families. For instance: Africans, Americans, Europeans, etc., or artists, scientists, farmers, etc. This can be done recursively. For instance, Europeans can be grouped according to their countries of leaving, while scientists can be grouped according to their scientific domains. And so on. This clustered and hierarchical structure of the social networks was already pointed out by several authors (cf., e.g., [1,2,10,14,16,30,34,35]). The model in [25] was the first model specified to capture this hierarchy (see also [29]). However, this model assumes that the hierarchy is induced by a specific structured graph, defined a priori. (More precisely, in the model, nodes are leaves of a complete b -ary tree, and the lower is the lowest common ancestor of two nodes, the more likely these two nodes are to be connected by a long-range link.) Moreover

the model in [25] reflects one type of hierarchy only (e.g., arts/music/opera) whereas social entities belong to several *interleaved* hierarchies such as those based on the place of living, the professional activity, the recreative activity, etc.

In contrast, a tree-decomposition of the “natural” acquaintances (i.e., the acquaintances described by the graph G) determines a hierarchy that is inherited from these acquaintances, and not specified a priori. This hierarchy is the expression of *all* the underlying interleaved hierarchies. (This “general” hierarchy could be interpreted a posteriori in the same way one interprets the result of a Principal Components Analysis, but this is beyond the scope of this paper.) The long-range contacts enable jumping across this hierarchy, and one may jump upwards as well as downwards across the hierarchy. In addition, the hierarchy is viewed differently from each node. In particular, nodes that are placed far apart in the tree-decomposition have very different views of the hierarchy.

References

1. L. Adamic, and E. Adar. How To Search a Social Network. Social Networks. Preprint submitted to Social Networks, 2004.
2. M. Aldenderfer and R. Blashfield. Cluster Analysis. Quantitative Applications in the Social Sciences, Vol. 44, SAGE Publications, London, 1984.
3. S. Arnborg, B. Courcelle, A. Proskurowski, and D. Seese. An algebraic theory of graph reduction. Journal of the ACM 40(5), pages 1134–1164, 1993.
4. J. Aspnes, Z. Diamadi, and G. Shah. Fault-Tolerant Routing in Peer-to-Peer Systems. In 21st ACM Symp. on Principles of Distributed Computing (PODC), pages 223–232, 2002.
5. L. Barrière, P. Fraigniaud, E. Kranakis, and D. Krizanc. Efficient Routing in Networks with Long Range Contacts. In 15th International Symposium on Distributed Computing (DISC), LNCS 2180, pages 270–284, Springer, 2001.
6. H. Bodlaender. Treewidth: algorithmic techniques and results. In 22nd Symp. on Mathematical foundations of computer science (MFCS), LNCS 1295, pages 19–36, Springer, 1997.
7. H. Bodlaender, J. Gilbert, H. Hafsteinsson, and T. Kloks. Approximating Treewidth, Pathwidth, Frontsize and Shortest Elimination Trees. Journal of Algorithms 18, pages 238–255, 1995.
8. H. Bodlaender and D. Thilikos. Treewidth for Graphs with Small Chordality. Discrete Applied Mathematics 79(1-3):45-61, 1997.
9. V. Bouchitté, D. Kratsch, H. Müller, and I. Todinca. On treewidth approximations. Discrete Applied Mathematics 136, pages 183–196, 2004.
10. A. Clauset, M. Newman, and C. Moore. Finding community structure in very large networks. Phys. Rev. E 70, 066111 (2004).
11. D. Coppersmith, D. Gamarnik, and M. Sviridenko. The Diameter of a Long-Range Percolation Graph. Random Structures and Algorithms 21(1):1–13, 2002.
12. B. Courcelle, J. Makowsky, and U. Rotics. Linear-time solvable optimization problems on graphs of bounded cliquewidth. In 24th Workshop on Graph-Theoretic Concepts in Computer Science (WG), LNCS 1517, pages 1–16, Springer, 1998.
13. P. Dodds, R. Muhamad, and D. Watts. An Experimental Study of Search in Global Social Networks. Science 301:827-829, 2003.

14. L. Donetti and M. Muñoz. Detecting Network Communities: A New Systematic and Efficient Algorithm. *J. of Statistical Mechanics: Theory and Experiment*, P10012, 2004.
15. P. Duchon, N. Hanusse, E. Lebhar, and N. Schabanel. Could any graph be turned into a small world? Technical Report LIP RR-2004-61, ENS-Lyon, Dec 2004.
16. B. Everitt, S. Landau and M. Leese. *Cluster Analysis*. Arnold, London, 4th edition, 2001.
17. U. Feige, M. Hajiaghayi, and J. Lee. Improved approximation algorithms for minimum-weight vertex separators. In *37th ACM Symposium on Theory of Computing (STOC)*, 2005.
18. P. Fraigniaud and C. Gavoille. Routing in trees. In *29th International Colloquium on Automata, Languages and Programming (ICALP)*, LNCS 2076, pages 757–772, Springer, 2001.
19. P. Fraigniaud and C. Gavoille. End-to-end routing. In *17th Symposium on Distributed Computing (DISC)*, LNCS 2848, pages 211–223, Springer, 2003.
20. P. Fraigniaud, C. Gavoille, and C. Paul. Eclecticism shrinks even small worlds. In *23rd ACM Symp. on Principles of Distributed Computing (PODC)*, pages 169–178, 2004.
21. C. Gavoille, M. Katz, N. Katz, C. Paul, and D. Peleg. Approximate Distance Labeling Schemes. In *9th European Symposium on Algorithms (ESA)*, LNCS 2161, pages 476–487, Springer, 2001.
22. M. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New-York, 1980.
23. J. Kleinberg. The Small-World Phenomenon: An Algorithmic Perspective. In *32nd ACM Symp. on Theory of Computing (STOC)*, pages 163–170, 2000.
24. J. Kleinberg. Navigation in a Small-World. *Nature* 406:845, 2000.
25. J. Kleinberg. Small-World Phenomena and the Dynamics of Information. In *15th Neural Information Processing Systems (NIPS)*, 2001.
26. E. Lebhar and N. Schabanel. Searching for optimal paths in long-range contact networks. In *31st International Colloquium on Automata, Languages and Programming (ICALP)*, LNCS 3142, pages 894–905, Springer, 2004.
27. G. Manku, M. Naor, and U. Wieder. Know Thy Neighbor’s Neighbor: The Power of Lookahead in Randomized P2P Networks. In *36th ACM Symp. on Theory of Computing (STOC)*, 2004.
28. C. Martel and V. Nguyen. Analyzing Kleinberg’s (and other) Small-world Models. In *23rd ACM Symp. on Principles of Distributed Computing (PODC)*, pages 178–187, 2004.
29. C. Martel and V. Nguyen. Analyzing and Characterizing Small-World Graphs. In *16th ACM Symp. on Discrete Algorithms (SODA)*, 2005.
30. F. Menczer. Growing and Navigating the Small World Web by Local Content. *Proc. Natl. Acad. Sci. USA* 99(22):14014–14019, 2002.
31. S. Milgram. The Small-World Problem. *Psychology Today*, 60–67, 1967.
32. N. Megiddo, S. Hakimi, M. Garey, D. Johnson and C. Papadimitriou. The complexity of searching a graph. *Journal of the ACM* 35(1), pages 18–44, 1988.
33. N. Robertson and P. D. Seymour. Graph minors II, Algorithmic Aspects of Tree-Width. *Journal of Algorithms* 7, pages 309–322, 1986.
34. D. J. Watts, P. S. Dodds, M. E. J. Newman. Identity and Search in Social Networks. *Science* 296:1302–1305, 2002.
35. D. Watts and S. Strogatz. Collective Dynamics of Small-World Networks. *Nature* 393:440–442, 1998.

Making Chord Robust to Byzantine Attacks

Amos Fiat¹, Jared Saia², and Maxwell Young²

¹ Department of Computer Science,
Tel Aviv University, Tel Aviv, Israel
fiat@math.tau.ac.il

² Department of Computer Science, University of New Mexico,
Albuquerque, NM 87131-1386
{saia, young}@cs.unm.edu

Abstract. Chord is a distributed hash table (DHT) that requires only $O(\log n)$ links per node and performs searches with latency and message cost $O(\log n)$, where n is the number of peers in the network. Chord assumes all nodes behave according to protocol. We give a variant of Chord which is robust with high probability for any time period during which: 1) there are always at least z total peers in the network for some integer z ; 2) there are never more than $(1/4 - \epsilon)z$ Byzantine peers in the network for a fixed $\epsilon > 0$; and 3) the number of peer insertion and deletion events is no more than z^k for some tunable parameter k . We assume there is an adversary controlling the Byzantine peers and that the IP-addresses of all the Byzantine peers and the locations where they join the network are carefully selected by this adversary. Our notion of robustness is rather strong in that we not only guarantee that searches can be performed but also that we can enforce any set of “proper behavior” such as contributing new material, etc. In comparison to Chord, the resources required by this new variant are only a polylogarithmic factor greater in communication, messaging, and linking costs.

1 Introduction

A distributed hash table (DHT) is a structured peer-to-peer network which provides for scalable storage and lookup of data items (see e.g. [21,26,28]). Because peer-to-peer networks have little to no admission control, there has been significant effort in designing DHT’s which are robust to *Byzantine* faults. When a peer suffers a Byzantine fault it is assumed to be controlled by an omniscient adversary who uses that peer to try to disrupt the network.

In this paper, we consider the *Byzantine join attack*. Under this attack, a stream of Byzantine peers join and leave the network over a time period during which: 1) there are always at least z total peers in the network for some integer z ; 2) there are never more than $(1/4 - \epsilon)z$ Byzantine peers in the network for a fixed $\epsilon > 0$; and 3) the number of peer insertion and deletion events is no more than z^k for some tunable parameter k . We assume an adversary controls the stream of Byzantine peers joining the network and that this adversary carefully

chooses the IP-addresses of these Byzantine peers¹ and where they join the network in order to place them at critical locations in the network. We assume that all direct links in the overlay network are private communication channels. However, the adversary is otherwise computationally unbounded and omniscient i.e. it possesses full knowledge of the network topology, protocols, where data is stored, etc.

1.1 Our Contributions

In this paper, we describe a variant of Chord, *S-Chord*, which is robust to the Byzantine join attack. We define a *z-good* interval to be a time interval during which: 1) there are always at least z total peers in the network for some integer z ; 2) there are never more than $(1/4 - \epsilon)z$ Byzantine peers in the network for a fixed $\epsilon > 0$; and 3) the number of peer insertion and deletion events is no more than z^k for some tunable parameter k . Theorem 1 states our main result.

Theorem 1. *During any z -good interval, the following properties hold for S-Chord with high probability (specifically with probability of error polynomially small in z)*

- All functionality of Chord is preserved.
- We can enforce a rule-set for all peers in the network.
- For n peers in the network, the resource costs are as follows:
 - $O(\log n)$ latency and expected $\Theta(\log^2 n)$ messages sent per lookup operation.
 - $\Theta(\log n)$ latency and $\Theta(\log^3 n)$ messages sent per peer join operation.
 - $O(\log^2 n)$ links stored at each peer.

In addition to being robust to the Byzantine join attack, S-Chord also has the following properties.

- S-Chord can enforce a set of rules describing “proper behavior” such as: “For every 20 search that a peer issues, that peer must service one search request”. In particular, the consequences of not obeying the rules will be disconnection from the network. To the best of our knowledge, S-Chord is the first peer-to-peer network with this property.
- S-Chord is based on Chord and thus inherits many of Chord’s good properties. Moreover, we feel that the general techniques used in this paper can be applied to a wide-range of other DHT’s.
- S-Chord requires $\Theta(\log^2)$ messages for lookups in expectation. Previous DHT’s which are robust to the random Byzantine attack require $\Theta(\log^3 n)$ messages.

¹ i.e. by spoofing.

1.2 Related Work

Recent years have witnessed the advent of large scale real-world peer-to-peer applications such as Gnutella, Napster, Kazaa, Morpheus, BitTorrent, and many others. In addition, several distributed hash tables (DHTs) have been introduced which are provably robust to random peer deletions (i.e. fail-stop faults) [1,13,16,21,22,26,28].

We are aware of several results that deal with the more challenging problem of designing DHTs which are robust to Byzantine faults. One class of DHTs are robust to the *random Byzantine attack*. This is an instantaneous attack during which each peer in the network suffers a Byzantine fault independently at random with constant probability (less than $1/2$). Fiat and Saia describe a DHT which uses expander graphs and a butterfly network to achieve robustness to this attack [11]. This result was extended to be fully dynamic in [23]. Naor and Wieder describe a much simpler DHT which is robust to the random Byzantine attack and is also fully dynamic [19]. Hildrum and Kubiatowicz describe how to modify two popular DHTs, Pastry [22] and Tapestry [28], in order to make them robust to the random Byzantine attack [14]. Their modified DHTs are fully dynamic.² In all three of these results, lookups have $\Theta(\log n)$ latency and require $\Theta(\log^3 n)$ messages. We note that S-Chord is also robust to the random Byzantine attack.

Scheideler and Awerbuch [3,2] describe protocols for implementing a robust distributed naming service. Under their scheme, each node must re-inject itself into the system after a certain number of time steps and data must be continually published to remain in the system. Their system also assumes the existence of “bootstrap peers” which are a set of peers that 1) always remain in the system, 2) are all good, and 3) are known by joining peers. These assumptions are reasonable for a distributed name service application; however, they are problematic when trying to design a widely-applicable distributed hash table. Recent work by Scheideler in [24] demonstrates how to spread Byzantine peers via rotations during peer joins. This work focuses only on one aspect of the join protocol for a peer-to-peer system.

S-Chord makes use of secure multiparty computation in order to choose random IDs for joining peers by consensus. There is a significant body of work in the area of secure multiparty computation (see e.g. [4,5,6,8,12,15,20,25,27]). We also make use of Scheideler’s result [24] for spreading Byzantine peers and a result by King and Saia [17] for choosing a peer uniformly at random from the set of all peers in a DHT.

2 Overview

2.1 Chord

We now briefly describe Chord [26].³ For convenience, we will assume that the “key space” of Chord is scaled so it is in the range $(0, 1]$ and will think of Chord

² We emphasize here that S-Chord is also fully-dynamic.

³ For ease of exposition, our description will defer slightly from that of [26], but will not be fundamentally different.

as a circle with unit circumference, which we will call the *unit circle*. All of the peers in Chord have identifiers (or IDs for short) which are points on the unit circle that we call *peer points*. Chord provides one basic operation: *successor()*. For a point k on the unit circle, *successor*(k) returns the peer, p , whose peer point minimizes the clockwise distance between k and p . Typically, k represents a key for some data item and *successor*(k) is the peer responsible for storing that data item. Thus, the *successor()* operation provides for easy storage and lookups of data items.

We now briefly sketch how Chord implements the operation *successor()*. We assume that all peers in the network know some number m which is always greater than the number of peers in the network⁴. For a point p on the unit circle and integer i between 0 and $\log m - 1$, let $f(p, i)$ be the point $p + 2^i/m$. For each i between 1 and $\log m - 1$, each peer p maintains a link to the peer whose peer point is closest clockwise to the point $f(p, i)$. When a peer p links to a peer p' , the peer p simply stores the IP address of p' . The number of unique peers that a peer p links to is $O(\log n)$. For points p and k on the unit circle, let *next*(p, k) be the point in the set $\{f(p, 0), f(p, 1), f(p, 2), \dots, f(p, \log m - 1)\}$, which has closest clockwise distance to k .

We can now describe the *successor()* operation. Assume that some peer p calls *successor*(k) for some key k on the unit circle. If *next*(p, k) = p , then p already knows the successor of k : it is simply the closest clockwise peer to p . The search terminates by returning this peer. If *next*(p, k) = p' where $p' \neq p$, then p forwards the search request to p' . The same procedure is repeated until the search terminates.

2.2 Notation

For any two points x and y on the unit circle, let $d(x, y)$ be the distance from x to y traveling clockwise along the perimeter of the unit circle (i.e. if $y \geq x$, then $d(x, y) = y - x$ else $d(x, y) = 1 - x + y$). When referring to intervals or points on the unit circle, all addition is performed modulo 1. We will call a peer controlled by the adversary *faulty* and call a peer not controlled by the adversary (i.e. a peer that follows the protocol) *correct*.

2.3 S-Chord

In S-Chord, peers do not get to choose their own ID's. Instead they are assigned, by our protocol, a random ID between 0 and 1 when they first join the network. Following convention, for a given peer p , we will frequently use p to refer both to the peer and to the ID of the peer. The precise meaning should be clear from the context.

As in [2,3,7,9], we make use of the concept of small sets of peers working together as a single functional unit. Central to S-Chord is the notion of a *swarm*⁵. For every point x on the unit circle, we define the swarm, $S(x)$, to be the set

⁴ In practice, m is the number of bits in the ID's of the nodes.

⁵ This is essentially the same concept as a group in [2,3].

of peers whose ID's are located within a clockwise distance of $(C \ln n)/n$ of the point x (where C is a constant depending on our fault-tolerant parameters). For a given peer p , we will use $S(p)$ to mean the swarm associated with the peer p . All communication that p has with the DHT first passes through the swarm $S(p)$. Swarms, not peers, are the atomic functional units of S-Chord. We say that a swarm is *good* if at least a $3/4$ fraction of the peers in it are correct. Due to the fact that S-Chord randomly assigns ID's to peers, we can guarantee with high probability that over a z -good time interval, all swarms will be good. Thus, we can say that even though many peers are not correct, all of the swarms will be good. This fact is the basis for the robustness of S-Chord⁶.

Outline: We begin by assuming that all peers in the network know the values $\ln n$ and $(\ln n)/n$ exactly. In this extended abstract, we present protocols for 1) obtaining content from network and sending messages (Section 3) and 2) handling dynamic peer joins (Section 4).

In the full version of this paper, we provide the required modifications to S-Chord for the case where the peers do not know the values of $\ln n$ and $(\ln n)/n$. It also contains all proofs for results presented here as well as a protocol that allows for *SUCCESSOR* to incur only an expected constant factor increase in the number of bits sent over what is required for Chord. This second result assumes a computationally bounded adversary. A *STABILIZE* protocol, analogous to that given in the original Chord, is also provided in the full version.

2.4 Links Required

In this section, we state the links that each peer is required to maintain in S-Chord. We will often make statements referring to some correct peer p maintaining links to all peers in an interval $[a, b]$ for $a, b \in (0, 1]$. Assume that this means p maintains links to all *correct* peers and those faulty peers of which p is aware. Let C be a positive constant depending on k . Every peer p maintains links to all peers in the following intervals:

- *Center Interval:* $Center(p)$ is the set of peers in the interval $[p - (2C \ln n)/n, p + (2C \ln n)/n]$.
- *Forward Intervals:* For all i between 1 and $\log m - 1$, $Forward(p, i)$ is the set of peers in the interval $[p + 2^i/m - (C \ln n)/n, p + 2^i/m + (C \ln n)/n]$.
- *Backward Intervals:* For all i between 1 and $\log m - 1$, $Backward(p, i)$ is the set of peers in the interval $[p - 2^i/m - (C \ln n)/n, p - 2^i/m + (C \ln n)/n]$.

A peer p keeps track of the links in the Center interval so that 1) p knows all peers in $S(p)$, 2) p knows all peers p' such that $p \in S(p')$ and 3) p is able to help compute the *SUCCESSOR* algorithm described in Section 3. A peer p , keeps track of the Forward intervals so that is able to forward on requests for the *SUCCESSOR* function. While in Chord, requests for a successor are

⁶ It should be noted that S-Chord does not provide protection against the well-known Sybil attack [10].

Algorithm 1. $SUCCESSOR(p)$

- 1: p sends a request for k to all peers in $S(p)$;
 - 2: $S \leftarrow$ set of all peers in $S(p)$;
 - 3: $x \leftarrow$ identifier of p ;
 - 4: **while** $(d(x, k) > (C \ln n)/n)$ **do**
 - 5: $x' \leftarrow next(x, k)$;
 - 6: All peers in S send the request for k to all peers in $S(x')$;
 - 7: $S' \leftarrow$ set of all peers in $S(x')$ that received the above request from a majority of the peers in S ;
 - 8: $S \leftarrow S'$;
 - 9: $x \leftarrow x'$;
 - 10: **end while**
 - 11: The peers in S send back pointers to all the peers in $S(k)$. These pointers are sent backwards along the same path, in the same manner, to the peer p ;
-

forwarded to a single peer, in S-Chord, they are forwarded to an entire swarm. A peer p , keeps track of the Backward intervals so that it is able to recognize legitimate requests sent during computations of the $SUCCESSOR$ function. In our protocol, we do not trust a peer to tell us its identifier (i.e. where it is located on the unit circle). Thus, a peer p specifically requires links to Backward intervals in order to keep track of the IDs of those peers who may legitimately send p messages. All messages sent to p from peers which are not in one of p 's Backward intervals are ignored.

3 Sucessor Protocol

Algorithm 1 gives the pseudocode for our robust $SUCCESSOR$ protocol (which is analogous to the *successor* operation of Chord). For a point k on the unit circle, $SUCCESSOR(k)$ returns pointers to the peers in $S(k)$. As in Chord, k typically represents a key for some data item. $SUCCESSOR(k)$ returns pointers to the *set* of peers responsible for storing that data item. Thus, the $SUCCESSOR$ operation provides for redundant storage and lookups of data items.

For a key k and peer p , $SUCCESSOR(k)$ works as follows when called by p . Peer p initially sends the request for k to all peers in $S(p)$. Let x equal the ID of p and S be $S(p)$. Until $d(x, k) \leq (C \ln n)/n$, the following loop repeats: the peers in S forward the request to all peers in $S(x')$ where $x' = next(x, k)$. Let S' be the set of peers in $S(x')$ which receive the request from a majority of peers in S . The loop now repeats with S set to S' and x set to x' . When the loop terminates, $d(x, k) \leq (C \ln n)/n$, so all peers in the set S have pointers to all peers in $S(k)$. These pointers to peers in $S(k)$ are then sent backwards along the same path, in the same manner, to the originating peer p .

For a given peer p , message m and an interval I on the unit circle, we define $SEND_MESSAGE(m, I)$ to be an algorithm which allows p to send message m to all peers in the interval I . If I is of length $\Theta((\ln n)/n)$, it's straightforward to see how $O(1)$ calls to a modified $SUCCESSOR$ algorithm will create a $SEND_MESSAGE$ algorithm with latency $O(\log n)$ and message cost $O(\log^3 n)$ (the detailed pseudocode is omitted). When writing the $JOIN$ protocol, we will make use of the $SEND_MESSAGE$ algorithm.

We now describe conditions under which we can show that all swarms are good.

Lemma 1. *Assume that 1) all peer points are distributed uniformly at random on the unit circle; and 2) the fraction of faulty peers is no more than $1/4 - \epsilon$. Let k be any fixed integer and C be sufficiently large but depending only on k , then with probability at least $1 - 1/n^k$, the following statement is true. For any point x on the unit circle, the swarm $S(x)$ is good.*

We now provide a description of how S-Chord allows for the enforcement of a rule set on all peers in the system, provided that all swarms are good. The desired rule set must be known in advance by all correct peers. The rule set can be enforced by having the correct peers in a swarm act in concert to stop any prohibited behavior. For instance, if a faulty peer p attempts to abuse bandwidth resources by making excessive calls to $SUCCESSOR$, the correct peers in $S(p)$ can simply refuse to participate in the $SUCCESSOR$ calls after a certain pre-defined cut-off point.

4 Peer Joins

Pseudocode for the $JOIN$ algorithm is given in Algorithm 2 and an example run of the algorithm is illustrated in Figure 1. The $JOIN$ algorithm makes use of an algorithm, based on secure multiparty computation protocols, which allows a good swarm to choose a random number in the range $(0,1]$. Additionally, this protocol employs a result by Scheideler's which shows how to keep Byzantine peers well distributed on the unit circle [24]. In particular, Scheideler proposes the following algorithm. When a new peer joins the unit circle, it is temporarily assigned a random peer point r . Then two other peers with peer points, p_1 and p_2 are selected uniformly at random from the set of all peers. Finally, the peers at positions r , p_1 and p_2 are rotated: the joining peer is assigned to position p_1 , the peer formerly at position p_1 is assigned to position p_2 , the peer formerly at position p_2 is assigned to position r . A result by Scheideler shows that a join protocol augmented with this type of rotation will ensure that all swarms are good for a polynomial number of insertions and deletions. The two random peer points required to do this rotation are chosen using the algorithm given in [17].

The $JOIN$ algorithm assumes that peer p knows some correct peer q . In the algorithm, p first contacts peer q with p 's request to join the network. Peer q alerts $S(q)$ to this request and the peers in $S(q)$ first choose a random ID r for p using the algorithm discussed in the full version of this paper. Two peers, p_1

Algorithm 2 JOIN(p)

- 1: Peer p contacts some correct peer q which notifies $S(q)$ of p 's request to join;
 - 2: All peers in $S(q)$ both 1) come to consensus on a random number $r \in (0, 1]$ and 2) select two random peer points, p_1 and p_2 , uniformly at random from all peers currently in the DHT using the algorithm in [17]. Assume that r, p_1 , and p_2 are ordered clockwise along the unit circle;
 - 3: Using the *SEND_MESSAGE* algorithm, all peers in $S(p)$ notify peers in $Center(p_1)$ that p has joined the network and that p is taking the location of p_1 who is relocating. In same way, all peers in $S(p)$ notify peers in $Center(p_2)$ that p_1 is joining and that p_1 is taking the location of p_2 who is relocating. Finally, all peers in $S(p)$ notify all peers in $Center(r)$ that p_2 is joining;
 - 4: All peers in $S(q)$ get pointers to the peers in $Center(p_1)$, using $O(1)$ calls to the *SUCCESSOR* algorithm. All peers in $S(q)$ send these pointers to p . In a similar fashion, $S(q)$ sends pointers to the peers of $Center(p_2)$ to p_1 and sends pointers to peers of $Center(r)$ to p_2 ;
 - 5: The peers in $Center(p_1)$ send data items for all keys k such that $p \in S(k)$ and p then stores copies of these data items. Similar processes for 1) $Center(p_2)$ and p_1 and 2) $Center(r)$ and p_2 are performed;
 - 6: *PLACEMENT*(p);
 - 7: *PLACEMENT*(p_1);
 - 8: *PLACEMENT*(p_2);
-

and p_2 , are selected uniformly at random and rotation is effected. The peers in $S(q)$ introduce p to the peers of $Center(p_1)$.

The steps for updating of Forward and Backward intervals for p , p_1 , and p_2 are contained in the *PLACEMENT* protocol whose pseudocode is omitted from extended abstract. In *PLACEMENT*, all peers in $S(p)$ find all the peers in p 's Forward and Backward intervals. In addition, the peers in $S(p)$ introduce p to all peers, p' , in the network such that p is now in a Center, Forward or Backward interval for p' . In a similar fashion p_1, p_2 are rotated into their new positions and their new Center, Forward, and Backward intervals are established.

Lemma 2. *The JOIN protocol has the following properties with high probability:*

- JOIN has $\Theta(\log n)$ latency and $\Theta(\log^3 n)$ message complexity.
- After JOIN completes, peer p knows all peers in its Center, Forward and Backward intervals.

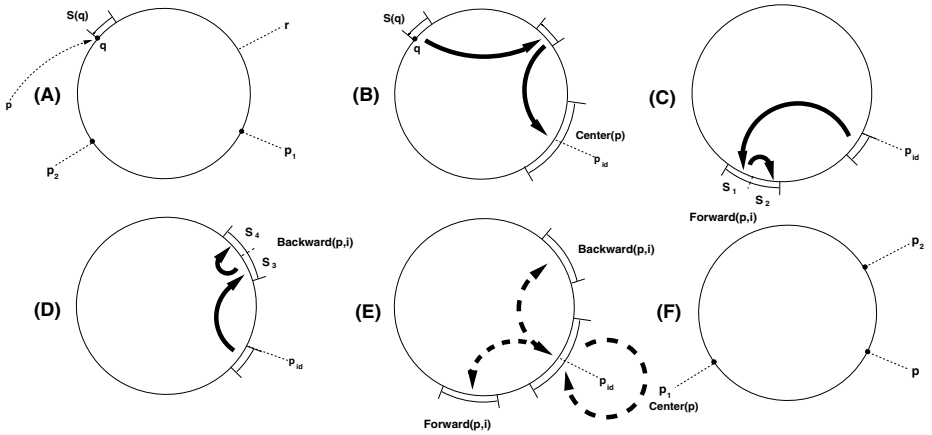


Fig. 1. An illustration of how p enters the network - the details for the rotation of p_1 and p_2 are omitted. (A) Peer p contacts q asking to join the network. The peers in $S(q)$ generate a random number $r \in (0, 1]$ and select two peer points uniformly at random. (B) All peers in $S(q)$ notify all peers in $Center(p)$ that p is joining and send to p the identifiers of and pointers to all peers in $Center(p)$. (C) Peers in $S(p)$ obtain the identifiers of and pointers to the peers in the i^{th} Forward interval of p . All peers in this Forward interval are informed of p 's arrival. This process is repeated with all Forward intervals of p . (D) Peers in $S(p)$ obtain the identifiers of and pointers to the peers in the i^{th} Backward interval of p . All peers in this Backward interval are informed of p 's arrival. Again, this process is repeated with all Backward intervals of p . (E) Links established after the join protocol. The thick dashed arrows illustrate links between p and the peers in its Forward, Backward, and Center intervals. There are links between p and the peers in all of its Forward and Backward intervals although this is not shown in this figure.

- Let q be any peer with the property that p is in a Center, Forward or Backward interval for q . Then after JOIN completes, q knows about the peer p .
- Assume, before p joins the network, that the fraction of faulty peers is no more than $1/4 - \epsilon$ and that all peer points are distributed uniformly at random on the unit circle. Then after p joins the network, all peer points are distributed uniformly at random on the unit circle.

5 $\Theta(\log^2 n)$ Expected Messages For SUCCESSOR

It is possible to improve *SUCCESSOR* so that it sends only $\Theta(\log^2 n)$ messages in expectation. We assume that all peers have a hash function h_1 which maps peer identifiers to the positive integers. We make the random oracle assumption about h_1 i.e. for any input, all outputs are equally likely. We also assume that the number of peers in any swarm is $\Theta(\log n)$ and at least $C \log n$ for some fixed constant C and that all swarms are good.

Algorithm 3 Message Sending Protocol

- 1: Each peer
- $x \in S_{j-1}$
- sends a message to peer
- $y \in S_j$
- iff

$$h_1(x) = h_1(y) \pmod{\log n}$$

- 2: Each peer
- $y \in S_j$
- accepts a message from peer
- $x \in S_{j-1}$
- iff

$$h_1(x) = h_1(y) \pmod{\log n}$$

- 3: Each peer
- $y \in S_j$
- , upon receiving messages from at least
- $2/3$
- rds of the peers that it would accept from, does majority filtering on all the messages received to decide which message, if any, to propagate to the next swarm.
-

Our algorithm for reducing message cost when sending from swarm S_{j-1} to swarm S_j is given in Algorithm 3. It assumes that swarm S_{j-1} wants to send a message to a swarm S_j (For ease of exposition, for a real number r , we will write r instead of $\lceil r \rceil$. It should be clear from context which is meant.). This algorithm is used in steps 6 and 7 of the *SUCCESSOR* pseudocode given in Algorithm 1.

Lemma 3. *For C sufficiently large but depending only on k' , the following is true with probability at least $1 - 1/n^{k'}$:*

- All calls to *SUCCESSOR* succeed.
- All calls to *SUCCESSOR* send $\Theta(\log^2 n)$ messages in expectation.

6 Conclusion

In this extended abstract, we have introduced the Byzantine join attack, an attack model under which an omniscient adversary causes a large number of Byzantine peers to join a network. We assume that the adversary carefully chooses the IP-addresses of these peers and where they join the network in order to try to place them at critical locations. We have described S-Chord, a variant of Chord that is provably robust to the Byzantine join attack. S-Chord also allows us to enforce a rule set on the peers in the network and thereby prevent undesirable behavior. In comparison to Chord's *successor*, this robustness is gained at the cost of an expected $\log n$ factor increase in the number of messages per *SUCCESSOR* operation and a $\log n$ factor increase in the number of links stored per peer. We believe that the techniques described here can be easily extended to a number of other ring-based DHTs that have a finger-function f which satisfies $|f(x) - f(x + \delta)| \leq \delta$ for any point x on the ring.

Acknowledgements

We gratefully thank Uri Nadav for his help with this paper. This research was supported by NSF grant CCR-0313160 and SURP grant No. 191445.

References

1. Aspnes, J., Shah, G.: Skip Graphs. Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (2003) 384–393
2. Awerbuch, B., Scheideler, C.: Robust Distributed Name Service. International Workshop on Peer-to-Peer Systems (IPTPS) (2004) 237–249
3. Awerbuch, B., Scheideler, C.: Group Spreading: A Protocol for Provably Secure Distributed Name Service. Proceedings of the Thirty-First Int. Colloquium on Automata, Languages, and Programming (ICALP) (2004) 183–195
4. Ben-Or, M., Canetti, R., Goldreich, O.: Asynchronous Secure Computation. Proceedings of the Twenty-Fifth ACM Symposium on the Theory of Computing (STOC) (1993)
5. Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computing. Proceedings of the Twentieth ACM Symposium on the Theory of Computing (STOC) (1988) 1–10
6. Ben-Or, M., Kelmer, B., Rabin, T. Asynchronous Secure Computations with Optimal Resilience. Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing (PODC) (1994) 183–192
7. Castro, M., Druschel P., Ganesh, A., Rowstron, A., Wallach, D.: Secure Routing for Structured Peer-to-Peer Overlay Networks. Proceedings of the 5th Usenix Symposium on Operating Systems Design and Implementation (OSDI) (2002) 299–314
8. Chaum, D., Crépeau, C., Damgård, I.: Multiparty Unconditionally Secure Protocols. Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing (STOC) (1988) 11–19
9. Dabek, F., Kaashoek, M., Karger, D., Morris, R., Stoica, I.: Wide-area cooperative storage with CFS. Proceedings of the 18th ACM Symposium on Operating Systems Principles (2001) 202–215
10. Douceur, J.: The Sybil Attack. Proceedings of the Second International Peer-to-Peer Symposium (IPTPS) (2002)
11. Fiat, A., Saia, J.: Censorship Resistant Peer-to-Peer Content Addressable Networks. Proceedings of the Thirteenth ACM Symposium on Discrete Algorithms (SODA) (2002)
12. Goldreich, O., Micali, S., Wigderson, A.: How to Play Any Mental Game - A Completeness Theorem for Protocols With Honest Majority. Proceedings of the Nineteenth ACM Symposium on Theory of Computing (STOC) (1987) 218–229
13. Harvey, N., Jones, M., Saroiu S., Theimer, M., Wolman, A.: SkipNet: A Scalable Overlay Network with Practical Locality Properties. Fourth USENIX Symposium on Internet Technologies and Systems(USITS) (2003)
14. Hildrum, K., Kubiawicz, J.: Asymptotically Efficient Approaches to Fault-Tolerance in Peer-to-peer Networks. Proceedings of the 17th International Symposium on Distributed Computing (2004)
15. Hirt, M., Nielsen, J., Przydatek, B.: An Asynchronous Multi-Party Computation Protocol. In Submission (2004)

16. Kashoek, M., Karger, D.: Koorde: A Simple Degree-Optimal Distributed Hash Table. Proceedings of the Second International Workshop on Peer-to-Peer Systems (IPTPS) (2003)
17. King, V., Saia, J.: Choosing a Random Peer. Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing (PODC) (2004)
18. Luby, M., Mitzenmacher, M., Shokrollahi, M., Spielman, D., and Stemann, V.: Practical loss-resilient codes. Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing (1997) 150–159
19. Naor, M., Wieder, U.: A simple fault tolerant distributed hash table. Proceedings of the Second International Workshop on Peer-to-Peer Systems (IPTPS) (2003)
20. Prabhu, B., Srinathan, K., Rangan, C.: Asynchronous Unconditionally Secure Computation: An Efficiency Improvement. INDOCRYPT 2002, Lecture Notes in Computer Science, Springer-Verlag **2551** (2002) 93–107
21. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A Scalable Content-Addressable Network. Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (2001)
22. Rowstron, A., Druschel, P.: Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, (2001) 329–350
23. Saia, J., Fiat, A., Gribble, S., Karlin, A., Saroiu, S.: Dynamically fault-tolerant content addressable networks. Proceedings of the First International Workshop on Peer-to-Peer Systems (2002)
24. Scheideler, C.: How to Spread Adversarial Nodes? Rotate! Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing (2005) 704–713
25. Srinathan, K., Rangan, C.: Efficient Asynchronous Secure Multiparty Distributed Computation. INDOCRYPT 2000, Lecture Notes in Computer Science, Springer-Verlag **1977** (2000) 117–129
26. Stoica, I., Morris, R., Karger, D., Kaashoek, M., Balakrishnan, H.: Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. Proceedings of the 2001 ACM SIGCOMM Conference (2001)
27. Yao, A.: Protocols for Secure Computations. Proceedings of the Twenty-Third IEEE Symposium on the Foundations of Computer Science (FOCS) (1982) 160–164
28. Zhao, B., Kubiawicz, J., Joseph, A.: Tapestry: An Infrastructure for Fault-Resilient Wide-Area Location and Routing. University of California at Berkeley Technical Report, UCB//CSD-01-1141, (April 2001)

Bucket Game with Applications to Set Multicover and Dynamic Page Migration*

Marcin Bienkowski¹ and Jarosław Byrka²

¹ International Graduate School of Dynamic Intelligent Systems,
University of Paderborn, Germany
young@upb.de

² Centrum voor Wiskunde en Informatica,
Kruislaan 413, NL-1098 SJ Amsterdam, Netherlands
J.Byrka@cwi.nl

Abstract. We present a simple two-person *Bucket Game*, based on throwing balls into buckets, and we discuss possible players' strategies. We use these strategies to create an approximation algorithm for a generalization of the well known Set Cover problem, where we need to cover each element by at least k sets. Furthermore, we apply these strategies to construct a randomized algorithm for Dynamic Page Migration problem achieving the optimal competitive ratio against an oblivious adversary.

1 Introduction

In this paper we present a simple two-player *Bucket Game*. In this game we have a set of n initially empty buckets, an infinite set of balls and a constant parameter $0 < c < 1$. In each turn, player A associates arbitrarily a non-negative weight w_i with each bucket i . We can easily extend the notion of weight to sets of buckets, i.e. the weight of a set X is the sum of weights of all buckets from X (the total weight is the sum of weights of all n buckets). On the basis of weights $\{w_i\}$, player B chooses a subset of buckets, whose weight is at least a fraction c of the total weight, and throws a ball into each bucket from this subset. The goal of player A is to fill each bucket with at least T balls for a given threshold T in as few rounds as possible, whereas the goal of player B is to postpone it.

One of the most straightforward questions that arises is: “Assuming the optimal strategy of player B , how fast can player A achieve the given threshold T ?”.

* Extended abstract. The full version of this paper is available under <http://www.hni.upb.de/publikationen/>.

¹ Partially supported by DFG-Sonderforschungsbereich 376 “Massive Parallelität: Algorithmen Entwurfsmethoden Anwendungen”, and by the Future and Emerging Technologies programme of the EU under EU Contract 001907 DELIS “Dynamically Evolving, Large Scale Information Systems”.

² Supported by the EU Marie Curie Research Training Network ADONET, Contract No MRTN-CT-2003-504438.

In this paper we address this issue, proving tight bounds in Sect. 2. The trivial answers to this question which we present are $\mathcal{O}(T \cdot \log n)$ and $\Omega(T + \log n)$. We develop a simple *Exponential Balancing* technique, which constructively yields an algorithm for player A . This algorithm is guaranteed to fill all the buckets to the given threshold T in $\mathcal{O}(T + \log n)$ rounds. All logs in the paper are to base 2, unless stated otherwise.

Although the Bucket Game might be interesting itself, the main contribution of our paper is applying the Exponential Balancing scheme to improve approximation and competitive ratios of Set Multicover and Dynamic Page Migration problems, respectively. We present the problems and summarize our results, separately, in the two following subsections.

1.1 Set Cover, Set Multicover and k -SetCover

The Set Cover problem is defined as follows. Given a collection \mathcal{C} of subsets of S , such that $\bigcup_{s_i \in \mathcal{C}} s_i = S$, find a collection $\mathcal{C}' \subseteq \mathcal{C}$, such that $\bigcup_{s_i \in \mathcal{C}'} s_i = S$ and $|\mathcal{C}'|$ is minimal.

The Set Cover problem is NP-complete. Moreover, it was proved by Raz and Safra [12] that the existence of an $(c \cdot \log n)$ -approximation algorithm for Set Cover with $c < 1$ would imply that $\text{NP} \subseteq \text{DTIME}(n^{\log \log n})$. On the other hand, Johnson [8] and Lovász [9] showed two different algorithms, both of which approximate Set Cover within a factor of $H_n = \sum_{i=1}^n \frac{1}{i} = \Theta(\log n)$.

A natural generalization of Set Cover is the Set Multicover problem, where each element x of S needs to be covered by at least l_x sets, and each subset $s_i \in \mathcal{C}$ may be used arbitrary number of times. This problem was addressed by Rajagopalan and Vazirani in [11], where an H_n -approximation algorithm was presented. We will, however, concentrate on the particular case where each element must be covered by at least k subsets (i.e. $l_x = k$ for all $x \in S$). We call this problem k -SetCover. A similar problem – a variant where each set may be used at most once (known as Constrained Set Multicover) – was recently shown in [3] to have applications to reverse engineering of protein and gene networks.

Our Contribution. We propose an algorithm for the k -SetCover problem, that in polynomial time produces a k -cover with $\mathcal{O}((k + \log n) \cdot c^*)$ sets where c^* is a number of sets in the optimal solution to the classical Set Cover problem. In other words, c^* is the cost of the optimal solution of the original input instance with $k = 1$. Furthermore, our algorithm can be extended without loss of approximation guarantee to handle a weighted version of k -SetCover, i.e. the one with different costs charged for using particular sets in the cover.

Our algorithm is based on a reduction from Set Cover to Uncapacitated Facility Location (UFL) problem. The reduction is similar to the one used by Guha and Khuller [7] to prove a lower bound on the approximation factor for UFL.

Although we do not improve the approximation ratio for k -SetCover achieved by the greedy algorithm of Vazirani, we believe the result is worth of interest. It relates the cost of the computed solution to c^* which is for some instances $\Omega(k)$ times smaller than the optimal solution to k -SetCover. We give an example of instances for which our algorithm computes $\Omega(\log n)$ times cheaper solutions.

1.2 Dynamic Page Migration

The *Dynamic Page Migration* (DPM) problem [4,5] arises in a network of n processors (nodes) v_1, v_2, \dots, v_n , which share one indivisible memory page (shared variable) of size D . This variable is stored in the local memory of one of these processors, initially at v_1 . The processors are placed in a metric space (\mathcal{X}, d) , i.e. the distances between the points from \mathcal{X} are given by the metric d .

We assume discrete time steps $t = 1, 2, \dots$. At the beginning of the step the adversary may move each node by at most a constant distance Δ , i.e. for any node v_i , its positions $p_{t-1}(v_i)$ and $p_t(v_i)$ in two consecutive time steps cannot be too far apart, $d(p_{t-1}(v_i), p_t(v_i)) \leq \Delta$. A tuple describing the positions of all the nodes in time t is called *configuration at time step t* and is denoted by \mathcal{C}_t .

After moving nodes, the adversary chooses one node, denoted by σ_t , which wants to access (read or write) a single unit of data from a page. If the page is stored in its local memory, then such a transaction is free. Otherwise, the node has to send a request to the processor holding the page, say v^* , and appropriate data is sent back. This incurs a cost, which is defined to be $c_t(\sigma_t, v^*) := d_t(\sigma_t, v^*) + 1$. The function $d_t(v_a, v_b)$ denotes the distance in step t between any two nodes v_a and v_b , i.e. $d_t(v_a, v_b) := d(p_t(v_a), p_t(v_b))$.

To avoid the problem of maintaining consistency among multiple copies of the page, the model allows only one copy of the page to be stored within the network. After serving the request, the algorithm may decide to migrate the page to another processor. The migration cost between two nodes v_a and v_b is equal to $D \cdot c_t(v_a, v_b)$.

The goal is to decide, online, when and where to move the page in order to exploit the locality of the request, and *minimize the total cost of communication* for all possible pairs of sequences of requests (σ_t) and network changes (\mathcal{C}_t) .

We consider only online algorithms, i.e. the ones which make decision in step t solely on the basis of the initial part of the input up to step t . To analyze the performance of an online algorithm ALG we use competitive analysis [13]. We say that a randomized algorithm ALG is c -competitive, if for all input sequences $(\mathcal{C}_t, \sigma_t)$ it holds $\mathbf{E}[C_{\text{ALG}}((\mathcal{C}_t, \sigma_t))] \leq c \cdot C_{\text{OPT}}((\mathcal{C}_t, \sigma_t))$, where the expected value is taken over all random choices made by an algorithm. $C_{\text{ALG}}((\mathcal{C}_t, \sigma_t))$ and $C_{\text{OPT}}((\mathcal{C}_t, \sigma_t))$ are the cost of ALG and the *optimal offline solution*, respectively, on the input sequence $(\mathcal{C}_t, \sigma_t)$. The factor c is called the *competitive ratio* of the algorithm. In this paper we consider only oblivious adversaries [2], which have no access to the random bits used by the algorithm.

Related Work. The case in which network is static, i.e. $\mathcal{C}_t = \mathcal{C}_{t+1}$ for all time steps t and the constant overhead for communication is not present, called *Page Migration*, had been introduced in [6]. The best known algorithms (deterministic and randomized), achieving constant competitive ratios for general topologies, were presented in [1,14].

For the DPM problem Bienkowski, Dynia, and Korzeniowski gave in [4] a deterministic, $\mathcal{O}(\min\{\sqrt{D} \cdot n, D, \lambda\})$ -competitive algorithm MARK, where λ is the maximum distance between any pair of nodes occurring during runtime. This result is up to a constant factor optimal due to the matching lower bound for

any randomized algorithm playing against an adaptive-online adversary given in [5]. In [4] it was also shown that a direct randomization of MARK yields the algorithm R-MARK which is $\mathcal{O}(\min\{\sqrt{D} \cdot \log n, D, \lambda\})$ -competitive against an oblivious adversary. The best known lower bound for this case, given in [4], is $\Omega(\min\{\sqrt{D} \cdot \log n, D^{2/3}, \lambda\})$.

Our Contribution. In Sect. 4 we partially close the gap mentioned above. We use Exponential Balancing technique to approximate the node holding page of the optimal algorithm by an accurate probability distribution over all nodes. We prove that our algorithm is $\mathcal{O}(\sqrt{D} \cdot \log n)$ -competitive. Since it can be combined with trivial $\mathcal{O}(D)$ and $\mathcal{O}(\lambda)$ algorithms from [5], we get an algorithm which is $\mathcal{O}(\min\{\sqrt{D} \cdot \log n, D, \lambda\})$ -competitive against an oblivious adversary.

2 Bucket Game

In this section we formally define a two-player *Bucket Game* and discuss possible strategies for each player.

Definition 1 (Bucket Game). *Assume we have a set of n buckets, which are initially empty, numbered from 1 to n and let $[n] := \{1, 2, \dots, n\}$. We also have an infinite set of balls. For any $i \in [n]$, let c_i be the current number of balls in bucket i . Let $0 < c < 1$ be any fixed constant. The Bucket Game is played in rounds by two players A and B . Each round of the game is defined as follows.*

- *Player A defines a sequence of non-negative weights $\{w_i\}_{i=1}^n$ and shows it to player B .*
- *Player B chooses some subset $X \subseteq [n]$ of buckets, s.t. $\sum_{i \in X} w_i \geq c \cdot \sum_{i=1}^n w_i$, and throws exactly one ball into each bucket from X .*

The game ends when each of the buckets contains at least T balls (i.e. $c_i \geq T$ for all $i \in [n]$). The goal of player A is to minimize the number of rounds, while B wants to play as long as possible.

Let us make the following simple observations. First of all, to throw at least one ball into each bucket (i.e. for the case $T = 1$), $\mathcal{O}(\log n)$ rounds are sufficient. Player A simply defines $w_i = 1$ for empty buckets and $w_i = 0$ for non-empty ones. Then in each round at least a fraction c of empty buckets gets a ball. Hence, after at most $\log_{1/(1-c)} n$ rounds there is no empty bucket left. Thus, to fill each bucket to the threshold T , player A can repeat this scheme T times, which yields an upper bound of $\mathcal{O}(T \cdot \log n)$.

Second, for any T , there exists a player B strategy which prevents finishing the game in less than $\Omega(T + \log n)$ rounds. Since each bucket may get at most one ball per round, the number of rounds cannot be smaller than T . On the other hand, suppose that in every round B chooses the subset with the smallest number of empty buckets. With this strategy, in the i -th round, at most $\lceil c \cdot e_i \rceil$ of empty buckets get a ball, where e_i is the number of empty buckets at the beginning of the i -th round. Thus $\Omega(\log n)$ rounds are also necessary.

Surprisingly, there is a simple player A strategy, which we call *Exponential Balancing*, and which asymptotically matches the above-mentioned lower bound.

Theorem 1. *For $T = \lfloor \log_2 n \rfloor$, there exists a player A strategy which guarantees finishing the game in $\mathcal{O}(\log n)$ rounds.*

Proof. In each round A defines $w_i = \frac{n}{2^{c_i}}$. We call w_i a *value of a bucket*, and define the *total value of the game* as $\mathcal{W} = \sum_{i \in [n]} w_i$.

Initially, all buckets are empty, $w_i = n$ for all i . Hence, the initial value of the game is n^2 . In any round each bucket “offers” to player B a half of its current value for putting a ball into this bucket. According to the rules of the game, B must collect at least a fraction c of this offered value. Thus, in every round, the game loses at least a fraction $\frac{1}{2} \cdot c$ of its value. If \mathcal{W} and \mathcal{W}' denote the game values in two consecutive rounds, then $\mathcal{W}' \leq (1 - c/2) \cdot \mathcal{W}$. If in some round $\mathcal{W} \leq 1$, then the threshold T is reached and the game ends. Precisely, $\mathcal{W} = \sum_i \frac{n}{2^{c_i}} \leq 1$ implies $\frac{n}{2^{c_i}} \leq 1$ for all $i \in [n]$, and thus $c_i \geq \log n$ for all $i \in [n]$. It remains to observe that the value of the game can be reduced from n^2 to 1 in at most $\log_{1/(1-c/2)} n^2 = \frac{2}{1-\log_2(2-c)} \cdot \log_2 n$ rounds. \square

If threshold T is larger than $\lfloor \log_2 n \rfloor$, then player A may act as if he was playing $\lceil \frac{T}{\lfloor \log_2 n \rfloor} \rceil$ times a game with threshold $\lfloor \log_2 n \rfloor$. By Theorem 1, each of these sub-games lasts $\mathcal{O}(\log n)$ rounds, and thus the whole game ends after at most $\mathcal{O}(T)$ rounds. Thus, we get the following.

Corollary 1. *For any T , there exists a player A strategy which guarantees finishing the game in $\mathcal{O}(T + \log n)$ rounds.*

3 Application to Set Multicover

In this section we present a new approximation algorithm for the k -SetCover problem, the modification of the Set Multicover defined in Sect. 1.1. Our algorithm uses an approximation algorithm for the Uncapacitated Facility Location (UFL) problem as a subroutine.

In the following, we say that A is a λ -approximation algorithm for a minimization problem P , if for any instance of the problem P it produces, in polynomial time, a solution with a cost at most λ times higher than the cost of an optimal solution.

Uncapacitated Facility Location (UFL) Problem. In the UFL problem we are given a set \mathcal{F} of n_f *facilities* and a set \mathcal{C} of n_c *cities*. For every facility $i \in \mathcal{F}$, a non-negative number f_i denotes the *opening cost* of the facility. Furthermore, for every city $j \in \mathcal{C}$ and facility $i \in \mathcal{F}$, c_{ij} is a *connection cost* between facility i and city j . The goal is to open a subset of the facilities $\mathcal{F}' \subseteq \mathcal{F}$, and connect each city to an open facility so that the total cost is minimized.

The UFL problem is NP-complete, and MAX SNP-hard (see [7]). A UFL instance is *metric* if its *connection cost* function satisfies the *triangle inequality* (i.e. $c_{ij} \leq c_{ik} + c_{kj}$ for any $i, j, k \in \mathcal{C} \cup \mathcal{F}$). There are several approximation

algorithms for the metric UFL problem, the currently best one achieving the approximation ratio of 1.52 [10].

Guha and Khuller [7] have proved by a reduction from Set Cover that there is no polynomial time λ -approximation algorithm for metric UFL with $\lambda < 1.463$, unless $NP \subseteq DTIME(n^{\log \log n})$. Another of their results was a $(1.463 \dots + \epsilon)$ -approximation algorithm for the case when all the connection costs are either 1 or 3. We use this algorithm in our construction.

Computing Partial Set-Covers. First, we address a problem of computing *partial set-covers* for a given instance of the Set Cover problem. By a *set-cover* we mean a feasible solution to an instance of the Set Cover problem (i.e. a family $C' \subset C$ covering every element of S), whereas a *partial set-cover* is a family $C'' \subset C$ that covers at least a certain fraction of elements of S .

We put weights on elements to indicate that covering certain elements is more important than covering others. We would like to know how much we can cover with at most $k \cdot c^*$ sets, for a constant $k \in \mathcal{R}_+$ and c^* denoting the number of sets in an optimal set-cover.

Let λ_0 be the approximation factor of an algorithm for the metric UFL problem with connection costs 1 or 3 (By [7], we may choose $\lambda_0 \approx 1.463$).

Lemma 1. *For all $k > \frac{2\lambda_0-1}{2-\lambda_0}$ there exists an algorithm, that given $((S, C), c^*, w)$ (where (S, C) is an instance of the Set Cover problem, c^* is the number of subsets in its optimal solution, and $w : S \rightarrow N_+$ is a weight function on the elements of S) runs in polynomial time in $\sum_{x \in S} w(x)$, outputs a partial set-cover C_p with the number of sets $c_p \leq k \cdot c^*$, and the total weight of elements not covered by C_p is at most $\frac{k(\lambda_0-1)}{2k-2\lambda_0} \cdot \sum_{x \in S} w(x)$.*

To prove this lemma, we use a reduction of a Set Cover instance to a UFL instance with distances 1 and 3 and an approximation algorithm for the UFL problem. The core of the reduction and the algorithm were proposed by Guha and Khuller [7]. We slightly extended the original reduction to encode the elements' weights into quantities of groups of cities representing particular elements. The complete proof of Lemma 1 can be found in the full version of the paper.

In Lemma 1 we bounded the fraction of uncovered weight of elements by $\frac{k(\lambda_0-1)}{2k-2\lambda_0}$, when $k \cdot c^*$ subsets are used. Suppose we want to cover certain fraction c of elements' weight and wonder how many sets do we need to use. We may consider the uncovered weight fraction $1-c = \frac{k(\lambda_0-1)}{2k-2\lambda_0}$ and conclude the following.

Corollary 2. *For any $0 < c < \frac{3-\lambda_0}{2} \approx 0.768$ there exists an algorithm that, in polynomial time, computes a partial set-cover that covers at least a fraction c of elements' weight, using at most $\frac{2\lambda_0 \cdot (1-c)}{3-\lambda_0-2c} \cdot c^*$ sets. We call this algorithm Set-UFL_c .*

3.1 Approximation Algorithm for k -SetCover

Now we combine Corollary 2 with Theorem 1 to present an algorithm for the k -SetCover problem. First, we present an algorithm A_1 (see Fig. 1), solving the

```

1.  $Sol \leftarrow \emptyset$  /* empty multiset */
2. guess  $c^*$  - the number of sets in an optimal set-covera
3. define weight function  $w : S \rightarrow \mathbb{N}_+$ 
4. compute a partial set-cover  $C_p \leftarrow \text{Set-UFL}_c((S, C), c^*, w)$ 
5. add this partial set-cover to the current solution ( $Sol \leftarrow Sol \cup C_p$ )
6. if  $Sol$  does not cover each element of  $S$  at least  $\lfloor \log_2 n \rfloor$  times, go to step 3
7. return  $Sol$ 

```

^a There are only polynomially many possible values of c^* .

Fig. 1. Algorithm A1

problem for $k = \log_2 n$, where $n = |S|$. A1, given an instance of a $(\log_2 n)$ -SetCover problem (S, C) , produces as a solution a multiset Sol .

Lemma 2. *There exists an (efficiently computable) weight function for step 3 of Algorithm A1, such that the algorithm produces, in polynomial time, a solution to $(\log_2 n)$ -SetCover instance with at most $\mathcal{O}(\log n \cdot c^*)$ sets.*

Proof. We use the Exponential Balancing technique to upper-bound the total number of sets in Sol . Let $c \in (0, 0.768)$ be a fixed parameter of the Algorithm A1. From Corollary 2, in step 4 of Algorithm A1, we may cover a fraction c of weight with at most $\frac{2\lambda_0(1-c)}{3-\lambda_0-2c} \cdot c^*$ sets.

Defining a weight function w in step 3 is like playing a role of player A in the Bucket Game, except for the fact that now the weights are restricted to be positive integers. Fortunately, the proof of Theorem 1 may be easily modified to use only integer, polynomially bounded weights. Hence, by Theorem 1, we may bound the number of used partial covers by $\frac{2\log_2 n}{1-\log_2(2-c)}$.

Concluding, $|Sol|$ - the total number of used sets, may be bounded as

$$|Sol| \leq \frac{4\lambda_0(1-c)}{(1-\log_2(2-c))(3-\lambda_0-2c)} \cdot \log_2 n \cdot c^* . \tag{1}$$

When we set $c = 0.553$ and $\lambda_0 \approx 1.463$, we obtain $|Sol| < 12.006 \cdot \log_2 n \cdot c^*$. \square

Theorem 2. *There exists an algorithm that for any instance of the k -SetCover problem, in polynomial time, produces a k -set-cover with $\mathcal{O}((k + \log n) \cdot c^*)$ sets, where c^* is the number of sets in an optimal classical set-cover.*

Proof. Let $r = \lceil k/\lfloor \log_2 n \rfloor \rceil$, and let $r * Sol$ denote a multi-set containing the same elements as the multi-set Sol , but the quantity of each element e in $r * Sol$ is r times the quantity of e in Sol . We use Algorithm A1 to compute a solution Sol , that covers each element at least $\lfloor \log_2 n \rfloor$ times, and multiply it r times to obtain a solution $r * Sol$. By Lemma 2, Sol has $\mathcal{O}(\log n \cdot c^*)$ sets. Thus, $r * Sol$ has $\mathcal{O}((k + \log n) \cdot c^*)$ sets. \square

Note on Weighted Version of Set Multicover. Like in the case of the classical Set Cover problem, one may generalize the Set Multicover problem

by defining a weight function on the subsets representing the cost of using a particular set in the solution. Same as in the unweighted version, we define the cost of using a set l times to be l times the cost of using this set only once.

All the results concerning the Set Multicover problem (especially Theorem 2) presented in this paper may be easily generalized to the Weighted Set Multicover problem formulation. The proof will be presented in the full version of the paper.

Theorem 3. *There exists an algorithm that for a given instance of the Weighted k -Set-Cover problem, in polynomial time, produces a k -set-cover with cost at most $\mathcal{O}(k + \log n)$ times the cost of an optimal solution to the classical Weighted Set Cover problem on this instance.*

Motivating Example. Let us consider the following instances of the $(\log n)$ -Set-Cover problem. Let the family C consist of subsets P_1, P_2, \dots, P_{n+1} with weights c_1, c_2, \dots, c_{n+1} such that $P_j = \{j\}$ and $c_j = 1/j$ for $j = 1, 2, \dots, n$, whereas $P_{n+1} = \{1, 2, \dots, n\}$ and $c_{n+1} = 1 + \epsilon$ for some $\epsilon \in \mathbb{R}_+$. Consider the greedy algorithm of Vazirani, i.e. the algorithm that consecutively chooses the most *cost effective* set. If we use it to produce $(\log n)$ -set-cover for the instance above, it outputs a multi-set containing each of the sets P_1, P_2, \dots, P_n $\log n$ times. This solution has a cost equal to $H_n \cdot \log n$. However, since the optimal solution to the classical Set Cover has cost $c^* = 1 + \epsilon$, by Theorem 3, our algorithm produces $(\log n)$ -set-cover with cost $\mathcal{O}(\log n)$.

4 Application to Dynamic Page Migration

In this section we design a randomized algorithm EBM (Exponential Balancing Marking) and prove that it achieves a competitive ratio of $\mathcal{O}(\sqrt{D \cdot \log n})$. Our algorithm is based on the MARK algorithm [4]. First, we construct a marking scheme, which induces the partition of input sequence into *epochs*. This partition is independent of the algorithm, and depends only on the input. On the basis of the computed marking we construct the EBM algorithm. We also use Bucket Game as an underlying concept, however the relation between EBM and the game is more obscure here.

Marking Scheme. We divide input sequence into chunks of length $K := 2 \cdot \sqrt{D/\log n}$ time steps. The partitioning of input sequence into epochs is performed as follows. Each epoch consists of some non-empty sequence of chunks. Let M_i be the number of marks that v_i has; initially all M_i are set to 0. Marks are DPM's equivalents of the balls from the Bucket Game.

The first epoch starts with the beginning of the input. Let \mathcal{E} denote the current epoch; at the beginning of input sequence $\mathcal{E} = \emptyset$. By a *subsequence* we understand any time interval of the input sequence. For any subsequence \mathcal{S} and any $v_i \in V$, let $A_i(\mathcal{S})$ denote the cost (of serving requests) of an algorithm which remains in v_i for the whole \mathcal{S} and does not move its page. After each chunk I_j we run the marking routine depicted in Fig. 2. Marking of v_i is computed entirely on the basis of $A_i(\mathcal{E})$. If at the end of I_j node v_i becomes marked (with one or

```

 $\mathcal{E} := \mathcal{E} \uplus I_j$ 
for each  $v_i \in V$  do  $M_i := \lfloor A_i(\mathcal{E}) \cdot \frac{\log n}{D} \rfloor$ 
if  $M_i \geq \log n$  for all  $v_i \in V$  then
  for each  $v_i \in V$  do set  $M_i := 0$ .
 $\mathcal{E} := \emptyset$  /* beginning of a new epoch */
    
```

Fig. 2. Marking routine after chunk I_j

more marks), i.e. the corresponding value of M_i increases, then I_j is called a *marking chunk* for v_i , and we say that v_i is *marked in I_j* . Additionally, if \mathcal{S} is any subsequence, then by $M_i(\mathcal{S})$ and $M'_i(\mathcal{S})$ we denote the number of marks v_i has before \mathcal{S} and after \mathcal{S} , respectively. We also define $\Delta M_i(\mathcal{S}) = M'_i(\mathcal{S}) - M_i(\mathcal{S})$. An epoch ends when all nodes are marked at least $\log n$ times.

As an important fact, we get that $C_{\text{OPT}}(\mathcal{E}) = \Omega(D)$ for each epoch \mathcal{E} . Actually, if OPT remains at one node v_i , then v_i is marked at least $\log n$ times during \mathcal{E} , and thus OPT pays $A_i(\mathcal{E}) \geq \log n \cdot \frac{D}{\log n} = D$. Otherwise, OPT pays at least D for moving the page between nodes.

Jump Sets. Before we construct our algorithm, we adapt the construction of Jump Sets from [4] for our needs. We consider one single chunk I and we number time steps within I from 1 to K . Then σ_i denotes the node which issues a request in the i -th step of I , and $d_i(\cdot)$, $c_i(\cdot)$ are the distance and cost functions in the i -th step. The following lemma is a simple reformulation of [4-Lemma 2].

Definition 2. A gravity center for I is a vertex v , which minimizes the sum $\sum_{i=1}^K c_K(v, \sigma_i)$. If there is more than one such vertex, then we choose any of them. We denote this node by \mathcal{G}_I .

Definition 3. For any chunk I and any integer $k \geq 1$, a k -JumpSet, which we denote by $J_k(I)$, is the set of all nodes whose distance to \mathcal{G}_I , measured in the last step of I is at most $9 \cdot k \cdot K$, i.e. $J_k(I) = \{v \in V : d_K(v, \mathcal{G}_I) \leq 9 \cdot k \cdot K\}$.

Lemma 3. For any chunk I of K steps, any node $v_i \in V$, and any $k \geq 1$, if $v_i \notin J_k(I)$ at the end of I , then $A_i(I) \geq \frac{k}{4} \cdot K^2 \geq k \cdot \frac{D}{\log n}$. This implies that if v_i gets k marks in I , then $v_i \in J_{k+1}(I)$.

4.1 The EBM Algorithm

Algorithm EBM works in chunks, i.e. it remains at one node for a chunk, and then at the end of the chunk, it makes its decision (where to move the page) on the basis of computed gravity centers and marking. If chunk I is the last chunk of the epoch, then EBM moves its page to \mathcal{G}_I . Otherwise, if the node holding the algorithm's page is marked in I , then at the end of I , EBM chooses randomly a node v^* , further called *I -JumpCandidate*, and moves its page to v^* . Any node v_i is chosen with probability $2^{-M_i} / (\sum_{i \in [n]} 2^{-M_i})$.

Theorem 4. *The algorithm EBM is $\mathcal{O}(\sqrt{D \cdot \log n})$ -competitive.*

Consider any epoch \mathcal{E} , and let m be the number of its chunks, i.e. $\mathcal{E} = (I_1, I_2, \dots, I_m)$. Movements of EBM’s page partition \mathcal{E} into a sequence of p phases, i.e. $\mathcal{E} = (P_1, P_2, \dots, P_p)$, each phase consisting of one or more chunks. In one phase P_i , EBM remains at one node, denoted by $P_{\text{EBM}}(P_i)$. First, we prove that the expected number of phases in one epoch is bounded.

Lemma 4. *The expected number of phases in one epoch is $\mathcal{O}(\log n)$.*

Proof. We define a *value of a node* after any chunk I as $n \cdot 2^{-M'_I(I)}$. The *total value* after I is defined as the sum of nodes’ values, i.e. $\mathcal{W}_I := \sum_{i \in [n]} n \cdot 2^{-M'_I(I)}$. We make two key observations. First, \mathcal{W}_I is monotonically non-increasing within \mathcal{E} . Second, $\mathcal{W}_I \leq n^2$ for any chunk $I \in \mathcal{E}$, and $\mathcal{W}_{I_{m-1}} \geq 1$ (because after I_{m-1} at least one node has less marks than $\log n$).

EBM starts \mathcal{E} at some node v and remains there till v becomes marked. This first phase lasts for at least one chunk, and after it the total value is at most n^2 . After the first phase, for the analysis, we may safely assume that EBM chooses a jump candidate v^* at the beginning of a phase, moves its page to v^* and remains at v^* till it becomes marked. Thus, this choice of a node v_i determines where the phase ends: either at the first marking chunk for v_i , or at I_m , if v_i is not marked in the remaining part of \mathcal{E} . This chunk we call *stopping* for v_i .

We show that, with probability at least $1/2$, one phase reduces the total value by a constant factor or is the last phase in the epoch. We call such phase *successful*. If the total value is reduced below 1, then the corresponding phase ends with I_{m-1} or with I_m , and thus at most one additional phase consisting of the last chunk I_m is sufficient. Thus, $\mathcal{O}(\log n)$ successful phases are sufficient to finish the whole epoch, and therefore the expected number of phases in epoch is $\mathcal{O}(2 \cdot \log n)$.

Consider the beginning of any phase. We sort the nodes in the order induced by their stopping chunks, obtaining a sorted sequence v_{i_1}, \dots, v_{i_n} . Let p_{i_1}, \dots, p_{i_n} be the probabilities of choosing these nodes as jump candidates. Let j be the smallest index for which $\sum_{k=1}^j p_{i_k} \geq 1/2$, and I' be the stopping chunk for v_{i_j} . Since j is the smallest index with this property, it follows immediately that, with probability $\sum_{k=j}^n p_{i_k} \geq 1/2$, EBM chooses one of $v_{i_j}, v_{i_{j+1}}, \dots, v_{i_n}$ as a jump candidate. Any such choice guarantees that the phase lasts at least to the end of I' . If $I' = I_m$, then the process ends here and the lemma follows. Otherwise, note that between the end of I and the end of I' , $v_{i_1}, v_{i_2}, \dots, v_{i_j}$ are marked at least once. Since probabilities p_{i_k} are directly proportional to the corresponding values of nodes, and $\sum_{k=1}^j p_{i_k} \geq 1/2$, these nodes’ values constitute at least half of the total value \mathcal{W}_I . By marking them once, one half of their values (and thus at least $1/4$ of the total value) is removed. Thus, $\mathcal{W}_{I'} \leq \frac{3}{4} \cdot \mathcal{W}_I$. \square

Before we analyze the cost of EBM in a single phase, we introduce the notion of potential Φ . If the distance between nodes holding the pages of OPT and EBM is equal to L , then $\Phi := 2 \cdot D \cdot L$. If \mathcal{S} is any subsequence of steps, then by $\Delta\Phi(\mathcal{S})$ we denote the difference between the potential right after and right before \mathcal{S} .

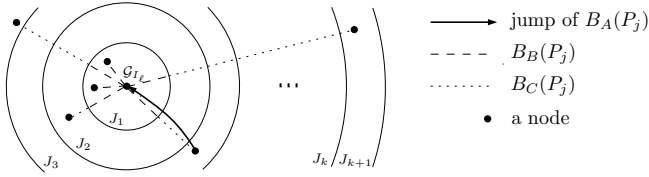


Fig. 3. Transports at the end of phase P_j

By an *amortized cost* of an action we understand the actual cost of this action plus the change in the potential this action induced.

In the following we bound the amortized cost of EBM in any phase P_j . Assume that P_j consists of ℓ chunks numbered from 1 to ℓ , i.e. $P_j = (I_1, I_2, \dots, I_\ell)$, and consider the following thought experiment. At the end of P_j , instead of moving directly to I_ℓ -JumpCandidate v^* , EBM first moves its page to \mathcal{G}_{I_ℓ} , and then to v^* . Note, that for the last phase in \mathcal{E} we do not need the latter part of this movement. Obviously, the (amortized) cost of this combined movement upper-bounds the (amortized) cost of the actual move.

We denote the part of the *amortized cost* EBM pays for serving all the requests in P_j and moving to \mathcal{G}_{I_ℓ} by $B_A(P_j)$. The remaining part of the cost depends on the random choice of v^* . Since we are interested only in the expected value of this variable, we can view this move as transporting parts of the page from \mathcal{G}_{I_ℓ} to the nodes. These transports are schematically presented in Fig. 3. Precisely, to node v_i we transport a part $p_i := 2^{-M'_i(P_j)} / (\sum_k 2^{-M'_k(P_j)})$ of the page, paying $p_i \cdot D \cdot c_K(\mathcal{G}_{I_\ell}, v_i)$. We divide the total *amortized cost* of this transport into two parts: a transport within the boundary of the 1-JumpSet (dashed lines in Fig. 3), denoted by $B_B(P_j)$, and a transport from this boundary to the appropriate nodes (dotted lines in Fig. 3), denoted by $B_C(P_j)$. We note that $B_B(P_j)$ and $B_C(P_j)$ are random variables. The following two lemmas are straightforward generalizations of [5–Lemma 3,4].

Lemma 5. *For any phase P holds $B_A(P) \leq \mathcal{O}(D/K) \cdot C_{OPT}(P) + \mathcal{O}(D \cdot K)$.*

Lemma 6. *For any phase P holds $B_B(P) \leq \mathcal{O}(D \cdot K)$.*

Lemma 7. *For any epoch $\mathcal{E} = (P_1, P_2 \dots P_p)$ holds $\mathbf{E}[\sum_{P_j \in \mathcal{E}} B_C(P_j)] = \mathcal{O}(D \cdot K \cdot \log n)$.*

The complete proof of Lemma 7 can be found in the full version of the paper. Here we mention only that the proof uses Lemma 3 to argue, that the B_C part of cost in one phase can be high (i.e. the page is transported far away from the gravity center), only if these far nodes received a lot of marks in this phase. This implies that an epoch \mathcal{E} cannot contain many of such phases.

Finally, we can combine the lemmas above to prove EBM’s competitiveness.

Proof (of Theorem 4). Consider any epoch \mathcal{E} and let $\mathcal{E} = (P_1, P_2, \dots, P_p)$ be its division into phases. Then, $\mathbf{E}[C_{EBM}(\mathcal{E}) + \Delta\Phi(\mathcal{E})] \leq \sum_{j=1}^p [B_A(P_j) + B_B(P_j)] +$

$\mathbf{E}[\sum_{j=1}^p B_C(P_j)]$, and by Lemmas 5, 6, and 7, the amortized cost is bounded by $\mathcal{O}(D/K) \cdot C_{\text{OPT}}(\mathcal{E}) + \mathbf{E}[p] \cdot \mathcal{O}(D \cdot K) + \mathcal{O}(D \cdot K \cdot \log n)$. Thus, using $\mathbf{E}[p] = \mathcal{O}(\log n)$ and $C_{\text{OPT}}(\mathcal{E}) \geq D$, we finally get $\mathbf{E}[C_{\text{EBM}}(\mathcal{E}) + \Delta\Phi(\mathcal{E})] \leq \mathcal{O}(\sqrt{D} \cdot \log n) \cdot C_{\text{OPT}}(\mathcal{E})$. By summing this inequality over all the epochs, the proof follows. \square

References

1. Y. Bartal, M. Charikar, and P. Indyk. On page migration and other relaxed task systems. In *Proc. of the 8th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 43–52, 1997.
2. S. Ben-David, A. Borodin, R. M. Karp, G. Tardos, and A. Wigderson. On the power of randomization in online algorithms. In *Proc. of the 22nd ACM Symp. on Theory of Computing (STOC)*, pages 379–386, 1990.
3. P. Berman, B. DasGupta, and E. Sontag. Randomized approximation algorithms for set multicover problems with applications to reverse engineering of protein and gene networks. In *Proc. of the 7th Int. Workshop on Approximation Algorithms for Combinatorial Optimization Problems*, pages 39–50, 2004.
4. M. Bienkowski, M. Dynia, and M. Korzeniowski. Improved algorithms for dynamic page migration. In *Proc. of the 22nd Symp. on Theoretical Aspects of Computer Science (STACS)*, pages 365–376, 2005.
5. M. Bienkowski, M. Korzeniowski, and F. Meyer auf der Heide. Fighting against two adversaries: Page migration in dynamic networks. In *Proc. of the 16th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 64–73, 2004.
6. D. L. Black and D. D. Sleator. Competitive algorithms for replication and migration problems. Technical Report CMU-CS-89-201, Department of Computer Science, Carnegie-Mellon University, 1989.
7. S. Guha and S. Khuller. Greedy strikes back: Improved facility location algorithms. In *Proc. of the 9th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 228–248, 1998.
8. D. S. Johnson. Approximation algorithms for combinatorial problems. In *Proc. of the 5th ACM Symp. on Theory of Computing (STOC)*, pages 38–49, 1973.
9. L. Lovász. On the ratio of the optimal integral and fractional covers. *Discrete Mathematics*, 13:383–390, 1975.
10. M. Mahdian, Y. Ye, and J. Zhang. Improved approximation algorithms for metric facility location problems. In *Proc. of the 5th Int. Workshop on Approximation Algorithms for Combinatorial Optimization Problems*, pages 229–242, 2002.
11. S. Rajagopalan and V. V. Vazirani. Primal-dual RNC approximation algorithms for set cover and covering integer programs. *SIAM Journal on Computing*, 28(2):525–540, 1999.
12. R. Raz and S. Safra. A sub-constant error-probability low-degree test, and a sub-constant error-probability PCP characterization of NP. In *Proc. of the 29th ACM Symp. on Theory of Computing (STOC)*, pages 475–484, 1997.
13. D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.
14. J. Westbrook. Randomized algorithms for multiprocessor page migration. *DI-MACS Series in Discrete Mathematics and Theoretical Computer Science*, 7: 135–150, 1992.

Bootstrapping a Hop-Optimal Network in the Weak Sensor Model

Martín Farach-Colton, Rohan J. Fernandes, and Miguel A. Mosteiro

Department of Computer Science, Rutgers University,
Piscataway, NJ 08854, USA
{farach, rohanf, mosteiro}@cs.rutgers.edu

Abstract. Sensor nodes are very weak computers that get distributed at random on a surface. Once deployed, they must wake up and form a radio network. Sensor network bootstrapping research thus has three parts: one must model the restrictions on sensor nodes; one must prove that the connectivity graph of the sensors has a subgraph that would make a good network; and one must give a distributed protocol for finding such a network subgraph that can be implemented on sensor nodes.

Although many particular restrictions on sensor nodes are implicit or explicit in many papers, there remain many inconsistencies and ambiguities from paper to paper. The lack of a clear model means that solutions to the network-bootstrapping problem in both the theory and systems literature all violate constraints on sensor nodes. For example, random geometric graph results on sensor networks predict the existence of subgraphs on the connectivity graph with good route-stretch, but these results do not address the degree of such a graph, and sensor networks must have constant degree. Furthermore, proposed protocols for actually finding such graphs require that nodes have too much memory, whereas others assume the existence of a contention-resolution mechanism.

We present a formal WEAK SENSOR MODEL that summarizes the literature on sensor node restrictions, taking the most restrictive choices when possible. We show that sensor connectivity graphs have low-degree subgraphs with good *hop-stretch*, as required by the WEAK SENSOR MODEL. Finally, we give a WEAK SENSOR MODEL-compatible protocol for finding such graphs. Ours is the first network initialization algorithm that is implementable on sensor nodes.

1 Introduction

Advances in technology have made it possible to integrate sensing, processing and communication in a low-cost device, popularly known as a *sensor node*. Sensor nodes are randomly deployed over an area and must self-organize as a radio-communication network called a *sensor network*. Even though communication

This research was supported in part by DIMACS, Center for Discrete Mathematics & Theoretical Computer Science, grants numbered NSF CCR 00-87022, NSF EIA 02-05116 and Alfred P. Sloan Foundation 99-10-8.

among sensor nodes is through radio broadcast, it is useful to set up explicit links between nodes in order to establish routing paths and prevent flooding.

A sensor network is capable of achieving large tasks through the coordinated effort of sensor nodes, but individual nodes have severe limitations on memory size, life cycle, range of communication, etc. Any sensor network initialization algorithm must be fast and distributed, and must resolve channel contention issues. The network constructed by such an algorithm must be connected and must have low degree and diameter. The limitations on individual sensors nodes make this problem non-trivial, and its adequate resolution is crucial for making sensors useful.

There are two main types of issues in sensor network formation: those relating to geometric properties and those relating to network protocols; and any solution achieved for either must be compatible with an accurate model of sensor nodes. On the one hand, coverage and connectivity in sensor networks are dependent on the distribution of nodes in an area and the range of transmission of each node. Additionally, the density of nodes in an area determines the minimum path length between any two nodes in the induced connectivity graph. The limited range of transmission makes these properties geometric. On the other hand, protocols for sensor network formation are limited by the fact that sensor nodes share a common channel of communication and that they do not typically have access to directional or positional information. Memory limitations in sensor nodes also impose the restriction that a node can only keep track of $O(1)$ neighbors.

The existing literature on sensor network initialization does not sufficiently handle all aspects of the problem. All random geometric graph results related to ad-hoc wireless networks require $\omega(1)$ degree (see e.g. [9]). All proposed protocols for sensor network formation include some inappropriate hardware assumptions. For example, the sensor network formation protocol in [14] builds a constant-degree network, but relies on positional information hardware. The protocol proposed in [2] also builds a constant degree network, but relies on the preexistence of a scheme for channel-contention resolution. The different models implicit in such results are inadequate and poorly reflect the various limitations under which sensor nodes operate, and indeed, there seems to be considerable confusion in the literature as to what are or are not reasonable assumptions about the capabilities of sensor nodes.

In this paper, we present a formal WEAK SENSOR MODEL that summarizes the literature on sensor node restrictions, taking the most restrictive choices when possible. Given the WEAK SENSOR MODEL, we argue that a good sensor network must have constant degree and low *hop-stretch*, which we define below. We show that any appropriate random geometric graph has such a subgraph. Finally, we give a WEAK SENSOR MODEL-compatible protocol for finding such a subgraph. Ours is the first network initialization algorithm that is implementable on sensor nodes.

1.1 Related Work

Threshold Properties in Random Geometric Graphs. In the Random Geometric Graph Model $\mathcal{G}_{n,r,\ell}$, n nodes are distributed uniformly at random in $[0, \ell]^2$, and nodes are connected by an edge iff they are at Euclidean distance at most r , the *connectivity radius*. The node density depends on the relative values of n, r and ℓ . A specific instance of $\mathcal{G}_{n,r,\ell}$ is a *Random Geometric Graph (RGG)*, also referred to as $G(n, r, \ell)$.

Given two nodes u, v in a geometric graph, $stretch(u, v)$ is defined as the ratio of the shortest distance between u and v in the graph to the normed distance between the two points in the plane. *Route stretch* is the maximum of the $stretch(u, v)$ over all pairs of points in the graph. In a $G(n, r, \ell)$, the asymptotic behavior of route stretch is studied as $\ell \rightarrow \infty$ while maintaining sufficient density to preserve connectivity. In a seminal paper, Gupta and Kumar [5] computed the minimum radius needed to obtain a large connected component with high probability in $\mathcal{G}_{n,r,1}$. In $\mathcal{G}_{n,r,\ell}$, tight thresholds for connectivity, coverage and route stretch were shown by Muthukrishnan and Pandurangan [9] using an overlapping dissection technique called *bin-covering*. More recently, Goel, Krishnamachari and Rai [4] showed that all monotone graph properties have sharp thresholds for random geometric graphs.

Sensor Networks. A protocol for bootstrapping sensor networks was presented in [13]. In order to avoid collisions, the number of channels needed is a function of the density, which makes it infeasible. A network formation protocol, where node degree k is a constant tuned to ensure connectivity w.h.p., is given in [2]. This protocol relies on expensive distance estimation hardware such as GPS. Recently, an energy efficient topology control scheme was presented in [14]. This algorithm requires the use of a directional antenna and distance estimation hardware. In all these schemes, no contention resolution mechanism is given, and $\omega(1)$ memory size is assumed.

Bluetooth. A significant amount of research related to scatternet (a type of bounded degree network) formation has been done for Bluetooth. In these networks the nodes have less restrictive constraints (like power supply, range of transmission, memory capacity, etc.) than in sensor networks. Several schemes for scatternet formation have been proposed [7, 15, 16, 3]. Techniques proposed in these are either strictly heuristic or cannot be implemented on sensor nodes.

1.2 Roadmap

The remainder of this paper is organized as follows. In §2 we describe the WEAK SENSOR MODEL. In §3 we analyze geometric properties of good sensor networks. In §4 we present and analyze a distributed algorithm for finding good sensor networks. We conclude in §5.

2 The Weak Sensor Model

Bar-Yehuda, Goldreich and Itai [1] presented a formal model of a radio network that specifies many of the important restrictions on sensor nodes, including e.g.

limits on contention resolution, but they make no mention of computational limits, such as small memory. Since then, some papers, such as [10, 6, 8], have added more restrictions, although often such restrictions are implicit in the text or algorithms rather than fully specified.

Here, we specify the WEAK SENSOR MODEL:

- MEMORY SIZE: Sensor nodes may store a constant number of $O(\log n)$ bit words.
- LOW-INFORMATION CHANNEL CONTENTION: The communication with neighbors is through broadcast on a shared channel. If more than one message is sent at the same time, a collision occurs and no message is delivered. Furthermore, we require NO COLLISION DETECTION, where only two states are feasible, single transmission and silence/collision [1]. Finally, sensors nodes have NON-SIMULTANEOUS RECEPTION AND TRANSMISSION, so that transmitters also cannot detect collisions.
- DISCRETE TRANSMISSION RANGE: We assume that sensor nodes can adjust their power of transmission to only a *constant* number of levels.
- ASYNCHRONICITY: No global clock or other synchronizing mechanism is assumed, but all sensor nodes have the same clock frequency. We assume that time is divided into *slots*. This does not affect the asymptotic time complexity [11].
- Other: LIMITED LIFE CYCLE, SHORT TRANSMISSION RANGE, ONE CHANNEL OF COMMUNICATION, NO POSITION INFORMATION, ADVERSARIAL NODE WAKE-UP SCHEDULE, UNRELIABILITY.

3 Geometric Analysis of Sensor Networks

Recall that sensor nodes may only set up links with a constant number of neighbors, a consequence of the memory size limitation in the WEAK SENSOR MODEL (WSM), and since sensor nodes are distributed uniformly at random, the potential connectivity relation defines a *Random Geometric Graph* (RGG). Hence, any protocol for network formation must set up links defining a constant-degree spanning subgraph of the RGG. However, ignoring potential links may result in an increase in path lengths in the subgraph. This increase in path length can be measured in two ways: in terms of increase in the number of *hops* or increase in route stretch.

In applications where the propagation delay is significant, route stretch is an appropriate measure of optimality. However, sensor networks have small inter-node distances, and propagation delay is low. One of our primary concerns in the WSM is that we should minimize energy consumption at each node so as to maximize the life cycle. Thus, a Sensor Network is optimal when it minimizes the number of transmissions, which is to say, minimizes the number of hops in each path, rather than the weighted path length. A formal definition of stretch in terms of hops follows.

Let the *length* of a path connecting two nodes in a given graph be the number of edges of such a path. Let $d_{min}(u, v)$ be the shortest path between two nodes

u and v in the RGG $G(n, r, \ell)$. Let $D(u, v)$ be the Euclidean distance between u and v in the plane. Note that in $G(n, r, \ell)$, $\lceil D(u, v)/r \rceil$ is a lower bound on $d_{min}(u, v)$. Call this lower bound, $d_{opt}(u, v)$. The *hop-stretch* of (u, v) is defined as the ratio $d_{min}(u, v)/d_{opt}(u, v)$. The *hop-stretch* of $G(n, r, \ell)$ is the maximum of the *hop-stretch* of (u, v) over all pairs of points (u, v) in $G(n, r, \ell)$.

Note that schemes have been proposed that attempt to minimize energy consumption [14], and these favor many short hops over a few long ones. However, any such scheme ignores the contention resolution overhead of the extra hops and, furthermore, requires an $\omega(1)$ number of transmission power levels. In the rest of this section we will outline a scheme to obtain a constant degree hop-optimal subgraph from a sufficiently dense random geometric graph.

3.1 Disk Covering Scheme for Network Formation

Before describing the scheme, we introduce some necessary terminology. A *Random Geometric Graph* (RGG) or $G(n, r, \ell)$ is an instance of $\mathcal{G}_{n,r,\ell}$, where r is the connectivity radius. Given a sufficiently dense $G(n, r, \ell)$ as input, the Disk Covering Scheme produces as output a spanning subgraph with constant degree and asymptotically optimal path length. The precise nature of the path length optimality is given in the proof of Theorem 2. The graph so obtained is called the *Constant-degree Hop-optimal Spanning Graph* (CHSG). In the Disk Covering Scheme, a and b are tunable parameters that affect maximum degree and hop-stretch of the CHSG.

The following pseudocode summarizes the Disk Covering Scheme.

1. Add all nodes from the RGG to the CHSG.
2. Lay down *small* disks of radius $ar/2$, $0 < a < 1$ centered on nodes, such that no central node is covered by more than one small disk and no node is left uncovered. We call each central node a *bridge*. Note that the bridges form a Maximal Independent Set (MIS) of the spanning subgraph $G(n, ar/2, \ell) \subseteq G(n, r, \ell)$.
3. Add to the CHSG all edges from the RGG that connect bridges.
4. Expand the small disks into *big* disks of radius $br/2$, $a < b \leq 1$.
5. Add to the CHSG the necessary edges to form a spanner of constant degree among nodes covered by the same big disk. We call this spanner a *disk-spanner*.

3.2 Analysis of the Disk Covering Scheme

In this section the Disk Covering Scheme described in §3.1 is proved to produce a CHSG with asymptotically optimal path length. In section§3.2 we establish a bound on the maximum degree of a node in the CHSG. In section§3.2 two useful results for a connected $G(n, r, \ell)$ are established: A bound on hop-stretch and bounds on the node density. Finally, in section§3.2 we prove a theorem on the hop-optimality of the CHSG.

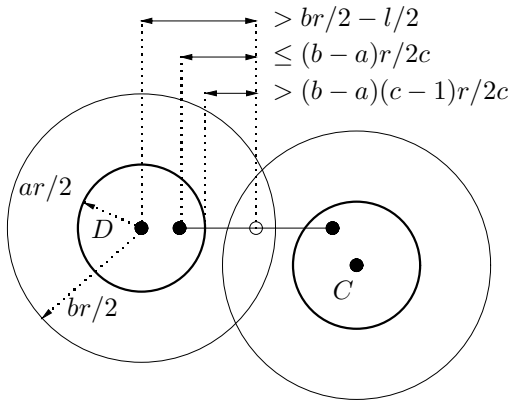


Fig. 1. Illustrating the proof of coverage of edges of length $\leq (b - a)r/c$

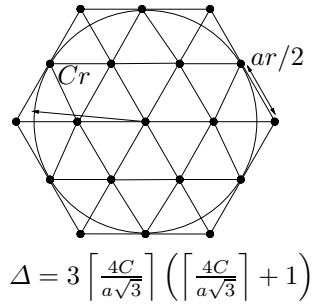


Fig. 2. Illustrating the upper bound on the degree, where $C = 1$ and degree is $\Delta + 1$ for bridges, and $C = b/2$ and degree is $\Delta * \text{Spanner degree}$ for non-bridges

Degree Bound

Lemma 1. *At the end of the Disk Covering Scheme, each edge of length at most $(b - a)r/c$ has both endpoints within a single big disk w.h.p, for any constant $c > 1$.*

Proof. The proof is illustrated in figure 1 and it is omitted here for brevity.

Lemma 2. *The degree of any node in the CHSG is in $O(1)$.*

Proof. The proof is illustrated in figure 2 and it is omitted here for brevity.

Hop Stretch and Density in $G(n, r, \ell)$. Theorem 1 demonstrates the existence of a path with an asymptotically optimal hop-stretch. The proof of the theorem uses an overlapping dissection technique, called bin-covering, presented by Muthukrishnan and Pandurangan [9].

Theorem 1. *In a $G(n, r, \ell)$ satisfying the following conditions: $r^2n = k\ell^2 \ln \ell$, $r = \theta(\ell^\epsilon f(\ell))$, $f(\ell) \in o(\ell^\gamma)$, $\gamma > 0$, $0 \leq \epsilon < 1$, and $0 < \alpha \leq 1$ is a fixed constant, for any constant $k > 5 \frac{4+\alpha^2}{\alpha}$, the hop-stretch is $1 + \sqrt{\alpha^2 + 4}$ w.h.p.*

Proof. It is enough to show that for any pair of nodes (u, v) , there is a path P defined by a sequence of nodes $\langle u = x_0, x_1, \dots, x_m = v \rangle$ such that the ratio between the length of P and the number of hops, m is bounded upwards by $1 + \sqrt{\alpha^2 + 4}$ w.h.p.

For a given pair of nodes (u, v) , the bin covering technique is applied as follows. Let r' be the shortest horizontal projection of a segment of length r contained in the strip, i.e. $r' = r/\sqrt{1 + (\alpha/2)^2}$. The line connecting u and v is

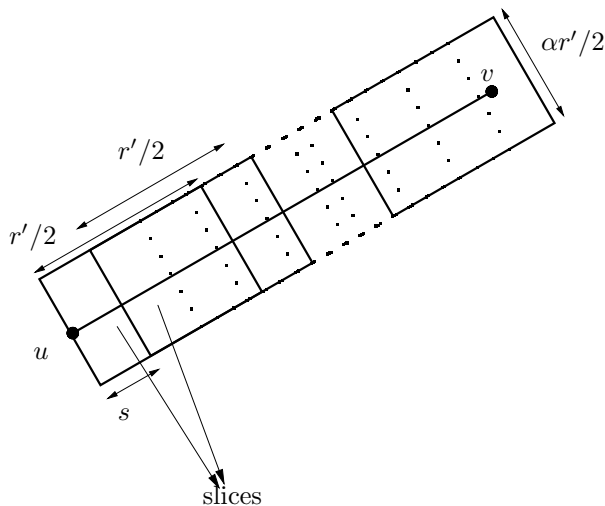


Fig. 3. Strip between nodes u and v showing bin covering and slices

covered with overlapping bins of dimension $r'/2 \times \alpha r'/2$ with a spacing parameter s , as shown in figure 3. This bin layout will be referred to as a *strip*.

The coordinate system is rotated such that the line segment $\overline{u,v}$ is parallel to the x axis. In what follows all distances are specified within this rotated frame of reference. Let $D_h(x,y)$ and $D_v(x,y)$ be the horizontal and vertical distances respectively between the nodes x and y .

Given a node x_j in the path P the node x_{j+1} is selected using the following criteria:

- The node x_{j+1} lies within the strip.
- $D_h(x_j, x_{j+1}) \leq r'$.
- The horizontal distance $D_h(x_{j+1}, v)$ is minimized.

A *hole* is a rectangle of dimension $r'/2 \times \alpha r'/2$, within a strip, that is devoid of nodes and adjoins a node on the side closest to u .

Consider any 3 consecutive nodes along the path x_{i-1}, x_i, x_{i+1} where $0 < i < m$, and assume that along any strip there is no hole, then $D_h(x_{i-1}, x_i) \geq r'/2$. To see that this claim is true, assume for the sake of contradiction that $D_h(x_{i-1}, x_i) < r'/2$. The distance $D_h(x_{i-1}, x_{i+1}) > r'$, otherwise x_{i+1} would have been selected as the successor of x_{i-1} . Thus, the distance $D_h(x_i, x_{i+1}) > r'/2$. Since there cannot be any hole in the strip, there exists a node y such that $D_h(x_i, y) < r'/2$. This implies that $D_h(x_{i-1}, y) < r'$. Note that $D_h(y, v) < D_h(x_i, v)$, therefore y should have been chosen as the successor of x_{i-1} by the construction criteria, which is a contradiction. The initial assumption of $D_h(x_{i-1}, x_i) < r'/2$ is thus proven false which proves the truth of the claim.

Since $D_h(x_{i-1}, x_i) \geq r'/2$ for $0 < i < m - 1$, the number of hops in the path P is

$$m \leq \left\lceil \frac{D(u, v)}{r'/2} \right\rceil = \left\lceil \sqrt{\alpha^2 + 4} \frac{D(u, v)}{r} \right\rceil .$$

If $D(u, v) \leq r$ the path is simply the edge connecting u and v and the hop-stretch is trivially 1. Otherwise, $D(u, v) > r$ and so, the hop-stretch is $1 + \sqrt{\alpha^2 + 4}$.

It remains to show that there is no hole *w.h.p.*

To bound the probability that there is a hole in any strip, consider the sequence of small rectangles (call them *slices*) defined by the spacing parameter, of size $s \times \alpha r'/2$. The slices are numbered in ascending order from u to v .

For any node x_i that is contained in some slice j , let E_i be the event that the node x_{i+1} is contained in the slice $j - 1 + \lceil r'/2s \rceil$ at a horizontal distance greater than r' from x_i . Then,

$$Pr[E_i] \leq \binom{n-1}{1} \frac{\alpha r' s}{2\ell^2} \left(1 - \frac{\alpha r'^2}{4\ell^2}\right)^{n-2} .$$

If x_{i+1} is contained in a slice closer to x_i then there is no hole. If x_{i+1} is contained in a slice farther than $j - 1 + \lceil r'/2s \rceil$ then there is at least one empty bin in the strip. The probability that some bin is empty is bounded by

$$Pr[\text{EmptyBin}] \leq \frac{\max_{(u,v)} D(u, v)}{s} \left(1 - \frac{\alpha r'^2}{4\ell^2}\right)^n .$$

Therefore, the probability that there is a hole within any strip is

$$\begin{aligned} Pr[\text{Hole}] &\leq \binom{n}{2} \left(n(n-1) \frac{\alpha r' s}{2\ell^2} \left(1 - \frac{\alpha r'^2}{4\ell^2}\right)^{n-2} + \frac{\max_{(u,v)} D(u, v)}{s} \left(1 - \frac{\alpha r'^2}{4\ell^2}\right)^n \right) \\ &\leq n^2 \frac{1}{e^{n\alpha r'^2/4\ell^2}} \left(\frac{n^2 \alpha r' s}{2\ell^2} e^{\alpha r'^2/2\ell^2} + \frac{\sqrt{2}\ell}{s} \right) . \end{aligned}$$

This expression is minimized when

$$s = \left(\frac{2\sqrt{2}\ell^3}{n^2 \alpha r' e^{\alpha r'^2/2\ell^2}} \right)^{1/2} .$$

Then,

$$\begin{aligned} Pr[\text{Hole}] &\leq \frac{2k^3 \ell^6 \ln^3 \ell}{r^6 \ell^{1+(k\alpha/(4+\alpha^2))}} \left(\frac{\alpha r' e^{\alpha r'^2/2\ell^2}}{2\sqrt{2}\ell} \right)^{1/2} \\ &\in O(\ell^{-\gamma}) \text{ for } k > 5 \frac{4 + \alpha^2}{\alpha} . \end{aligned}$$

Lemma 3. *In a $G(n, r, \ell)$ satisfying the parameter conditions of Theorem 1, the number of nodes contained in a circle of radius $\Theta(r)$ is $\Theta(\log \ell)$ *w.h.p.**

Proof. The proof uses a simple application of the Chernoff-Hoeffding bounds and is omitted in this extended abstract for the purpose of brevity.

Hop Optimality of the CHSG

Lemma 4. *Consider the RGG $G(n, r, l)$, where n satisfies the parameter conditions of Theorem 1 for a reduced connectivity radius of $r' = (b - a)r/c$. For any pair of nodes (u, v) in the RGG at Euclidean distance $D(u, v)$, there exists a path between them in the CHSG of at most $\lceil c\sqrt{\alpha^2 + 4D(u, v)/(b - a)r} \rceil - 1 + O(\log \ell)$ edges w.h.p.*

Proof. Theorem 1 states that: In the RGG that satisfies the parameter conditions of Theorem 1, there exists a path of $\lceil \sqrt{\alpha^2 + 4D(u, v)}/r \rceil$ edges w.h.p. We can thus imply that: If the RGG satisfies the same parameter conditions for a reduced connectivity radius of $r' = (b - a)r/c$, there exists a path between u and v using $\lceil c\sqrt{\alpha^2 + 4D(u, v)/(b - a)r} \rceil$ edges of length at most $(b - a)r/c$. Let p be such a path and e_1, e_2, \dots, e_m be its sequence of edges.

In the description of the Disk Covering Scheme, two kinds of disks were defined for clarity: big disks and small disks. In order to prove hop-optimality of the CHSG, we only refer to big disks and simply call them disks.

Lemma 1 states that every edge in the path p is completely covered by one disk. Therefore, there exists a sequence $d_1, d_2, \dots, d_{m'}$ of overlapping disks, where any edge e_i in p is covered by some disk d_j in this sequence. A disk may completely cover more than one edge, hence $m' \leq m$. Let D_i be the bridge (center) of disk d_i .

Define a path p' using only edges of the CHSG as follows. Connect u and the bridge D_1 with a path p_1 of disk-spanner edges defined by the disk d_1 . For each edge i , $1 \leq i \leq m$, replace the edge e_i in p with the node D_i . Connect all consecutive bridges D_i and D_{i+1} within the path of overlapping disks with edge $\overline{D_i D_{i+1}}$. Consecutive bridges are adjacent to each other in the RGG, because their disks overlap and the radius of each disk is $br/2$ with $b \leq 1$. Finally, connect the bridge D_m and v with a path p_m of disk-spanner edges defined by the disk d_m . The length of p' is given by: $length(p') \leq length(p_1) + (m - 1) + length(p_m)$. Using the stretch bound, $length(p') \leq \lceil c\sqrt{\alpha^2 + 4D(u, v)/(b - a)r} \rceil - 1 + length(p_1) + length(p_m)$ w.h.p. Only disk-spanner edges are used in p_1 and p_m . It is shown in Lemma 3 that the number of nodes within a disk is $O(\log \ell)$ w.h.p. Therefore, $length(p_1) + length(p_m) = O(\log \ell)$ w.h.p. completing the proof.

The following theorem shows the main result.

Theorem 2. *For every pair of nodes in an RGG, there is a path in the CHSG, whose length is asymptotically optimal w.h.p.*

Proof. The optimal path between any pair of nodes (u, v) separated by a distance $D(u, v)$ has at least $\lceil D(u, v)/r \rceil$ edges. If $\log \ell$ is also an asymptotic lower bound on the length of such a path w.h.p., then $(D(u, v)/r + \log \ell)/2$ is also an asymptotic lower bound, and the result proved in Lemma 4 is a constant factor approximation. It remains to show that $\log \ell$ is an asymptotic lower bound on the length of an optimal path in a constant-degree random geometric graph w.h.p.

In a δ -regular graph, the expected distance between any pair of nodes randomly chosen is at least $\log_{\delta-1} n$. A $\Theta(1)$ degree random geometric graph is a subgraph of some regular graph. Hence, in a $\Theta(1)$ degree random geometric graph, the expected distance between any pair of nodes randomly chosen is in $\Omega(\log n)$. The previous result is true w.h.p. because for some constant β

$$\begin{aligned} Pr(D(u, v) < \beta \log n) &\leq \frac{1}{n-1} \sum_{i=0}^{\beta \log n - 2} \delta(\delta - 1)^i \\ &\in O(n^{-\gamma}) . \end{aligned}$$

Using the union bound, under the parameter conditions of Lemma 4, $D(u, v) \in \Omega(\log \ell)$ for all pairs of nodes (u, v) w.h.p.

4 Distributed Algorithm

In this section we describe how to distributedly implement the steps of the Disk Covering Scheme for network formation. Step 2 of the Disk Covering Scheme can be achieved distributedly by means of a Maximal Independent Set (MIS) computation with nodes transmitting in a range of $ar/2$. An algorithm to compute an MIS in a weak model is presented in [8]. This algorithm can be tailored to our setting and can be shown to have a running time of $O(\log^2 \ell)$. The details are omitted here for the sake of brevity.

Steps 3 and 4 of the Disk Covering Scheme require uncolliding transmissions of each bridge in a radius of r and $br/2$ respectively. All nodes assigned to the same bridge will participate in a common spanner construction. Additionally bridge nodes must set up links with all bridge nodes at a distance of at most r . A $O(\log \ell)$ time algorithm to achieve these types of uncolliding transmissions exists and is easy to demonstrate. The details are omitted here for brevity.

Spanner Construction and Its Analysis. The implementation of step 5 of the disk covering scheme is described in this section. After nodes are covered by one or more bridges, they have to connect locally to neighboring nodes covered by the same bridge, i.e. within the same disk. Nodes can be covered by more than one bridge. Hence, interference of transmissions not only from the local disk but also from neighboring disks must be taken into account. However, any node is covered by at most a constant number of disks as explained in Lemma 2, then the number of interfering transmissions with respect to the local disk is increased only by a constant factor that we fold into the constants involved in this analysis.

Since the diameter is not constrained, we adopt the simplest topology, i.e. a linked list. In order to minimize the running time, we avoid handshaking among nodes and all the construction is done by broadcasting. We start with every node choosing an integer index uniformly at random from the interval $[1, \ell]$. Since there are $O(\log \ell)$ nodes within the same range w.h.p. as shown before, no two nodes choose the same index w.h.p. Then, every node forever transmits its

index with probability $1/\beta_4 \log \ell$ where β_4 is a constant. If a neighbor's index is received, links to the predecessor and successor are updated if necessary.

Lemma 5. *Any node running the spanner algorithm joins the spanner within $O(\log^2 \ell)$ steps w.h.p.*

Proof. The proof uses a simple balls and bins argument and is omitted in this extended abstract for the purpose of brevity.

A Small-Diameter Spanner. In the previous construction, the distance between any two nodes is at most the number of nodes within the disk, i.e. $O(\log \ell)$. Although a diameter of $\Theta(\log \ell)$ for the disk spanner is optimal (Theorem 2) for a $\Theta(1)$ degree random geometric graph, a $\Theta(1)$ degree spanner with diameter $o(\log \log \ell)$ is also possible.

The structure we utilize, is popularly known as a *butterfly network* [12]. Butterfly networks are used in many parallel computers to provide paths of length $\log m$ connecting m inputs to m outputs. In our case, all nodes have the same role and a message between any pair of nodes can be sent in $O(\log m)$ hops. Then, given that there are $\Theta(\log \ell)$ nodes in any disk, the diameter obtained is $o(\log \log \ell)$. Notice that, once unique consecutive labels are assigned to all nodes, each node can easily compute to which nodes is connected. Then, our goal is to assign unique consecutive indexes to all nodes within the disk.

A $O(\log^2 \ell)$ distributed algorithm to construct a butterfly network exists. The details are omitted here for brevity.

5 Conclusions

The bootstrapping protocol presented in this paper builds a hop-optimal $\Theta(1)$ degree sensor network under the constraints of the WSM in $O(\log^2 \ell)$ time w.h.p.

There is a trade-off among the maximum degree, the length of the optimal path and the density given by:

There is a u, v path of $\leq \left\lceil \frac{D(u,v)}{r} \frac{c\sqrt{4+\alpha^2}}{b-a} \right\rceil - 1 + O(\log \ell)$ hops w.h.p.

The degree of any node is $\leq 3 \lceil \frac{4}{a\sqrt{3}} \rceil \left(\lceil \frac{4}{a\sqrt{3}} \rceil + 1 \right)$ w.h.p.

The density of nodes is $\frac{n}{\ell^2} > 5 \frac{4+\alpha^2}{\alpha} \left(\frac{c}{b-a} \right)^2 \frac{\ln \ell}{r^2}$.

Here $0 < a < 1$, $a < b \leq 1$, $c > 1$ and $0 < \alpha \leq 1$. The longer the edges covered, the lower the density and the smaller the number of hops in the optimal path but at the cost of a higher degree.

In our construction, only three ranges of transmission are used, namely $ar/2$, $br/2$ and r . Hence, the specific values of a and b are hardware dependent. No synchronicity assumption is needed, and neighboring disks do not need to be running the same phase of the algorithm.

Regarding failures, the distributed network formation algorithm is also a maintenance algorithm, since both bridge and non-bridge nodes keep broadcasting forever. If a bridge node fails, after some time non-bridge nodes will detect

the absence of their bridge broadcast and will restart the MIS algorithm to obtain a new bridge. On the other hand, if a non-bridge node fails, its successor and predecessor will interconnect within the next round of the spanner construction. If the butterfly network spanner is used instead and a link is lost, the butterfly network can be simply rebuilt locally from scratch.

References

1. R. Bar-Yehuda, O. Goldreich, and A. Itai. On the time-complexity of broadcast in multi-hop radio networks: An exponential gap between determinism and randomization. *Journal of Computer and System Sciences*, 45:104–126, 1992.
2. D. M. Blough, M. Leoncini, G. Resta, and P. Santi. The k-neigh protocol for symmetric topology control in ad hoc networks. In *MobiHoc*, 2003.
3. F. Ferraguto, G. Mambriani, A. Panconesi, and C. Petrioli. A new approach to device discovery and scatternet formation in bluetooth networks. In *Proc. of IPDPS*, 2004.
4. A. Goel, B. Krishnamachari, and S. Rai. Sharp thresholds for monotone properties in random geometric graphs. In *Proc. of STOC*, 2004.
5. P. Gupta and P. R. Kumar. Critical power for asymptotic connectivity in wireless networks. In *Stochastic Analysis, Control, Optimization and Applications*. Birkhauser, 1998.
6. V. S. A. Kumar, M. V. Marathe, S. Parthasarathy, and A. Srinivasan. End-to-end packet-scheduling in wireless ad-hoc networks. In *SODA*, 2004.
7. C. Law and K.-Y. Siu. A Bluetooth scatternet formation algorithm. In *Proceedings of the IEEE Symposium on Ad Hoc Wireless Networks*, November 2001.
8. T. Moscibroda and R. Wattenhofer. Maximal independent sets in radio networks. In *PODC*, 2005.
9. S. Muthukrishnan and G. Pandurangan. The bin-covering technique for thresholding random geometric graph properties. In *Proc. of ACM-SODA*, 2005.
10. K. Nakano and S. Olariu. Energy-efficient initialization protocols for radio networks with no collision detection. In *Proc. of ICPP*, 2000.
11. L. G. Roberts. Aloha packet system with and without slots and capture. *Computer Communication Review*, 5(2):28–42, 1975.
12. G. Schmidt. The butterfly parallel processor. In *Proc. of ICS*, pages 362–365, 1987.
13. K. Sohrabi, J. Gao, V. Ailawadhi, and G. J. Pottie. Protocols for self-organization of a wireless sensor network. *Personal Communications, IEEE*, 7(5):16–27, 2000.
14. W. Song, Y. Wang, X. Li, and O. Frieder. Localized algorithms for energy efficient topology in wireless ad hoc networks. In *MobiHoc*, 2004.
15. Z. Wang, R. J. Thomas, and Z. Haas. Bluenet - A new scatternet formation algorithm. In *HICSS*, 2002.
16. G. V. Zaruba, S. Basagni, and I. Chlamtac. Bluetrees - Scatternet formation to enable Bluetooth-based ad hoc networks. In *Proc. of IEEE ICC*, 2001.

Approximating Integer Quadratic Programs and MAXCUT in Subdense Graphs

Andreas Björklund

Department of Computer Science, Lund University, Box 118, 22100 Lund, Sweden

Abstract. Let A be a real symmetric $n \times n$ -matrix with eigenvalues $\lambda_1, \dots, \lambda_n$ ordered after decreasing absolute value, and b an $n \times 1$ -vector. We present an algorithm finding approximate solutions to $\min x^*(Ax+b)$ and $\max x^*(Ax+b)$ over $x \in \{-1, 1\}^n$, with an absolute error of at most $(c_1|\lambda_1| + |\lambda_{\lceil c_2 \log n \rceil}|)2n + O((\alpha n + \beta)\sqrt{n \log n})$, where α and β are the largest absolute values of the entries in A and b , respectively, for any positive constants c_1 and c_2 , in time polynomial in n .

We demonstrate that the algorithm yields a PTAS for MAXCUT in regular graphs on n vertices of degree d of $\omega(\sqrt{n \log n})$, as long as they contain $O(d^4 \log n)$ 4-cycles. The strongest previous result showed that $\Omega(n/\log n)$ average degree graphs admit a PTAS.

We also show that smooth n -variate polynomial integer programs of constant degree k , always can be approximated in polynomial time leaving an absolute error of $o(n^k)$, answering in the affirmative a suspicion of Arora, Karger, and Karpinski in STOC 1995.

1 Introduction

In a novel paper [3] Arora, Karger, and Karpinski presented a general framework for showing polynomial time approximation schemes for dense instances of some NP-hard optimization problems. In particular they showed that degree d polynomial integer programs on n variables in which each monomial of degree g has a coefficient of absolute value $O(n^{d-g})$, called *smooth* integer programs, could be approximated with an absolute error of at most $(\epsilon + o(1))n^d$ for any fixed $\epsilon > 0$ in polynomial time. In the special case of quadratic polynomials, we show how to modify their algorithm in order to reduce the error term.

Theorem 1. *Let A be a real symmetric $n \times n$ -matrix, with no entry exceeding α in absolute value, and eigenvalues $\lambda_1, \dots, \lambda_n$, ordered after decreasing absolute value, and b a column vector of length n , with no entry exceeding β in absolute value, then $\min x^*Ax + x^*b$ and $\max x^*Ax + x^*b$ over the domain $x \in \{-1, 1\}^n$, can be approximated with an absolute error of at most*

$$(c_1|\lambda_1| + |\lambda_{\lceil c_2 \log n \rceil}|) 2n + O\left((\alpha n + \beta)\sqrt{n \log n}\right)$$

for arbitrary positive constants c_1 and c_2 , in time polynomial in n .

For some hardness results on the problem, consult [2] and the references therein.

The minimisation version of the problem for zero-diagonal matrices A with only non-negative entries and b set to the zero-vector is in one-to-one correspondence with a well studied graph problem, the problem of finding a weighted MAXCUT in an undirected graph. Given a graph with positive weights on the edges, you want to find the cut dividing the vertices in two parts such that the sum of all weights of edges crossing the cut is maximised. Order the vertices 1 through n and put half of an edge ij 's weight at the entry in A on row i and column j , and the other half at row j and column i . Interpret the solution vector x such that all vertices having a 1-entry is on one side of the cut, and the rest of the vertices are on the other. Thus theorem 1 applies to the graph problem. MAXCUT is known to be MAXSNP-hard in general [13], which according to the well-known PCP-theorem [4] implies that there is a constant within which we cannot approximate the problem in polynomial time, unless $P = NP$. The best approximation guarantee is the one due to [9] which says one can always get within a factor of .8785 of the optimum with polynomial time computations. In *dense* graphs we can do much better though, as observed and explored in [1][3][6][7] and [8]. The strongest result [8] shows that in unweighted undirected graphs of average degree $\Omega(n/\log n)$, there is a polynomial time approximation scheme (PTAS), i.e. for every fixed $\epsilon > 0$ there is a polynomial time algorithm approximating the solution with an absolute error of at most $\epsilon + o(1)$ times the optimum value.

Interestingly, all five results listed above adopt the technique of random sampling, admittedly partially encouraged by the success story on property testing sample size [10]. Our main technical contribution is to replace the machinery of random sampling for a simple utilisation of a structure implied by linear algebra. We show that essentially equally strong results may be derived, and in addition, we gain further insights by relating approximation success to the eigenvalue spectrum of the graph. To give an example we show that when the number of cycles on exactly four vertices is not too large, MAXCUT can be approximated within any constant efficiently also in subdense graphs.

Theorem 2. *There is a PTAS for MAXCUT on d -regular unweighted undirected graphs on n vertices, containing $O(d^4 \log n)$ simple 4-cycles, for d of $\omega(\sqrt{n \log n})$.*

The graph family covered in theorem 2 includes *all* regular graphs of degree $\Omega(n/\log n)$, since a d -regular graph on n vertices has $O(d^3 n)$ 4-cycles. This is the same class of regular graphs captured by [8], but using radically different methods. A most natural question to ask is whether the technique of random sampling proven so fruitful in the past can somehow be combined with ours to yield even stronger results, or why not.

An easy extension of our argument shows that the algorithm enables polynomial time approximation of smooth integer programs of constant degree k on n variables, with an absolute error of $o(n^k)$. [3] was able to bound the error by $(\epsilon + o(1))n^k$ for any fixed $\epsilon > 0$, but suspected that the error could be diminished

by finding an alternative for exhaustive random sampling. This is confirmed by the following result.

Theorem 3. *For every smooth integer program of constant degree k on n variables from $\{-1, 1\}$, there is a polynomial time approximation algorithm finding a solution which differs at most*

$$(\epsilon + o(1)) n^k \sqrt{\frac{\log \log n}{\log n}}$$

from the optimum in absolute value, for every fixed $\epsilon > 0$.

Many hard combinatorial problems can be expressed as smooth integer programs as described in [3], leading to results on their approximability on dense instances. To mention one problem in particular, theorem 3 improves on the approximability of dense MAX- k -SAT.

The rest of the paper is organized as follows. Section 2 describes the idea in brief terms. Section 3 and 4 presents some mathematical background used in the algorithm. Section 5 explains the algorithm, and section 6 analyses it. Finally, section 7 provides the proofs of theorem 2 and theorem 3.

2 The Approach

Let A be a real symmetric $n \times n$ -matrix and b a column vector of length n , and consider

$$\min_{x \in \{-1, 1\}^n} x^* A x + x^* b$$

where x is a column vector and $*$ denotes transpose. The idea is to find a low rank approximation UV^* of A where U and V are $n \times m$ -matrices for $m \ll n$ such that $|x^*(A - UV^*)x|$ is small for all $x \in \{-1, 1\}^n$. Then it is sufficient to approximate

$$\min_{x \in \{-1, 1\}^n} x^* UV^* x + x^* b \tag{1}$$

to find a good solution candidate. The latter problem is found by techniques similar to the ones outlined in [3]. Guess an estimate W of $V^* x_{opt}$, where x_{opt} is an optimal solution to the original problem, and solve the relaxed linear program

$$\begin{aligned} \min_x \quad & x^*(UW + b) \\ \text{s.t.} \quad & \\ & V^* x \in \mathcal{P} \\ & -1 \leq x_i \leq 1 \end{aligned}$$

where \mathcal{P} is a polytope capturing the vicinity of W , and apply randomized rounding of the solution x into a $\{-1, 1\}$ -vector.

3 Some Facts from Linear Algebra

An eigenvalue, eigenvector pair (λ_i, z_i) for an $n \times n$ -matrix A , consists of a scalar λ_i and a column vector z_i of unit Euclidean length such that $Az_i = \lambda_i z_i$. For real symmetric matrices A all n eigenvalues are real and the corresponding eigenvectors may always be chosen real and orthogonal to each other (see e.g. [11]). This is the basis of the following decomposition.

Lemma 1 (The Spectral Decomposition). *Let A be a real symmetric $n \times n$ -matrix, with eigenvalues $\lambda_1, \dots, \lambda_n$, and corresponding orthogonal eigenvectors z_1, \dots, z_n , then*

$$A = \sum_{i=1}^n \lambda_i z_i z_i^*$$

The *trace* of a square matrix A , denoted $tr(A)$, is the sum of all the diagonal elements.

Lemma 2 (The Trace Identity). *Let A be a real symmetric $n \times n$ -matrix, with eigenvalues $\lambda_1, \dots, \lambda_n$, and k a positive integer, then*

$$tr(A^k) = \sum_{i=1}^n \lambda_i^k$$

Lemma 3 (The Rayleigh Bounds). *Let A be a real symmetric $n \times n$ -matrix, with eigenvalues $\lambda_1 \leq \dots \leq \lambda_n$, then for all x*

$$\lambda_1 x^* x \leq x^* A x \leq \lambda_n x^* x$$

4 Points on the Hypersphere

For a point $p \in R^n$, represented as a column vector, we denote its Euclidean norm by $\|p\|_2 = \sqrt{p^* p}$. Let D_n denote the surface of the n -dimensional unit hypersphere, i.e. all points p for which $\|p\|_2 = 1$. We say a finite point set $P \subset D_n$ is δ -dense, if for any $q \in D_n$ there exists $p \in P$ for which $\|q - p\|_2 \leq \delta$. Note that an equivalent formulation is to require $q^* p \geq 1 - \delta^2/2$. We prefer to give an explicit construction although somewhat smaller sets may be obtained simply by choosing points uniformly at random from D_n .

Lemma 4. *For any fixed δ , $0 < \delta < 1$ there exists a δ -dense point set in m dimensions of size at most $\sqrt{m} 2^m (\pi/\delta + 1)^m$, which can be constructed in deterministic time polynomial in the size of the set.*

Proof. We will use a recursive construction. Let $P(k, \delta/\sqrt{2})$ be a $\delta/\sqrt{2}$ -dense point set in k dimensions, and define the point set $P(2k, \delta)$ in $2k$ dimensions by all points

$$r(j, p) = \left(\sin(\sqrt{2}\delta j_1) p_1, \cos(\sqrt{2}\delta j_1) p_1, \dots, \sin(\sqrt{2}\delta j_k) p_k, \cos(\sqrt{2}\delta j_k) p_k \right)$$

for $j = (j_1, \dots, j_k) \in \{0, 1, \dots, \lfloor \sqrt{2}\pi/\delta \rfloor\}^k$, and $p = (p_1, \dots, p_k) \in P(k, \delta/\sqrt{2})$. Consider a point $q = (q_1, \dots, q_{2k}) \in D_{2k}$. We will argue that there always exists a $r(j, p)$ for which

$$q^* r(j, p) \geq 1 - \delta^2/2$$

Choose j such that $\sin(\sqrt{2}\delta j_i)q_{2i-1} + \cos(\sqrt{2}\delta j_i)q_{2i}$ is maximised for all i . For two-dimensional vectors $a^* b = \|a\|_2 \|b\|_2 \cos(\phi)$, where ϕ is the angle between the vectors a and b . We use the fact that $\cos(\phi) \geq 1 - \phi^2/2$, and that there always exists j for which $\phi \leq \delta/\sqrt{2}$ by construction, to get

$$q^* r(j, p) \geq (1 - \delta^2/4) \sum_{i=1}^k p_i \sqrt{q_{2i-1}^2 + q_{2i}^2}$$

Furthermore, since $P(k, \delta/\sqrt{2})$ is $\delta/\sqrt{2}$ -dense, there exists a p such that

$$\sum_{i=1}^k p_i \sqrt{q_{2i-1}^2 + q_{2i}^2} \geq (1 - \delta^2/4)$$

Altogether we have that there is a $r(j, p)$ for which $q^* r(j, p) \geq 1 - \delta^2/2$, and hence, that $P(2k, \delta)$ is δ -dense. A similar construction is used for $P(2k - 1, \delta)$. Simply choose all points on the format

$$\left(\sin(\sqrt{2}\delta j_1)p_1, \cos(\sqrt{2}\delta j_1)p_1, \dots, \sin(\sqrt{2}\delta j_{k-1})p_{k-1}, \cos(\sqrt{2}\delta j_{k-1})p_{k-1}, p_k \right)$$

for $j = (j_1, \dots, j_k) \in \{0, 1, \dots, \lfloor \sqrt{2}\pi/\delta \rfloor\}^k$, and $p = (p_1, \dots, p_k) \in P(k, \delta/\sqrt{2})$. To see it is δ -dense, argue along the same lines as for $P(2k, \delta)$. Finally, set $P(1, \delta) = \{-1, 1\}$, for $0 < \delta < 1$, and note that δ -density holds.

It remains to bound the size of the construction. Let $S(k, \delta)$ denote the size of $P(k, \delta)$, then

$$S(k, \delta) = \begin{cases} 2 & : k = 1 \\ \lceil \frac{\sqrt{2}\pi}{\delta} \rceil^{\lfloor k/2 \rfloor} S(\lceil k/2 \rceil, \delta/\sqrt{2}) & : k > 1 \end{cases}$$

We argue by induction. Let $\xi(k)$ denote $\log_2 k$ for positive integers k and 0 for $k=0$. If $S(k, \delta) \leq \sqrt{2}^{2k+\xi(k-1)} (\pi/\delta + 1)^k$ for all $0 < k < l$ and $0 < \delta < 1$, then

$$\begin{aligned} S(l, \delta) &= \lceil \frac{\sqrt{2}\pi}{\delta} \rceil^{\lfloor l/2 \rfloor} \sqrt{2}^{2\lceil l/2 \rceil + \xi(\lceil l/2 \rceil - 1)} \left(\frac{\sqrt{2}\pi}{\delta} + 1 \right)^{\lceil l/2 \rceil} \leq \\ &\left(\frac{\sqrt{2}\pi}{\delta} + \sqrt{2} \right)^{\lfloor l/2 \rfloor} \sqrt{2}^{2\lceil l/2 \rceil + \xi(\lceil l/2 \rceil - 1)} \left(\frac{\sqrt{2}\pi}{\delta} + \sqrt{2} \right)^{\lceil l/2 \rceil} \leq \\ &\sqrt{2}^{2\lceil l/2 \rceil + \xi(\lceil l/2 \rceil - 1)} \sqrt{2}^l \left(\frac{\pi}{\delta} + 1 \right)^l \leq \\ &\sqrt{2}^{2l + \xi(l-1)} \left(\frac{\pi}{\delta} + 1 \right)^l \end{aligned}$$

since $\xi(l - 1) = 1 + \xi(\lceil l/2 \rceil - 1)$ for odd integers $l > 1$ and $\xi(l - 1) \geq \xi(\lceil l/2 \rceil - 1)$ for even integers $l > 1$. The bound is easily seen to hold for $S(1, \delta)$ with $0 < \delta < 1$ and thus the result follows.

Given δ -dense point sets, it is possible to construct polytopes approximating hyperspheres.

Lemma 5. *Let $W \in R^n$ be the center of a hypersphere S of radius r , and let P be a $\sqrt{2 - 2/f}$ -dense point set in n dimensions for some $f > 1$. Define the f -smooth polytope \mathcal{P} as all points x such that $(x - W)^*p \leq r$ for all $p \in P$. Then for all $x \in \mathcal{P}$, $\|x - W\|_2 \leq fr$, and yet, all points in S are contained in the polytope.*

Proof. Consider a point $x \in \mathcal{P}$. Since P is $\sqrt{2 - 2/f}$ -dense, there exists a $p \in P$ such that $(x - W)^*p = \|x - W\|_2/f$. But the definition of the polytope says $(x - W)^*p \leq r$, and thus $\|x - W\|_2 \leq fr$. To see that S is contained in \mathcal{P} , note that for all $x \in S$ and all $p \in P$ we have $(x - W)^*p \leq \|x - W\|_2 \leq r$, according to the Cauchy-Schwarz inequality.

5 The Algorithm

We want to solve $\min_{x \in \{-1,1\}^n} x^*Ax + x^*b$. Let x_{opt} denote an optimal solution in the following. The algorithm consists of two parts. The first finds a low rank approximation of the matrix A , i.e. a factoring $UV^* \approx A$ where U and V are $n \times m$ -matrices over the reals. The second step of the algorithm approximates the solution to (1) by linear programming and randomized rounding.

5.1 Factoring A

Finding two real $n \times m$ -matrices U and V such that $|x^*(A - UV^*)x|$ is small for all $x \in \{-1, 1\}^n$ is accomplished by considering the spectral decomposition of A in lemma 1. Methods for obtaining the eigenvalues and eigenvectors for real symmetric matrices are outlined in [11]. The next lemma describes the construction.

Lemma 6. *Let A be a real symmetric $n \times n$ -matrix, with eigenvalues $\lambda_1, \dots, \lambda_n$, sorted after decreasing absolute value with corresponding orthogonal eigenvectors z_1, \dots, z_n . Let U be the matrix consisting of $\sqrt{|\lambda_i|}z_i$ as columns, and V of $\text{sign}(\lambda_i)\sqrt{|\lambda_i|}z_i$, for all $i \leq m$, then*

$$|x^*(A - UV^*)x| \leq |\lambda_{m+1}|x^*x$$

for all x .

Proof. First note that $(A - UV^*) = \sum_{m < i \leq n} \lambda_i z_i z_i^*$ according to lemma 1. This matrix is symmetric and has no eigenvalues of absolute value larger than $|\lambda_{m+1}|$ since its eigenvalues are λ_i for $i > m$, and zero with multiplicity m . Hence, via lemma 3, the result follows.

5.2 Relaxation to Linear Programming

We want to solve

$$\min_{x \in \{-1,1\}^n} x^*UV^*x + x^*b$$

If we knew an approximation W of V^*x_{opt} such that $\|V^*x_{opt} - W\|_2 \leq e$, where e is a positive constant, then we could solve the linear program

$$\begin{aligned} \min x^*(UW + b) \\ \text{s.t.} \\ V^*x \in \mathcal{P} \\ -1 \leq x \leq 1 \end{aligned} \tag{2}$$

where \mathcal{P} is an f -smooth polytope from lemma 5 approximating the hypersphere of radius e centered at W . Note that a solution \hat{x} to the linear program fulfills

$$\hat{x}^*(UW + b) \leq x_{opt}^*(UW + b) \tag{3}$$

since x_{opt} is a feasible solution. Furthermore, for any feasible x we have

$$|x^*UW - x^*UV^*x| = |x^*U(W - V^*x)| \leq fe\|x^*U\|_2 \tag{4}$$

according to the Cauchy-Schwarz inequality. We need to bound the Euclidean length of the vectors x^*U and V^*x .

Lemma 7. *Let A be a real symmetric $n \times n$ -matrix with λ_1 the eigenvalue of largest absolute value, and let U and V be as in lemma 6, then*

$$\max_{x \in [-1,1]^n} \|x^*U\|_2 = \max_{x \in [-1,1]^n} \|V^*x\|_2 \leq \sqrt{|\lambda_1|n}$$

Proof. Note that $UU^* = VV^*$ is symmetric and has zeros, and $|\lambda_i|$ for $i \leq m$ as eigenvalues, where λ_i is the i th largest absolute eigenvalue of A . Thus by lemma 3, the Euclidean distance is bounded from above by $\sqrt{|\lambda_1|n}$.

5.3 Estimating V^*x_{opt}

To guess an estimate W of V^*x_{opt} we will once again use dense point sets from lemma 4.

Lemma 8. *Let P be a g -dense point set, and S a hypersphere of radius r centered at origo, both in n dimensions. Consider the set R for a small positive constant $h < 1$, consisting of the points $jhr \cdot p$ for each $j \in \{1, 2, \dots, \lceil 1/h \rceil\}$, and $p \in P$. Then, for every $q \in S$, there exists $t \in R$ such that $\|q - t\|_2 \leq \sqrt{g^2 + h^2}r$.*

Proof. Since P is g -dense, there exists $p \in P$ such that $q^*p \geq (1 - g^2/2)\|q\|_2$. Let j be the smallest positive integer for which $jhr \geq \|q\|_2$ and set $t = jhr \cdot p$ and $\Delta = \|t\|_2 - \|q\|_2$. Then,

$$\begin{aligned} \|q - t\|_2^2 &= \|q\|_2^2 - 2q^*t + \|t\|_2^2 \leq \\ \|q\|_2^2 - (2 - g^2)\|q\|_2\|t\|_2 + \|t\|_2^2 &= \\ \|t\|_2^2 - 2\Delta\|t\|_2 + \Delta^2 - (2 - g^2)(\|t\|_2 - \Delta)\|t\|_2 + \|t_2\|_2^2 &= \\ \Delta^2 + g^2(\|t\|_2 - \Delta)\|t\|_2 \leq \\ (h^2 + g^2)r^2 \end{aligned}$$

after noting that $\|t\|_2 - \Delta \leq \|t\|_2 \leq r$ and $|\Delta| \leq hr$.

We will construct such a set R for $r = \sqrt{|\lambda_1|n}$, since this is the maximum value of $\|V^*x_{opt}\|_2$ according to lemma 7, and solve the linear program in (2) for all points $W \in R$. Lemma 8 guarantees that one of the points is a close enough approximation, since it leaves

$$e = \sqrt{(g^2 + h^2)|\lambda_1|n} \tag{5}$$

5.4 Randomized Rounding

The technique of randomized rounding presented by Raghavan and Thompson [15] shows us how to go back from the relaxed LP-formulation to the original problem without inducing too much error.

Lemma 9. *Let $x = \{x_i\}$ be a vector of n variables, $0 \leq x_i \leq 1$, that satisfies a certain linear constraint $a^*x = b$, where each entry of a has absolute value less than or equal to s . Construct y_i randomly by setting $y_i = 1$ with probability x_i and 0 otherwise. Then $|a^*y - b|$ is $O(s\sqrt{n \log n})$ with probability at least $1 - n^{-c}$ for any fixed positive constant c ,*

Let x be the best solution to all the linear programs (2) run for different choices of W . Obtain $y \in \{-1, 1\}^n$ by setting $y_i = 1$ with probability $(1 + x_i)/2$, and $y_i = -1$ with probability $(1 - x_i)/2$. Denote by α the largest absolute value in A , and by β the largest absolute value in b . By lemma 9 with high probability each element in $Ax - Ay$ is $O(\alpha\sqrt{n \log n})$ in absolute value, and $y^*b - x^*b$ is $O(\beta\sqrt{n \log n})$ in absolute value. Thus, remembering that A is symmetric,

$$\begin{aligned} y^*Ay + y^*b &= x^*Ax + x^*b + (y^*A - x^*A)x + y^*(Ay - Ax) + (y^*b - x^*b) \leq \\ x^*Ax + x^*b + O((\alpha n + \beta)\sqrt{n \log n}) \end{aligned} \tag{6}$$

A derandomization of lemma 9 can be done by the method of conditional expectations as described in [14].

6 The Assembly Line

The approximation guarantee and running time of the algorithm described in the previous section depends on several tunable parameters to be fixed. They include,

- m , the column dimension of the matrices U and V in section 5.1.
- f , the smoothness of the polytope in section 5.2.
- g and h , the density parameters for the set R in section 5.3.

We begin by focusing on the approximation guarantee. Let y be a found solution, then

$$y^*Ay + y^*b \leq x^*Ax + x^*b + O\left((\alpha n + \beta)\sqrt{n \log n}\right)$$

according to (6), where x is a solution to the linear program (2). By lemma 6

$$x^*Ax \leq x^*UV^*x + |\lambda_{m+1}|n$$

and by (3) and (4)

$$x^*UV^*x + x^*b \leq x^*(UW + b) + fe\|x^*U\|_2 \leq x_{opt}^*(U\hat{W} + b) + fe\|x^*U\|_2$$

where W is the close approximation of V^*x , and \hat{W} is one for V^*x_{opt} . The inequality above holds since x is the best solution to all the linear programs run. From (4), and lemma 6 we have

$$x_{opt}^*U\hat{W} \leq x_{opt}^*Ax_{opt} + fe\|x_{opt}^*U\|_2 + |\lambda_{m+1}|n$$

Altogether, with the aid of lemma 7 and (5), we arrive at

$$y^*Ay + y^*b \leq x_{opt}^*Ax_{opt} + x_{opt}^*b + \tag{7}$$

$$2f\sqrt{g^2 + h^2}|\lambda_1|n + 2|\lambda_{m+1}|n + O\left((\alpha n + \beta)\sqrt{n \log n}\right)$$

We now turn to the running time, aiming at polynomial time computations. Obtaining the decomposition UV^* in lemma 6 is a cubic time task, see e.g. [11]. The linear program (2) will be run $|R|$ times and this is fine as long as the set R , as well as the linear program are of size polynomial in n , since polynomial time algorithms for linear programming exist, see e.g. [12]. The size of the f -smooth polytope \mathcal{P} , and hence the larger part of the linear program (2) is

$$\sqrt{m} \left(\frac{\sqrt{2\pi}}{\sqrt{1 - 1/f}} + 2 \right)^m$$

according to lemma 4 and lemma 5. Letting f be constant leaves a polynomial expression in n for m of $O(\log n)$. The set R of guesses W is of size

$$\lceil \frac{1}{h} \rceil \sqrt{m} \left(\frac{2\pi}{g} + 2 \right)^m \tag{8}$$

as seen from lemma 4 and lemma 8. The expression is polynomial in n for g and h fixed positive constants, and $m = c_2 \log n$ for any positive constant c_2 . Finally, note that the derandomization of the randomized rounding technique in [14] runs in polynomial time.

To finish the proof of theorem 1 we observe that finding the maximum is just the minimum in disguise: $\max x^*Ax + x^*b$ is equal to $\min x^*(-A)x + x^*(-b)$, over every domain.

7 Applications

We will give two examples of applications of the algorithm. The first concerns the MAXCUT problem. Let A be the adjacency matrix of the input graph $G = (V, E)$, with $n = |V|$, i.e. the symmetric $n \times n$ -matrix with ones at all entries $ij \in E$, and zeros elsewhere. The maximum cut, denoted $\phi(G)$, can be expressed through

$$\phi(G) = \frac{|E|}{2} - \frac{1}{4} \min_{x \in \{-1,1\}^n} x^* Ax$$

We will use theorem 1 to see that MAXCUT in subdense regular graphs without too many 4-cycles admits a PTAS.

Proof. (of theorem 2) Let $\lambda_1, \dots, \lambda_n$ be the eigenvalues of A , sorted after decreasing absolute value. We know that $\lambda_1 = d$, the degree of the graph, and that $tr(A^4)$ counts the number of closed walks visiting four vertices in the graph, see e.g. [5]. There are two types of walks, those who cross and those who don't. Walks of the first kind are easily counted in a regular graph: they amount to $2d^2n - dn$. Thus $tr(A^4) = 2d^2n - dn + 8C_4$, where C_4 is the number of non-crossing 4-cycles. The coefficient 8 reflects the fact that each cycle is counted once for each choice of start vertex, and in both directions. From lemma 2, we may conclude that $|\lambda_m| \leq (tr(A^4)/m)^{1/4}$ for $1 \leq m \leq n$. We set $m = c_2 \log n$ for some constant c_2 , since this is the value appearing in theorem 1. Now observe that for d of $\omega(\sqrt{n})$, a bound of $O(d^4 \log n)$ on C_4 implies $|\lambda_m| \leq cd$ where c is an arbitrary small positive constant inversely proportional to the 4th root of c_2 . Thus, for increasingly small c_1 and large c_2 we may bound the absolute error in theorem 1 as an arbitrarily small fraction of $|E|$ plus a term of $O(n^{1.5} \sqrt{\log n})$. The maximum cut is always at least $|E|/2$ as seen by an averaging argument, and thus as long as d is $\omega(\sqrt{n \log n})$, we meet the definition of a PTAS.

Secondly, through a slight modification of theorem 1, we note that we can always find an approximation of a smooth integer quadratic program in polynomial time which differs from the optimum only by a subquadratic term.

Proof. (of theorem 3) Consider a smooth quadratic program $\min x^* Ax + x^* b$. Let $\lambda_1, \dots, \lambda_n$ be the eigenvalues of A , sorted after decreasing absolute value. We know that $|\lambda_1|$ is $O(n)$, and that $tr(A^2)$ is $O(n^2)$ since every entry in A has absolute value $O(1)$. By lemma 2, we see that $|\lambda_m| \leq \sqrt{tr(A^2)/m}$ for $1 \leq m \leq n$. Let m be proportional to $\log n / \log \log n$ to obtain $|\lambda_m| \leq cn \sqrt{\log \log n / \log n}$, for some positive integer c . Let g and h be proportional to $1/\sqrt{\log n}$, and note by (8) that the running time still is polynomial in n for these values of g, h , and m , and that (7) fulfills the claimed error bound. For smooth integer programs of degree $k > 2$, we use the techniques of [3] to reduce a degree k program to one of degree $k - 1$, with an error boost bounded by a factor of n .

Acknowledgements

I thank Thore Husfeldt, Andrzej Lingas, and Mats Petter Pettersson for interesting and motivating discussions on the subject.

References

1. N. Alon, W. Fernandez de la Vega, R. Kannan, and M. Karpinski, "Random Sampling and Approximation of MAX-CSP Problems", Proc. 34th STOC, ACM, 534-543, 2002. The full paper can be found in Technical Report TR01-100, ECCC, 2001.
2. S. Arora, E. Berger, E. Hazan, G. Kindler, and M. Safra, "On Non-Approximability for Quadratic Programs", Technical Report TR05-58, ECCC, 2005. To appear at Proc. 46th FOCS, IEEE, 2005.
3. S. Arora, D. Karger, and M. Karpinski, "Polynomial time approximation schemes for dense instances of NP-hard problems", Proc. 27th STOC, ACM, 284-293, 1995.
4. S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy, "Proof verification and hardness of approximation problems", Proc. 33rd FOCS, IEEE, 14-23, 1992.
5. N. Biggs, "Algebraic Graph Theory", Cambridge University Press, ISBN 0-521-45897-8, 1996.
6. W. Fernandez de la Vega, "MAX-CUT has a Randomized Approximation Scheme in Dense Graphs", Random Structures and Algorithms, Vol. 8. No. 3, 187-198, 1996.
7. W. Fernandez de la Vega and M. Karpinski, "Polynomial time approximation of dense weighted instances of MAX-CUT", Random Structures and Algorithms, Vol. 16. No. 4, 314-332, 2000.
8. W. Fernandez de la Vega and M. Karpinski, "A Polynomial Time Approximation Scheme for Subdense MAX-CUT", Technical Report TR02-044, ECCC, 2002.
9. M. X. Goemans and D. P. Williamson, "Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming", J. ACM 42, 1115-1145, 1995.
10. O. Goldreich, S. Goldwasser, and D. Ron, "Property Testing and its Connection to Learning and Approximation", Proc. 37th FOCS, IEEE, 339-348, 1996. The full paper can be found in J. ACM 45, 653-750, 1998.
11. G. H. Golub and C. F. Van Loan. "Matrix Computations", Third Edition, The John Hopkins Universal Press, ISBN 0-8018-5414-8, 1996.
12. H. Karloff, "Linear Programming", Birkhuser Boston, ISBN 3-7643-3561-0, 1991.
13. C. H. Papadimitriou and M. Yannakakis, "Optimization, approximation, and complexity classes", J. Comput. System Sci. 43, 425-440, 1991.
14. P. Raghavan, "Probabilistic construction of deterministic algorithms: Approximate packing integer programs", J. Comput. System Sci. 37(2):130-143, 1988.
15. P. Raghavan and C. Thompson, "Randomized Rounding: a technique for provably good algorithms and algorithmic proofs", Combinatorica, 7:365-374, 1987.

A Cutting Planes Algorithm Based Upon a Semidefinite Relaxation for the Quadratic Assignment Problem

Alain Faye and Frédéric Roupin

CEDRIC, CNAM-Institut d'Informatique d'Entreprise,
18 allée Jean Rostand 91025 Evry cedex, France
{fayea, roupin}@iie.cnam.fr

Abstract. We present a cutting planes algorithm for the Quadratic Assignment Problem based upon a semidefinite relaxation, and we report experiments for classical instances. Our lower bound is compared with the ones obtained by linear and semidefinite approaches. Our tests show that the cuts we use (originally proposed for a linear approach) allow to improve significantly on the bounds obtained by the other approaches. Moreover, this is achieved within a moderate additional computing effort, and even in a shorter total time sometimes. Indeed, thanks to the strong tailing off effect of the SDP solver we have used (SB), we obtain in a reasonable time an approximate solution which is suitable to generate efficient cutting planes which speed up the convergence of SB.

1 Introduction

The Quadratic Assignment Problem (QAP) is one of the most challenging classical combinatorial problems. It is a model for many industrial applications [5]. The QAP has been intensively studied and many approaches have been proposed to solve it (e.g. [1,3,18]). Recently, semidefinite programming (SDP) has proved to be a powerful tool to obtain tight lower bounds for the QAP (e.g. [13,17,19,20]). The standard 0 – 1 quadratic formulation of the QAP is:

$$(QAP) \begin{cases} \text{Min } \sum_{i,j,k,l} C_{ijkl} x_{ij} x_{kl} \\ \text{s.t. } \sum_{i=1}^n x_{ij} = 1 & \forall j \in \{1..n\} \\ \sum_{j=1}^n x_{ij} = 1 & \forall i \in \{1..n\} \\ x \in \{0, 1\}^{n^2} \end{cases}$$

The paper is organized as follows. In Section 2, we recall the equivalence between two semidefinite relaxations of a general 0-1 quadratic program which contains linear equalities. Then we discuss some general issues about different semidefinite relaxations for the QAP and solving these SDP with the Spectral Bundle algorithm (SB) [9]. These preliminary results help us to choose the starting semidefinite relaxation for our cutting planes algorithm. In Section 3,

we describe the cuts used and give the details about the implementation of our algorithm. Numerical tests are presented in Section 4. Finally, we conclude in Section 5 and summarize our results.

2 Semidefinite Programming for the QAP

2.1 Semidefinite Programming: Preliminary Results

Let S_n be the space of symmetric real $n \times n$ matrices. The standard inner product over S_n is $(A, B) \in S_n^2 \rightarrow A \bullet B = Tr(AB) = \sum_{i=1}^n \sum_{j=1}^n A_{ij}B_{ij}$. We denote by $\mathbf{d}(A)$ the diagonal of a matrix $A \in S_n$, and we consider the set $S_n^+ = \{A : \forall z \in \mathbb{R}^n, z^T A z \geq 0\} \subset S_n$ of semidefinite positive matrices. For $A \in S_n^+$ we shall also write that $A \succeq 0$. A semidefinite program can be defined as the maximisation of a linear function of $X \in S_n^+$ subject to linear constraints:

$$(SDP) \begin{cases} \text{Max } A_0 \bullet X \\ \text{s.t. } A_i \bullet X = c_i \quad i = 1, \dots, m \\ X \succeq 0 \end{cases} \quad (DSDP) \begin{cases} \text{Min } c^T y \\ \text{s.t. } \sum_{i=1}^m y_i A_i - A_0 \succeq 0 \\ y \in \mathbb{R}^m \end{cases}$$

where $c \in \mathbb{R}^m$, and A_i ($i \in \{0, \dots, m\}$) are in S_n . $(DSDP)$ is the dual program of (SDP) . Semidefinite programming has proven to be an efficient approach to solve (or approximate) hard combinatorial problems, especially for problems that can be stated as (Q) , a general 0-1 quadratic program:

$$(Q) \min x^T Q_0 x + d_0^T x \text{ s.t. } \begin{cases} x^T Q_i x + d_i^T x = (\text{or } \leq) a_i \quad i \in \{1, \dots, p\} \\ Ax = b \\ x \in \{0, 1\}^n \end{cases}$$

where A is a real $m \times n$ matrix. (SDP_B) , the basic semidefinite relaxation of (Q) is equivalent to the dual of the total Lagrangian relaxation of (Q) when $x \in \{0, 1\}^n$ is written as $x_i^2 = x_i$ (for all $i \in \{1, \dots, n\}$) [14].

$$(SDP_B) \min_{x \in \mathbb{R}^n} Q_0 \bullet X + d_0^T x \text{ s.t. } \begin{cases} Q_i \bullet X + d_i^T x = (\text{or } \leq) a_i \quad i \in \{1, \dots, p\} \\ Ax = b \\ \begin{bmatrix} 1 & x^T \\ x & X \end{bmatrix} \succeq 0; \mathbf{d}(X) = x \end{cases}$$

Instead of simply keeping the linear constraints $Ax = b$, one can obtain tighter bounds by using particular treatments of “ $Ax = b$ ” (e.g. [14,19]). In particular, one can consider (SDP_P) by adding to (SDP_B) “ $\sum_{j=1}^n A_{kj} X_{ij} = b_k x_i$ for $i \in \{1, \dots, n\}$ and $k \in \{1, \dots, m\}$ ”, and (SDP_S) by adding to (SDP_B) “ $A_k A_k^T \bullet X = b_k^2$ for $k \in \{1, \dots, m\}$ ”. The semidefinite relaxations (SDP_S) and (SDP_P) are equivalent considering the results presented in [6,14,19]. In fact, they are both formulations of the dual of a partial Lagrangian relaxation of (Q) where the constraints $Ax = b$ are not relaxed. Nevertheless, as noticed in [6], the numerical solving process behavior of (SDP_S) and (SDP_Q) depend on the solver used. In the next section, we discuss such issues in order to choose the semidefinite relaxation used in our cutting planes algorithm.

2.2 Choosing the Starting Semidefinite Relaxation for the Cutting Planes Algorithm

Consider the following linear relaxation of (QAP) used in [18].

$$(QAPLP) \left\{ \begin{array}{l} \text{Min } \sum_{i,j,k,l} C_{ijkl} X_{ijkl} \\ \text{s.t. } (A_1) \sum_{i=1}^n x_{ij} = 1 \quad \forall j \in \{1..n\} \\ (A_2) \sum_{j=1}^n x_{ij} = 1 \quad \forall i \in \{1..n\} \\ (E_1) \sum_{i=1}^n X_{ijkl} = x_{kl} \quad \forall j, k, l \in \{1..n\} \quad l \neq j \\ (E_2) \sum_{j=1}^n X_{ijkl} = x_{kl} \quad \forall i, k, l \in \{1..n\} \quad k \neq i \\ (I_0) X_{ijkl} \geq 0 \quad \forall i, j, k, l \in \{1..n\} - J \end{array} \right.$$

where $J = \{(i, j, k, l) : (i = k \text{ and } j \neq l) \text{ or } (j = l \text{ and } i \neq k)\}$. In [19], an algorithm is proposed to build mechanically semidefinite relaxations for a bivalent quadratic program (Q) from any linear relaxation of (Q) . By applying it to $(QAPLP)$ one gets:

$$(SDP_0) \left\{ \begin{array}{l} \text{Min } \sum_{i,j,k,l} C_{ijkl} X_{ijkl} \\ \text{s.t. } (S_1) \sum_{i=1}^n x_{ij} = 1 \quad \forall j \in \{1..n\} \\ (R_1) \sum_{i=1}^n \sum_{k=1}^n X_{ijkj} = 1 \quad \forall j \in \{1..n\} \\ (S_2) \sum_{j=1}^n x_{ij} = 1 \quad \forall i \in \{1..n\} \\ (R_2) \sum_{j=1}^n \sum_{k=1}^n X_{ijik} = 1 \quad \forall i \in \{1..n\} \\ (P_1) \sum_{i=1}^n X_{ijkl} = x_{kl} \quad \forall j, k, l \in \{1..n\} \\ (P_2) \sum_{j=1}^n X_{ijkl} = x_{kl} \quad \forall i, k, l \in \{1..n\} \\ (P_0) X_{ijkl} \geq 0 \quad \forall i, j, k, l \in \{1..n\} \\ \begin{bmatrix} 1 & x^T \\ x & X \end{bmatrix} \succeq 0; \mathbf{d}(X) = x \end{array} \right.$$

First, the constraints (E_1) , (E_2) and (I_0) are copied respectively to (P_1) , (P_2) and (P_0) . Second, the constraints (S_1) , (R_1) , (S_2) and (R_2) are associated to (A_1) and (A_2) . Note that constraints (P_0) , (P_1) , (P_2) and $\mathbf{d}(X) = x$ imply $X_{ijkl} = 0$ for $(i, j, k, l) \in J$ (these constraints are also implied by (P_0) , (S_1) , (S_2) , (R_1) , (R_2) and $\mathbf{d}(X) = x$). It is easy to verify that (SDP_0) is tighter than $(QAPLP)$. This is a general property of the algorithm used to build the SDP relaxations from the linear ones [19]. The results recalled in Section 2.1 imply that one can remove (R_1) and (R_2) from (SDP_0) (or (P_1) and (P_2)) without modifying the optimal value. But numerical tests presented in [19] have shown that keeping them all leads experimentally to a faster convergence of the Spectral Bundle algorithm (SB) [10]. This point is illustrated in Figure 1, where (SDP_1) is (SDP_0) without constraints (R_1) and (R_2) , and (SDP_2) is (SDP_0) without constraints (P_1) and (P_2) . We have plotted the bounds for the first 500 seconds only, but, as expected, the three semidefinite programs lead to the same bound (568). Nevertheless, getting this bound by solving (SDP_2) requires more than 80 hours on the computer we have used (a Pentium IV 2.2 GHz with 1 Go Ram under Linux), whereas it takes only about one hour by using (SDP_0) .

In [20] Q. Zhao et al. propose three different semidefinite relaxations for the QAP, denoted by QAP_{R_1} , QAP_{R_2} and QAP_{R_3} in [17]. In this last paper, the

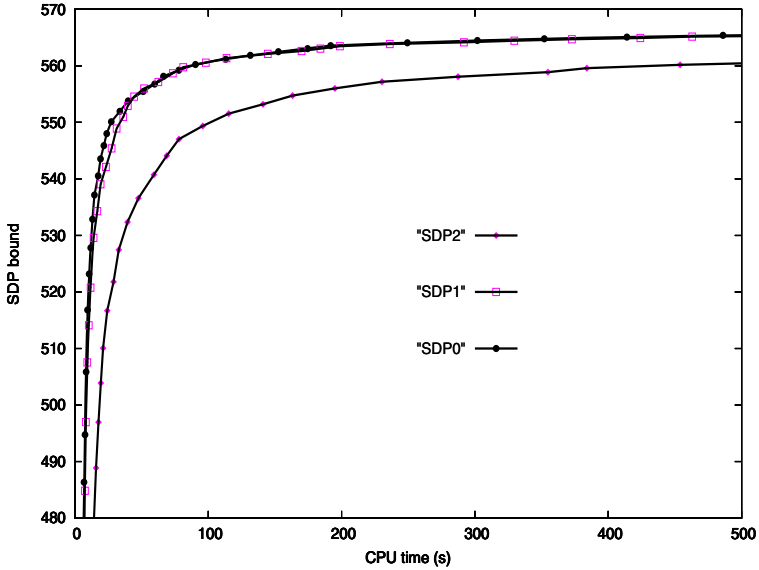


Fig. 1. Nug12 problem: bounds in dependence of CPU time when solving (SDP_0) , (SDP_1) , and (SDP_2) with the Spectral Bundle Algorithm

authors present a special version of the bundle algorithm and stop the solving process after a fixed number of iterations (equal to 300), in order to approximate the bounds proposed in [20]. The aim of this latter work is to provide good bounds for the QAP in a reasonable time, and thus which could be used in an exact method (Branch&Bound). The bounds obtained from (SDP_0) presented in [19] are better (for instance 2494 instead of 2451 for the Nug20 problem), but involve a larger number of iterations in the standard Spectral Bundle algorithm of Hemlberg and Rendl [9]. In Table 1, we give the bounds (rounded up to the next integer) obtained for some Nugxx problems [4] by using (SDP_0) in dependence of number of descent steps in the Spectral Bundle (SB) algorithm.

The last column (RS) contains the bounds presented in [17]. Some CPU times are indicated next to the bounds. These tests have been carried out on a Pentium IV 2.2 MHz computer with 1 GoBytes of RAM under Linux RedHat

Table 1. Solving (SDP_0) with the Spectral Bundle algorithm: bounds in dependence of number of descent steps

	Opt	20 steps	30 steps	40 steps	50 steps	60 steps	RS
Nug20	2570	1878(1mn8s)	2383(3mn32s)	2473(20mn2s)	2492(1h32mn)	2499(5h11mn)	2451
Nug24	3488	2171(2mn6s)	3158(7mn45s)	3343(45mn32s)	3378(3h53mn)	3389(13h34mn)	3310
Nug27	5234	3087(4mn13s)	4600(11mn6s)	4951(41mn22s)	5066(2h56mn)	5103(15h16mn)	4965
Nug30	6124	2186(3mn58s)	4471(12mn4s)	5537(33mn58s)	5799(1h54mn)	5888(8h47mn)	5803

9.0. The default parameters have been set for SB, except for one: we have scaled all the constraint matrices (option “-si 1”, see [10]). These results show that when solving (SDP_0) the convergence of SB is very fast for the first iterations. Indeed, after only 50 descent steps we obtain a good approximation of the optimal value of (SDP_0) (considering the optimal values of the problems: ”Opt” column). Moreover, note that the first 20 descent steps are computed in less than 5 minutes (even for Nug30). This strong tailing off effect of SB is well-known and is also illustrated in Figure 1.

Now we discuss the choice to keep in (SDP_0) all the positivity constraints $X_{ijkl} \geq 0 \forall i, j, k, l \in \{1, \dots, n\}$, or use them as cuts. Indeed, in papers where semidefinite programming is used to obtain lower bounds for 0-1 quadratic problems (and thus especially for the QAP), the authors generally do not include all these constraints (see e.g. [7,11,13,20]), because their number can be very large (for the QAP it is $O(n^4)$). On the contrary, we have chosen to use (SDP_0) (including all the positivity constraints (P_0)) as our starting relaxation. First, the Spectral Bundle algorithm (SB) has a nice behaviour when solving (SDP_0) (see Table 1). Moreover this solver can handle a very large number of constraints (for instance, for the Nug30 problem, (SDP_0) contains 459471 constraints). Second, as proposed in [17], one may think to use the standard linearization inequalities as cuts (other than $X_{ijkl} \geq 0$), i.e. $X_{ijkl} \leq x_{kl}$, and $x_{ij} + x_{kl} - 1 \leq X_{ijkl}$ for all i, j, k, l in $\{1, \dots, n\}$. The first ones are useless when using (SDP_0) , since constraints $(P_0), (P_1), (P_2)$ obviously imply $X_{ijkl} \leq x_{kl}$. The second set of constraints is also satisfied by any feasible solution (X, x) of (SDP_0) . Indeed, following the results presented in [2] we have $1 - x_{ij} - x_{kl} + X_{ijkl} = \sum_{r \neq j} x_{ir} - x_{kl} + X_{ijkl} = \sum_{r \neq j} x_{ir} - \sum_{r \neq j} X_{irkl} - X_{ijkl} + X_{ijkl} = \sum_{r \neq j} (x_{ir} - X_{irkl}) \geq 0$. Hence, keeping the constraints (P_0) leads to a strong SDP relaxation (including all the standard linearization constraints) that can be solved approximately by SB within a reasonable time.

3 The Cutting Planes Algorithm

3.1 Adding Cuts to Improve the Semidefinite Relaxation

Since the inequalities $X_{ijkl} \leq x_{kl}$, and $x_{ij} + x_{kl} - 1 \leq X_{ijkl}$ are already satisfied, we have considered the two sets I'_C and I'_L of cuts proposed by Blanchard et al. in [3], which have proved to be efficient for the linear program $(QAPLP)$. The two sets are defined as follows [3]:

$$I'_C : \sum_{c \in C} X_{ijlc} \leq \sum_{k \in A} X_{ijhk} + \sum_{c \in C} \sum_{b \in B, b \neq c} X_{lcb}$$

where i, h, l are distinct fixed row indices of x , j is a column indice of x , $(A, B, \{j\})$ is a partition of the index set $\{1, \dots, n\}$, and $C \subset B$.

$$I'_L : \sum_{c \in L} X_{ijcl} \leq \sum_{h \in A} X_{ijhk} + \sum_{c \in L} \sum_{b \in B, b \neq c} X_{clbk}$$

where j, k, l are distinct fixed column indices of x , i is a row indice of x , $(A, B, \{i\})$ is a partition of the index set $\{1, \dots, n\}$, and $L \subset B$. Unfortunately, (II), the associated separation problem is NP-complete. Indeed, Max-cut polynomially reduces to (II), but it is polynomially solvable when $|C| = 1$ (all these results are proved in [3]). To generate cuts with $|C| > 1$, we use the heuristic proposed in [3]. The main idea is: first, start with a cut generated with $|C| = 1$, second, add indices into C in order to obtain a cut which is more violated.

3.2 Implementation Issues

We have chosen to use the Spectral Bundle algorithm (SB) [9] which has proved to be one of the more efficient SDP solver [15]. Moreover, it is not very sensitive to a large number of constraints, and shows good initial progress (thus it allows us to generate cuts within a reasonable time). The default parameter settings of SB has been chosen expect for one: we have scaled the constraint matrices to norm one on input (see [10] for details). Recall that SB can only solve the dual of semidefinite programs with a constant trace matrix. Experiments for the QAP in [19] showed that giving to that constant the smallest possible value leads to a faster convergence of the solver SB. For (SDP_0) , thanks to the assignment constraints, one can easily verify that the trace of X equals to n . At each step of our cutting planes algorithm, the cuts are generated from an approximate solution of the current SDP. We interrupt the solving process of the SDP when the standard stopping criterion of SB is reached within a reasonable accuracy (0.01 for all the tests), or when 50 descent steps have been done. Recall that the stopping criterion of SB consists in considering that the maximal progress of the next step is small in comparison to the absolute value of the function (details are given in [10]). No more than 2000 cuts are added at each step (we add the most violated ones), and when no cut is violated by more than 0.001 the algorithm stops. The dual variables associated to the constraints are initialized as follows: for the existing constraints, we keep the current values; for the cuts generated, we choose a fixed positive value proportional to the degree of violation of the corresponding constraint. This can be seen as a simple “warm-start”. Finally, in order to have a reasonable total computing time, we do at most 5 steps in the cutting planes algorithm (2 for “easy” problems), and we stop the solving process of the last SDP if the standard stopping criterion of SB is reached within a accuracy of 10^{-5} or if 70 descent steps have been done in the Spectral Bundle algorithm.

3.3 Building a Feasible Solution for the QAP

When the cutting planes algorithms stop (based upon linear or semidefinite programming), we use the heuristic presented in [13] to get a feasible solution for (QAP) from the final matrix x' (solution of the linear or semidefinite approaches). By this way, when the solution is non integral, we obtain for several problems an optimal solution (see Section 4). Indeed we do not assume to know in advance the optimal value of the problems. The idea is to minimize $\|x' - x\|_F$ over the feasible solutions x of (QAP) , where $\|\cdot\|_F$ is the norm associated to

the trace inner product. One has $\|x' - x\|_F^2 = \|x'\|_F^2 + \|x\|_F^2 - 2Tr(x'x) = 2n^2 - 2 \sum_{i=1}^n \sum_{j=1}^n x'_{ij}x_{ij}$. This linear assignment problem can be solved easily.

4 Computational Results

In this section we present computational experiments to test the benefits of the cuts added to the semidefinite relaxation (SDP_0). All the numerical experiments have been carried out on a Pentium IV 2.2 MHz computer with 1 GoBytes of RAM under Linux RedHat 9.0. The semidefinite programs were solved by the Spectral Bundle algorithm (SB) [9,10], and the linear programs were solved by using CPLEX 9.0 (with the “baropt” function). In addition, we have used SDP_S [12], a SDP modeler that formulates automatically semidefinite relaxations following the algorithm presented in [19].

Table 2. Chrxx problems. $NLP = 5$, $NSDP = 2$, $MC = 2000$.

<i>pb</i>	<i>OPT</i>	<i>QAPLP</i>	<i>LPCUT</i>	<i>SDP</i>	<i>SDPCUT</i>	<i>RS</i>
Chr12a	9552	<u>9552</u> (9s)	<u>9552</u> (9s)	<u>9552</u> (13mn09s)	<u>9552</u> (6mn53s)	n.a.
Chr12b	9742	<u>9742</u> (9s)	<u>9742</u> (9s)	<u>9742</u> (12mn09s)	<u>9742</u> (5mn04s)	n.a.
Chr12c	11156	<u>11156</u> (9s)	<u>11156</u> (9s)	<u>11156</u> (50mn33s)	<u>11156</u> (11mn28s)	n.a.
Chr15a	9896	9514(1mn52s)	<u>9896</u> (7mn54s)	9877(1h10mn)	<u>9896</u> (1h13mn)	n.a.
Chr15b	7990	<u>7990</u> (1mn11s)	<u>7990</u> (1mn11s)	7987(39mn15s)	<u>7990</u> (27mn20s)	n.a.
Chr15c	9504	<u>9504</u> (1mn05s)	<u>9504</u> (1mn05s)	<u>9504</u> (1h13mn)	<u>9504</u> (22mn40s)	n.a.
Chr18a	11098	10759(12mn)	<u>11098</u> (1h14mn)	10985(1h59mn)	<u>11098</u> (2h40mn)	n.a.
Chr18b	1534	1534(4mn30s)	1534(4mn30s)	1497(5h14mn)	1504(1h25mn)	n.a.
Chr20a	2192	2176(21mn34s)	<u>2192</u> (1h04mn)	2097(2h27mn)	2163(4h36mn)	n.a.
Chr20b	2298	2287(14mn52s)	<u>2298</u> (48mn)	1946(37mn20s)	<u>2298</u> (9h53mn)	n.a.
Chr20c	14142	<u>14142</u> (18mn34s)	<u>14142</u> (18mn34s)	14130(5h43mn)	<u>14142</u> (2h42mn)	n.a.
Chr22a	6156	6143(1h07mn)	<u>6156</u> (2h48mn)	5223(51mn09s)	<u>6156</u> (8h13mn)	n.a.
Chr22b	6194	6181(1h14mn)	<u>6194</u> (2h36mn)	5013(46mn57s)	<u>6194</u> (12h13mn)	n.a.

In Tables 2, 4, 5, 6 and 7, *OPT* is the optimal value of the considered QAP instance [4]. *NLP* and *NSDP* are the maximum number of steps done in the cutting planes algorithms based respectively upon (*QAPLP*) and (SDP_0), and *MC* is the maximum number of cuts added at each step of the cutting planes algorithms. *QAPLP* is the bound obtained by solving (*QAPLP*). In the column *LPCUT*, we give the bounds obtained by applying the cutting planes algorithm based upon (*QAPLP*) (details are given in [3]). The bounds given in the column *SDP* are obtained by solving (SDP_0) when the standard stopping criterion of SB is reached within a accuracy of 10^{-5} or when 80 descent steps have been done. Consequently the values given in the *SDP* column are only lower bounds of the real optimal values of (SDP_0). For instance one can obtain 2198 for the Chr20b problem by solving (SDP_0), but it takes about 32 hours and requires 217 descent steps. In the column *SDPCUT*, we give the bounds obtained by

applying the cutting planes algorithm based upon (SDP_0) . The last column (RS) give the bounds presented in [17] ("n.a." means "not available"). All the bounds are rounded up to the next integer, and CPU times are indicated next to the bounds. Let us point out that the times given in the last column ($SDP CUT$) are the *total* CPU times, i.e. the times required to solve approximately all the semidefinite programs and to generate the cuts. When the heuristic presented in Section 3.3 allows to build a feasible solution with the same value (and thus optimal) for the considered problem, the bound is underlined.

Table 3. Nug20 problem: CPU times to reach several given bounds

Nug20(opt=2570)	2182	2292	2478	2503	2511
LP	38mn9s	-	-	-	-
LP CUT	38mn9s	21h48mn	-	-	-
SDP	1mn38s	2mn19s	23mn28s	14h45mn	-
SDP CUT	1mn38s	2mn19s	23mn28s	3h35mn	15h36mn

The linear approach outperforms the semidefinite one for the Chrxx problems (Table 2). This is not surprising, if one considers the quality of the bound provided by the linear relaxation ($QAPLP$) for these "easy" problems.

Table 4. Nugxx problems. $NLP = 30$, $NSDP = 5$, $MC = 2000$.

<i>pb</i>	<i>OPT</i>	<i>QAPLP</i>	<i>LPCUT</i>	<i>SDP</i>	<i>SDP CUT</i>	<i>RS</i>
Nug12	578	523(11s)	564(5h38mn)	568(2h09mn)	574(5h35mn)	557
Nug14	1014	923(52s)	994(8h38mn)	1010(12h14mn)	1014(9h05mn)	992
Nug15	1150	1041(1mn35s)	1113(11h)	1140(9h54mn)	1146(5h12mn)	1122
Nug16a	1610	1426(2mn55s)	1537(19h18mn)	1597(3h26mn)	1605(15h08mn)	1570
Nug16b	1240	1089(2mn40s)	1176(21h27mn)	1216(6h03mn)	1224(8h24mn)	1188
Nug20	2570	2182(38mn9s)	2292(21h48mn)	2503(14h45mn)	2511(15h36mn)	2451

For the others sets of problems, the results show that whatever the quality of bound one expects to obtain, the semidefinite approach is often the best choice. For instance in Table 3, it takes only 2mn19s to get 2292 for the Nug20 problem by using only (SDP_0) (SDP line), whereas linear programming (LP line) needs 38mn9s to get only 2182. The same relation exists between the "simple" semidefinite approach (SDP line) and our cutting planes algorithm ($SDP CUT$ line): for the Nug20 problem, it takes 14h45mn to get 2503 during the solving process of (SDP_0) (SDP column), whereas we get the same value in only 3h35mn ($SDP CUT$ line). Another interesting phenomenon is the speed up of the convergence of the SDP solver SB when one adds Blanchard et al's cuts to (SDP_0) . For instance in Table 5 we obtain for all the problems the optimal values by using only (SDP_0) , but it is better to apply our cutting planes algorithm.

Table 5. Hadxx problems. $NLP = 15$, $NSDP = 2$, $MC = 500$.

<i>pb</i>	<i>OPT</i>	<i>QAPLP</i>	<i>LPCUT</i>	<i>SDP</i>	<i>SDPCUT</i>	<i>RS</i>
Had12	1652	1622(14s)	<u>1652</u> (6mn05s)	<u>1652</u> (7mn49s)	<u>1652</u> (2mn22s)	1643
Had14	2724	2667(56s)	<u>2724</u> (21mn35s)	<u>2724</u> (13mn20s)	<u>2724</u> (5mn18s)	2715
Had16	3720	3561(2mn20s)	3688(4h13mn)	<u>3720</u> (3h37mn)	<u>3720</u> (14mn56s)	3699
Had18	5358	5088(9mn42s)	5228(3h49mn)	<u>5358</u> (8h55mn)	<u>5358</u> (2h11mn)	5317
Had20	6922	6579(33mn30s)	6753(14h43s)	<u>6922</u> (11h43mn)	<u>6922</u> (3h13mn)	6885

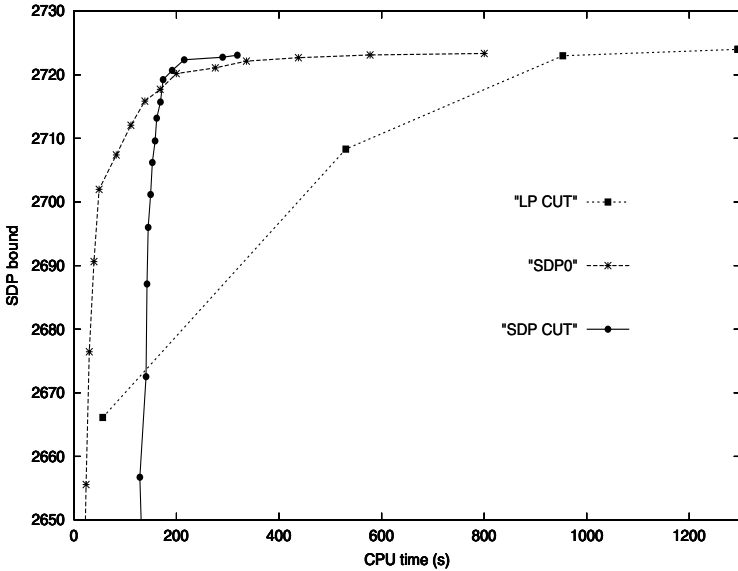


Fig. 2. Comparison of the three bounds in dependence of CPU time for the Had14 problem

Table 6. Scrxx and Rouxx problems. $NLP = 15$, $NSDP = 5$, $MC = 2000$.

<i>pb</i>	<i>OPT</i>	<i>QAPLP</i>	<i>LPCUT</i>	<i>SDP</i>	<i>SDPCUT</i>	<i>RS</i>
Scr12	31410	29828(2mn)	<u>31410</u> (43mn)	31409(10h56mn)	31409(2h01mn)	29321
Scr15	51140	49265(20mn)	<u>51140</u> (2h)	<u>51140</u> (8h03mn)	<u>51140</u> (2h06mn)	48836
Scr20	110030	95118(22mn)	99466(18h53mn)	105589(3h51mn)	105702(4h05mn)	94998
Rou12	235528	224303(1mn)	<u>235528</u> (5h)	234875(14mn56s)	235522(42mn20s)	223680
Rou15	354210	324902(8mn)	340389(23h)	347477(37mn15s)	348555(39mn26s)	333287
Rou20	725522	643364(22mn55s)	656337(19h17mn)	689229(1h17mn)	690472(1h15mn)	663833

This point is also illustrated in Figure 2 for the Had14 problem. For the semidefinite cutting planes algorithm (*SDPCUT*), we have plotted only the value of the bound in dependence of CPU time of the last semidefinite program (but translated it by 125s, the time spent to solve approximately the first SDP and to generate the cuts). We get the optimal value in about 5 minutes, whereas

Table 7. Taixx problems. $NLP = 15$, $NSDP = 5$, $MC = 2000$.

pb	OPT	$QAPLP$	$LPCUT$	SDP	$SDPCUT$	RS
Tai12a	224416	222187(12s)	224416(2mn07s)	224416(4mn47s)	224416(6mn20s)	222784
Tai15a	388214	352891(1mn16s)	367214(6h26mn)	373526(20mn42s)	376337(58mn40s)	364761
Tai17a	491812	442703(4mn28s)	454950(6h28mn)	469251(27mn22s)	475013(2h13mn)	451317
Tai20a	703482	618526(19mn54s)	629565(14h30mn)	660855(42mn52s)	667899(2h33mn)	637300

Table 8. Escxx problems. $NLP = 5$, $MC = 2000$.

pb	OPT	$QAPLP$	$LPCUT$	SDP_0 40 steps	SDP_0 60 steps	RS
Esc16a	68	48(1mn26s)	54(34mn19s)	61(10mn40s)	64(5h27mn)	59
Esc16b	292	278(1mn24s)	281(57mn35s)	284(8mn47s)	290(1h24mn)	288
Esc16c	160	118(1mn14s)	126(59mn03s)	142(12mn33s)	152(56mn38s)	142
Esc16d	16	4(1mn15s)	8(1h02mn)	10(11mn13s)	12(38mn12s)	8
Esc16e	28	14(1mn25s)	17(1h24mn)	20(6mn05s)	26(50mn54s)	23
Esc16g	26	14(1mn25s)	18(1h11mn)	21(4mn35s)	25(1h30mn)	20
Esc16h	996	704(1mn24s)	729(1h18mn)	902(7mn24s)	976(7h05mn)	970
Esc16i	14	0(1mn14s)	10(36mn37s)	7(5mn19s)	11(1h09mn)	9
Esc16j	8	2(1mn13s)	6(44mn41s)	5(2mn38s)	8(20mn17s)	7

the cutting planes approach based on linear programming requires about 21 minutes to reach the optimal value.

Nevertheless for some instances (the Escxx problems, [4]) no cuts are found or do not lead to an improvement on the bound given by (SDP_0) alone. Therefore, in Table 8 we do not give results for the cutting planes algorithm based upon (SDP_0), but only the bounds obtained during the solving process of (SDP_0) respectively after 40 and 60 descent steps in SB. We cite in the last column (RS) the bounds presented in [17]. Here, the SDP approach clearly outperforms the linear one: we get always better bounds in a shorter time. Indeed, even for the “Esc16i” problem, the value 10 is obtained by solving (SDP_0) in about 8 minutes.

5 Concluding Remarks

First, we have obtained a significative gain upon the simple SDP and the linear approaches for many instances of the QAP. In Table 9, we give the average error gaps obtained by the four approaches for the different sets of problems presented in the previous section. Second, even when (SDP_0) provides already the optimal value, adding cuts speeds up the convergence of the SDP solver (see Tables 3, 5, and Figure 2). Therefore, our cutting planes algorithm can be used to get in a shorter time a bound already obtained by a simpler approach.

However, further improvements are certainly possible. First, this is a work in progress, and more numerical tests must be carried out, especially for larger problems. Second, other sets of cuts may be used in addition of I'_C and I'_L . Third, different heuristics for the separation problem (Π) could be considered

Table 9. Average error gaps for the different sets of problems

Problems	LP	LP CUT	SDP	SDP CUT
Chr	0.7%	0%	4.4%	0.2%
Had	3.6%	1.2%	0%	0%
Nug	11.1%	4.7%	1.4%	0.8%
Scr	7.4%	3.2%	1.4%	1.3%
Rou	8.1%	6.0%	3.9%	3.7%
Tai	8.0%	5.8%	3.6%	2.9%
Esc	48.4%	27.3%	7.9%	7.9%

(see Section 3). Finally, when using the Spectral Bundle algorithm, a smarter “warm-start” for the semidefinite relaxations may speed up the solving process.

References

1. K. Anstreicher and N. Brixius. A New Bound for the Quadratic Assignment Problem Based on Convex Quadratic Programming. *Math. Prog.* 89:341-357, 2001.
2. A. Billionnet and S. Elloumi. Best reduction of the quadratic semi-assignment problem. *Discrete Applied Mathematics* 109(3):197-213, 2001.
3. A. Blanchard, S. Elloumi, A. Faye and N. Wicker. Un algorithme de génération de coupes pour le problème de l'affectation quadratique. *INFOR* 41(1):35-49, 2003.
4. R.E. Burkard, S.E. Karisch and F. Rendl, QAPLIB. A Quadratic Assignment Problem Library, *J. of Global Opt.* 10:391-403, 1997.
5. F.Çela. *The Quadratic Assignment Problem: Theory and Algorithms*. Kluwer, Massachusetts, USA, 1998.
6. A. Faye and F. Roupin. Partial Lagrangian and Semidefinite Relaxations of Quadratic Problems. In proceedings ROADEF'2005, Tours, 14-16 february 2005. Research report RC673, available at <http://cedric.cnam.fr>.
7. C. Helmberg and F. Rendl. Solving quadratic (0,1)-problems by semidefinite programs and cutting planes. *Math. Progr.* 82(3,A):291-315, 1998.
8. C. Helmberg. *Semidefinite Programming for Combinatorial Optimization*. Habilitationsschrift, TU Berlin, ZIB-report ZR-00-34, KZZI, Takustraße 7, 14195 Berlin, Germany, 2000.
9. C. Helmberg and F. Rendl. A spectral bundle method for semidefinite programming. *SIAM J. Optim.* 10(3):673-696, 2000.
10. C. Hemberg. A C++ implementation of the Spectral Bundle Method. <http://www-user.tu-chemnitz.de/~hemberg/SBmethod/>.
11. C. Helmberg. Cutting planes algorithm for large scale semidefinite relaxations. ZIB-Report ZR 01-26, KZZI, Takustraße 7, 14195 Berlin, Germany, 2001.
12. G. Delaporte, S. Jouteau and F. Roupin. SDP-S: a Tool to formulate and solve Semidefinite relaxations for Bivalent Quadratic problems. In Proceedings ROADEF 2003, Avignon 26-28 Février, 2003. <http://semidef.free.fr>.
13. S.E. Karisch. Nonlinear approaches for the quadratic assignment and graph partition problems. PhD thesis, Graz University of Technology, Graz, Austria, 1995.
14. C. Lemarechal and F. Oustry. Semidefinite relaxations and Lagrangian duality with application to combinatorial optimization. RR-3710, INRIA Rhone-Alpes, 1999.
15. Hans D. Mittelmann. An Independent Benchmarking of SDP and SOCP Solvers. *Math. Progr.* 95(2):407-430, 2003.

16. S. Poljak, F. Rendl and H. Wolkowicz. A recipe for semidefinite relaxation for (0,1)-quadratic programming. *J. of Global Opt.* 7:51-73, 1995.
17. F. Rendl and R. Sotirov. Bounds for the Quadratic Assignment Problem Using the Bundle Method. Research Report, University Of Klagenfurt, Universitaetsstrasse 65-67, Austria, 2003. Available at *Optimization-online.org*.
18. M.G.C Resende, K.G. Ramakrishnan, and Z. Drezner. Computing lower bounds for the quadratic assignment problem with an interior point algorithm for linear programming. *Operations Research* 43(5):781-791, 1995.
19. F. Roupin. From Linear to Semidefinite Programming: an Algorithm to obtain Semidefinite Relaxations for Bivalent Quadratic Problems. *J. of Comb. Opt.* 8(4):469-493, 2004.
20. Q. Zhao, S.E. Karisch, F. Rendl and H. Wolkowicz. Semidefinite programming relaxations for the quadratic assignment problem. *J. of Comb. Opt.* 2(1):71-109, 1998.

Approximation Complexity of min-max (Regret) Versions of Shortest Path, Spanning Tree, and Knapsack

Hassene Aissi, Cristina Bazgan, and Daniel Vanderpooten

LAMSADE, Université Paris-Dauphine, France
{aissi, bazgan, vdp}@lamsade.dauphine.fr

Abstract. This paper investigates, for the first time in the literature, the approximation of min-max (regret) versions of classical problems like shortest path, minimum spanning tree, and knapsack. For a bounded number of scenarios, we establish fully polynomial-time approximation schemes for the min-max versions of these problems, using relationships between multi-objective and min-max optimization. Using dynamic programming and classical trimming techniques, we construct a fully polynomial-time approximation scheme for min-max regret shortest path. We also establish a fully polynomial-time approximation scheme for min-max regret spanning tree and prove that min-max regret knapsack is not at all approximable. We also investigate the case of an unbounded number of scenarios, for which min-max and min-max regret versions of polynomial-time solvable problems usually become strongly *NP*-hard. In this setting, non-approximability results are provided for min-max (regret) versions of shortest path and spanning tree.

Keywords: min-max, min-max regret, approximation, fptas, shortest path, minimum spanning tree, knapsack.

1 Introduction

The definition of an instance of a combinatorial optimization problem requires to specify parameters, in particular objective function coefficients, which may be uncertain or imprecise. Uncertainty/imprecision can be structured through the concept of *scenario* which corresponds to an assignment of plausible values to model parameters. There exist two natural ways of describing the set of all possible scenarios. In the *interval data case*, each numerical parameter can take any value between a lower and an upper bound. In the *discrete scenario case*, the scenario set is described explicitly. In this case, that is considered in this paper, we distinguish situations where the number of scenarios is bounded by a constant from those where the number of scenarios is unbounded. Kouvelis and Yu [3]

* This work has been partially funded by grant CNRS/CGRI-FNRS number 18227. The second author was partially supported by the ACI Scurit Informatique grant-TADORNE project 2004.

proposed the min-max and min-max regret criteria, stemming from decision theory, to construct solutions hedging against parameters variations. The min-max criterion aims at constructing solutions having a good performance in the worst case. The min-max regret criterion, less conservative, aims at obtaining a solution minimizing the maximum deviation, over all possible scenarios, of the value of the solution from the optimal value of the corresponding scenario.

Complexity of the min-max and min-max regret versions has been studied extensively during the last decade. In [3], for the discrete scenario case, the complexity of min-max and min-max regret versions of several combinatorial optimization problems was studied, including shortest path, minimum spanning tree, assignment, and knapsack problems. In general, these versions are shown to be harder than the classical versions. More precisely, if the number of scenarios is unbounded, these problems become strongly *NP*-hard, even when the classical problems are solvable in polynomial time. On the other hand, for a constant number of scenarios, it was only partially known if these problems are strongly or weakly *NP*-hard. Indeed, the reductions described in [3] to prove *NP*-difficulty are based on transformations from the partition problem which is known to be weakly *NP*-hard [2]. These reductions give no indications as to the precise status of these problems. The only known weakly *NP*-hard problems are those for which there exists a pseudo-polynomial algorithm (shortest path, knapsack, minimum spanning tree on grid graphs, . . .). All these pseudo-polynomial algorithms described in [3] are based on dynamic programming.

In this paper we consider, for the first time in the literature, the approximation complexity of these versions for classical combinatorial optimization problems, focusing on three typical problems: shortest path, minimum spanning tree and knapsack.

After presenting preliminary concepts in Section 2, we investigate the existence of approximation algorithms for our reference problems when the number of scenarios is bounded by a constant (Section 3), and when it is unbounded (Section 4). The results we obtained are summarized in Table 1.

Table 1. Approximation results for min-max and min-max versions

	bounded		unbounded	
	min-max	min-max regret	min-max	min-max regret
shortest path	fptas	fptas	not $(2 - \epsilon)$ approx.	not $(2 - \epsilon)$ approx.
min spanning tree	fptas	fptas	not $(\frac{3}{2} - \epsilon)$ approx.	not $(\frac{3}{2} - \epsilon)$ approx.
knapsack	fptas	not at all approx.	not at all approx.	not at all approx.

2 Preliminaries

We consider in this paper the class \mathcal{C} of 0-1 problems with a linear objective function defined as:

$$\left\{ \begin{array}{l} \min \sum_{i=1}^n c_i x_i \quad c_i \in \mathbb{N} \\ x \in X \subset \{0, 1\}^n \end{array} \right.$$

This class encompasses a large variety of classical combinatorial problems, some of which are polynomial-time solvable (shortest path problem, minimum spanning tree, ...) and others are *NP*-difficult (knapsack, set covering, ...).

2.1 Min-max, min-max Regret Versions

Given a problem $\mathcal{P} \in \mathcal{C}$, the min-max (regret) version associated to \mathcal{P} has as input a finite set of scenarios S where each scenario $s \in S$ is represented by a vector (c_1^s, \dots, c_n^s) . We denote by $val(x, s) = \sum_{i=1}^n c_i^s x_i$ the value of solution $x \in X$ under scenario $s \in S$ and by val_s^* the optimal value in scenario s .

The min-max optimization problem corresponding to \mathcal{P} , denoted by MIN-MAX \mathcal{P} , consists of finding a solution x having the best worst case value across all scenarios, which can be stated as:

$$\min_{x \in X} \max_{s \in S} val(x, s)$$

Given a solution $x \in X$, its *regret*, $R(x, s)$, under scenario $s \in S$ is defined as $R(x, s) = val(x, s) - val_s^*$. The *maximum regret* $R_{max}(x)$ of solution x is then defined as $R_{max}(x) = \max_{s \in S} R(x, s)$.

The min-max regret optimization problem corresponding to \mathcal{P} , denoted by MIN-MAX REGRET \mathcal{P} , consists of finding a solution x minimizing the maximum regret $R_{max}(x)$ which can be stated as:

$$\min_{x \in X} R_{max}(x) = \min_{x \in X} \max_{s \in S} \{val(x, s) - val_s^*\}$$

When \mathcal{P} is a maximization problem, the max-min and min-max regret versions associated to \mathcal{P} are defined similarly.

2.2 Approximation

Let us consider an instance I , of size $|I|$, of an optimization problem and a solution x of I . We denote by $opt(I)$ the optimum value of instance I . The *performance ratio* of x is $r(x) = \max \left\{ \frac{val(x)}{opt(I)}, \frac{opt(I)}{val(x)} \right\}$, and its *error* is $\varepsilon(x) = r(x) - 1$.

For a function f , an algorithm is an $f(n)$ -*approximation algorithm* if, for any instance I of the problem, it returns a solution x such that $r(x) \leq f(|I|)$. An optimization problem has a *fully polynomial-time approximation scheme* (an *fptas*, for short) if, for every constant $\varepsilon > 0$, it admits an $(1 + \varepsilon)$ -approximation algorithm which is polynomial both in the size of the input and in $1/\varepsilon$. The set of problems having an fptas is denoted by *FPTAS*.

We recall the notion of *gap-introducing reduction* (see, e.g., [1,7]). Let \mathcal{P} be a decision problem and \mathcal{Q} a minimization problem. \mathcal{P} is gap-introducing reducible to \mathcal{Q} if there exist two functions f and α such that, given an instance I of \mathcal{P} , it is possible to construct in polynomial time an instance I' of \mathcal{Q} , such that

- if I is a positive instance then $opt(I') \leq f(I)$,
- if I is a negative instance then $opt(I') > \alpha(|I'|)f(I)$.

If \mathcal{P} is an *NP*-hard problem, and \mathcal{P} is gap-introducing reducible to \mathcal{Q} , then \mathcal{Q} is not $\alpha(n)$ -approximable if $\mathcal{P} \neq NP$.

2.3 Multi-objective Optimization

It is natural to consider scenarios as criteria (or objective functions) and to investigate relationships between min-max (regret) and multi-objective optimization, when it is usually assumed that the number of criteria is a constant.

The multi-objective version associated to $\mathcal{P} \in \mathcal{C}$, denoted by MULTI-OBJECTIVE \mathcal{P} , has for input k objective functions (or criteria) where the h th objective function has coefficients c_1^h, \dots, c_n^h . We denote by $val(x, h) = \sum_{i=1}^n c_i^h x_i$ the value of solution $x \in X$ on criterion h , and assume w.l.o.g. that all criteria are to be minimized. Given two feasible solutions x and y , we say that y dominates x if $val(y, h) \leq val(x, h)$ for $h = 1, \dots, k$ with at least one strict inequality. The problem consists of finding the set E of efficient solutions. A feasible solution x is *efficient* if there is no other feasible solution y that dominates x . In general MULTI-OBJECTIVE \mathcal{P} is intractable in the sense that it admits instances for which the size of E is exponential in the size of the input. A set F of feasible solutions is called an $f(n)$ -approximation of the set of efficient solutions if, for every efficient solution x , F contains a feasible solution y such that $val(y, h) \leq f(n)val(x, h)$ for each criterion $h = 1, \dots, k$. An algorithm is an $f(n)$ -approximation algorithm for a multi-objective problem, if for any instance I of the problem it returns an $f(n)$ -approximation of the set of efficient solutions. A multi-objective problem has an *fpas* if, for every constant $\varepsilon > 0$, there exists an $(1 + \varepsilon)$ -approximation algorithm for the set of efficient solutions which is polynomial both in the size of the input and in $1/\varepsilon$.

3 Bounded Number of Scenarios

3.1 Min-max Problems

Consider a minimization problem \mathcal{P} . It is easy to see that at least one optimal solution for MIN-MAX \mathcal{P} is necessarily an efficient solution. Indeed, if $x \in X$ dominates $y \in X$ then $\max_{s \in S} val(x, s) \leq \max_{s \in S} val(y, s)$. Therefore, we obtain an optimal solution for MIN-MAX \mathcal{P} by taking, among the efficient solutions, one that has a minimum $\max_{s \in S} val(x, s)$. Observe, however, that if MIN-MAX \mathcal{P} admits several optimal solutions, some of them may not be efficient, but at least one is efficient.

Theorem 1. *For any function $f : \mathbb{N} \rightarrow (1, \infty)$, if MULTI-OBJECTIVE \mathcal{P} has a polynomial-time $f(n)$ -approximation algorithm, then MIN-MAX \mathcal{P} has a polynomial-time $f(n)$ -approximation algorithm.*

Proof. Let F be an $f(n)$ -approximation of the set of efficient solutions. Since at least one optimal solution x^* for MIN-MAX \mathcal{P} is efficient, there exists a solution $y \in F$ such that $val(y, s) \leq f(n)val(x^*, s)$, for $s \in S$. Consider among the set F a solution z that has a minimum $\max_{s \in S} val(z, s)$. Thus, $\max_{s \in S} val(z, s) \leq \max_{s \in S} val(y, s) \leq \max_{s \in S} f(n)val(x^*, s) = f(n)opt(I)$. \square

Corollary 1. *For a bounded number of scenarios, MIN-MAX SHORTEST PATH, MIN-MAX SPANNING TREE, and MAX-MIN KNAPSACK are in FPTAS.*

Proof. For a bounded number of criteria, multi-objective versions of shortest path, minimum spanning tree, and knapsack problems, have an fptas as shown by Papadimitriou and Yannakakis in [5]. □

3.2 Min-max Regret Problems

General Results

As for min-max, at least one optimal solution for MIN-MAX REGRET \mathcal{P} is necessarily an efficient solution for MULTI-OBJECTIVE \mathcal{P} . Indeed, if $x \in X$ dominates $y \in X$ then $val(x, s) \leq val(y, s)$, for each $s \in S$, and thus $R_{max}(x) \leq R_{max}(y)$. Therefore, we obtain an optimal solution for MIN-MAX REGRET \mathcal{P} by taking, among the efficient solutions, a solution x that has a minimum $R_{max}(x)$. Unfortunately, given F an $f(n)$ -approximation of the set of efficient solutions, a solution $x \in F$ with a minimum $R_{max}(x)$ is not necessarily an $f(n)$ -approximation for the optimum value since the minimum maximum regret could be very small compared with the error that was allowed in F .

The following result deals with problems whose feasible solutions have a fixed size. In this context, we need to consider instances where some coefficients are negative but such that any feasible solution has a non-negative value. For an optimization problem \mathcal{P} , we denote by \mathcal{P}' the extension of \mathcal{P} to these instances.

Theorem 2. *For any polynomial-time solvable minimization problem \mathcal{P} whose feasible solutions have a fixed size and for any function $f : \mathbb{N} \rightarrow (1, \infty)$, if MIN-MAX \mathcal{P}' has a polynomial-time $f(n)$ -approximation algorithm, then MIN-MAX REGRET \mathcal{P} has a polynomial-time $f(n)$ -approximation algorithm.*

Proof. Let t be the size of all feasible solutions of any instance of \mathcal{P} . Consider an instance I of MIN-MAX REGRET \mathcal{P} where c_i^s is the value of coefficient c_i in scenario $s \in S$. Compute for each scenario s the value val_s^* of an optimum solution. We construct from I an instance \bar{I} of MIN-MAX \mathcal{P}' with the same number of scenarios, where $\bar{c}_i^s = c_i^s - \frac{val_s^*}{t}$. Remark that some coefficients could be negative but any feasible solution has a non-negative value. Let $\overline{val}(x, s)$ denote the value of solution x in scenario s in \bar{I} . The sets of the feasible solutions of both instances are the same and moreover, for any feasible solution x , and for any scenario $s \in S$, we have $R(x, s) = val(x, s) - val_s^* = \overline{val}(x, s)$ since any feasible solution is of size t . Therefore, an optimum solution for I is also an optimum solution for \bar{I} with $opt(\bar{I}) = opt(I)$. □

Min-Max Regret Spanning Tree

Corollary 2. *MIN-MAX REGRET SPANNING TREE, with a bounded number of scenarios, is in FPTAS.*

Proof. Using the algorithm proposed in [4] for computing determinants, one can solve the exact version of minimum spanning tree and extend the result from [5] concerning the existence of an fptas for the multi-objective version of minimum spanning tree to instances with negative coefficients but such that any feasible solution has a non-negative value. Hence, MIN-MAX REGRET (SPANNING TREE)' has an fptas. The result follows using Theorem 2. □

Min-Max Regret Shortest Path

We construct in the following an fptas for MIN-MAX REGRET SHORTEST PATH considering the multi-objective problem that consists of enumerating the paths whose regret vectors are efficient.

Theorem 3. MIN-MAX REGRET SHORTEST PATH, with a bounded number of scenarios, is in FPTAS.

Proof. We consider first the case when the graph is acyclic and we describe briefly at the end how to adapt this procedure for graphs with cycles.

Consider an instance I described by a directed acyclic graph $G = (V, A)$, where $V = \{1, \dots, n\}$ is such that if $(i, j) \in A$ then $i < j$, and a set S of k scenarios describing for each arc $(i, j) \in A$ its cost in scenario s by c_{ij}^s . Denote by c_{ij} the vector of size k formed by $c_{ij}^s, s \in S$. Let $(val_s^*)^i, s \in S, 1 \leq i \leq n$ be the value of a shortest path in graph G from 1 to i under scenario s and let $(val^*)^i$ be the vector of size k of these values $(val_s^*)^i, s \in S$.

In the following, we describe firstly a dynamic programming algorithm that computes at each stage $i, 1 \leq i \leq n$, the set R^i of efficient vectors of regrets for paths from 1 to i , for each scenario $s \in S$. Consider arc $(i, j) \in A$ and let P_i be a path in G from 1 to i of regret $r_s^i = val(P_i, s) - (val_s^*)^i, s \in S$. Denote by P_j the path constructed from P_i by adding arc (i, j) . The regret of P_j is $r_s^j = val(P_j, s) + c_{ij}^s - (val_s^*)^j = r_s^i + (val_s^*)^i + c_{ij}^s - (val_s^*)^j, s \in S$. The algorithm starts by initializing $R^1 = \{(0, \dots, 0)\}$, where $(0, \dots, 0)$ is a vector of size k and for $2 \leq j \leq n$ let

$$R^j = \text{Min}_{i \in \Gamma^{-1}(j)} \{r^i + (val^*)^i + c_{ij} - (val^*)^j : r^i \in R^i\}$$

where the operator "Min" preserves the efficient vectors.

Observe that, for $2 \leq j \leq n, R^j$, which contains all efficient regret vectors for paths from 1 to j , necessarily contains one optimal vector corresponding to a min-max regret shortest path from 1 to j . We also point out that, for this algorithm as well as for the following approximation algorithm, any path of interest can be obtained using standard bookkeeping techniques that do not affect the complexity of these algorithms.

Our approximation algorithm is a dynamic programming procedure combined with a trimming of the states depending on an accepted error $\epsilon > 0$. In this procedure, define set $T^1 = \{(0, \dots, 0)\}$, and sets U^j, T^j for $2 \leq j \leq n$ as follows

$$U^j = \cup_{i \in \Gamma^{-1}(j)} \{r^i + (val^*)^i + c_{ij} - (val^*)^j : r^i \in T^i\},$$

$$T^j = \text{Red}(U^j), \text{ where Red is an operator satisfying the following property}$$

$$\forall r \in U^j, \exists \tilde{r} \in T^j : \tilde{r} \leq r(1 + \epsilon)^{\frac{1}{n-1}}$$

where, given two vectors r', r'' of size $|S|$, we have $r' \leq r''$ if and only if $r'_s \leq r''_s, \forall s \in S$.

In the following, we prove by induction on j the proposition

$$P(j) : \forall r \in R^j, \exists \tilde{r} \in T^j \text{ such that } \tilde{r} \leq r(1 + \epsilon)^{\frac{j-1}{n-1}}$$

Obviously, proposition $P(1)$ is true. Supposing now that $P(i)$ is true for $i < j$, we show that $P(j)$ is true. Consider $r \in R^j$. Then there exists $i < j$ such that $(i, j) \in A$ and $r' \in R^i$ such that $r = r' + (val^*)^i + c_{ij} - (val^*)^j$. Since $(val^*)^i + c_{ij} \geq (val^*)^j$, we have $r \geq r'$. Using the induction hypothesis for i , there exists $\tilde{r} \in T^i$ such that $\tilde{r} \leq r'(1 + \varepsilon)^{\frac{i-1}{n-1}}$. Since $\tilde{r} \in T^i$ and $(i, j) \in A$, we have $\tilde{r} + (val^*)^i + c_{ij} - (val^*)^j \in U^j$ and, using the property satisfied by Red , there exists $\bar{r} \in T^j$ such that:

$$\begin{aligned} \bar{r} &\leq [\tilde{r} + (val^*)^i + c_{ij} - (val^*)^j](1 + \varepsilon)^{\frac{1}{n-1}} \leq \\ &\leq [r'(1 + \varepsilon)^{\frac{i-1}{n-1}} + r - r'](1 + \varepsilon)^{\frac{1}{n-1}} \leq r(1 + \varepsilon)^{\frac{i}{n-1}} \leq r(1 + \varepsilon)^{\frac{j-1}{n-1}}. \end{aligned}$$

Thus proposition $P(j)$ is true for $j = 1, \dots, n$. Obviously, there exists $r \in R^n$ such that $opt(I) = \max_{s \in S} r_s$. Applying $P(n)$ to $r \in R^n$, there exists $\tilde{r} \in T^n$ such that $\tilde{r} \leq r(1 + \varepsilon)$ and thus $\max_{s \in S} \tilde{r}_s \leq (1 + \varepsilon)opt(I)$.

We show in the following how this algorithm can be implemented in polynomial time in $|I|$ and $\frac{1}{\varepsilon}$. Let $c_{max} = \max_{(i,j) \in A, s \in S} c_{ij}^s$. For any $s \in S$ and $2 \leq j \leq n$, we have $r_s^j \leq (n - 1)c_{max}$. An operator Red can be implemented in polynomial time using the technique of interval partitioning described in Sahni [6]. The idea is to partition the domain of values, for each scenario, into subintervals such that the ratio of the extremities is $(1 + \varepsilon)^{\frac{1}{n-1}}$. Thus on each coordinate (or scenario) we have $\lceil \frac{(n-1)\log(n-1)c_{max}}{\log(1+\varepsilon)} \rceil$ subintervals. Operator Red can be implemented by selecting only one vector in each non-empty hypercube of the cartesian product of subintervals. Thus $|T^j| \leq (\frac{n \log nc_{max}}{\log(1+\varepsilon)})^k$, $2 \leq j \leq n$ and the time complexity of our algorithm is $O(n(\frac{n \log nc_{max}}{\log(1+\varepsilon)})^k)$ that is polynomial in $|I| = |A|k \log c_{max}$ and $\frac{1}{\varepsilon}$.

Consider now graphs with cycles. We can generalize the previous procedure, by defining a dynamic programming scheme with stages ℓ , $\ell = 1, \dots, n - 1$, containing sets of states R_j^ℓ which represent the set of efficient vectors of regrets for paths from 1 to j of length at most ℓ , $j = 2, \dots, n$. □

Min-Max Regret Knapsack

In this section, we prove that MIN-MAX REGRET KNAPSACK is not at all approximable even for two scenarios. For this, we use a reduction from MAXIMUM CONSTRAINED PARTITION defined in [1].

MAXIMUM CONSTRAINED PARTITION

Input: A finite set A and an integer size $s(a)$ for each $a \in A$, one element $a_0 \in A$ and a subset $B \subseteq A$.

Output: A feasible partition, i.e., a partition $(A', A \setminus A')$, $A' \subseteq A$ such that $\sum_{a \in A'} s(a) = \sum_{a \in A \setminus A'} s(a)$, with a maximum number of elements from B on the same side of the partition as a_0 .

MAXIMUM CONSTRAINED PARTITION was proved not approximable within $|A|^\varepsilon$ for some $\varepsilon > 0$ [8], but even deciding the existence of a feasible partition is NP-hard [2].

Theorem 4. *For any function $f : \mathbb{N} \rightarrow (1, \infty)$, MIN-MAX REGRET KNAPSACK is not $f(n)$ -approximable even for two scenarios, unless $P = NP$.*

Proof. We construct a reduction from MAXIMUM CONSTRAINED PARTITION to MIN-MAX REGRET KNAPSACK. Consider an instance I of MAXIMUM CONSTRAINED PARTITION characterized by a set $A = \{a_0, a_1, \dots, a_{n-1}\}$, a size $s(a)$ for each $a \in A$, and a subset $B \subseteq A$. We define an instance I' of MIN-MAX REGRET KNAPSACK as follows: the number of items is $n + 1$, the knapsack capacity is $d = \frac{1}{2} \sum_{a \in A} s(a)$, the items weights are $w_i = s(a_i)$ for $i = 0, \dots, n - 1$ and $w_n = d$. I' contains two scenarios and the values of the n items are defined as follows: $v_0^1 = n^3d$, $v_i^1 = 0$, for $i = 1, \dots, n$ and $v_i^2 = n^2s(a_i) + \delta_i$, for $i = 0, \dots, n - 1$, where $\delta_i = 1$ if $a_i \in B$ and $\delta_i = 0$ otherwise, and $v_n^2 = n^2d$.

Clearly, the optimum value in the first scenario of I' is n^3d . If I does not contain a feasible partition, the optimum value in the second scenario is n^2d . Indeed, candidate optimal solutions are either item n only with value n^2d or subsets T of items such that $\sum_{i \in T} s(a_i) < d$ with value $n^2 \sum_{i \in T} s(a_i) + \sum_{i \in T} \delta_i \leq n^2(d-1) + n < n^2d$. If I contains a feasible partition, let $opt(I)$ denote its optimal value and $(A', A \setminus A')$ an optimal partition. Suppose that $a_0 \in A'$, otherwise we exchange A' with $A \setminus A'$. In this case, the optimum value in the second scenario is $n^2d + opt(I)$ (the optimal solution is formed by items corresponding to elements from A').

If a solution x of I' does not contain a_0 then $R_{max}(x) = n^3d$. Consequently, any optimal solution x^* of I' must include item a_0 . If there exists a feasible partition in I , then we have $opt(I') = R_{max}(x^*) = 0$ and $opt(I') \geq n^2 - n$, otherwise.

Hence, MIN-MAX REGRET KNAPSACK is not approximable since otherwise, any polynomial-time approximation algorithm for this problem applied to I' could decide if I contains a feasible partition (we could even derive a maximum constrained partition by selecting for A' the set of items present in the optimal solution). □

We conclude this section giving some precisions about the complexity status of these problems. Pseudo-polynomial time algorithms were given in the case of a bounded number of scenarios for min-max (max-min) and min-max regret versions of shortest path, knapsack, and minimum spanning tree on grid graphs [3]. Our fptas for min-max and min-max regret spanning tree establish also the existence of pseudo-polynomial time algorithms for these problems. Thus min-max (max-min) and min-max regret versions of shortest path, minimum spanning tree and knapsack are weakly NP -hard.

4 Unbounded Number of Scenarios

When the number of scenarios is unbounded, min-max and min-max regret shortest path as well as min-max spanning tree and max-min knapsack were proved strongly NP -hard in [3]. We establish the strong NP -hardness of min-max regret knapsack and min-max regret spanning tree in Theorems 5 and 9 respectively.

Concerning approximability results, reductions used in [3] for proving the strong NP -hardness of min-max/min-max regret shortest path, and min-max spanning tree, which are based on the 3-partition problem, cannot be used to establish non-approximability results for these problems. Using alternative reductions, we establish such results in Theorems 6-9. On the other hand, the reduction used in [3] for proving the strong NP -hardness of max-min knapsack is stronger and can be used to establish non-approximability results. In fact, it is a gap-introducing reduction from the set covering problem which maps positive instances into instances with optimum value at least 1 and negative instances into instances with optimum value 0. Therefore, we can deduce from this reduction that MAX-MIN KNAPSACK is not $f(n)$ -approximable for any function $f : \mathbb{N} \rightarrow (1, \infty)$. Finally, regarding MIN-MAX REGRET KNAPSACK, we know already that it is not $f(n)$ -approximable for any function $f : \mathbb{N} \rightarrow (1, \infty)$, since even for two scenarios it is not approximable as shown in Theorem 4.

Now we state and prove the above-mentioned results.

Theorem 5. MIN-MAX REGRET KNAPSACK, with an unbounded number of scenarios, is strongly NP -hard.

Proof. We construct a gap-introducing reduction from VERTEX COVER. Given a graph $G = (V, E)$ on n vertices and m edges and a positive integer k , we define an instance I of MIN-MAX REGRET KNAPSACK with n items and a set of m scenarios $S = \{s_1, \dots, s_m\}$. The weights are $w_i = 1$, for any $i = 1, \dots, n$, the knapsack capacity is $d = k$ and the value of item i in scenario s_j is $v_i^{s_j} = 1$ if node $i \in V$ is incident to edge $j \in E$, and 0 otherwise.

Observe first that $val_{s_j}^* = 2$, for all $s_j \in S$, which is obtained by taking the two items corresponding to the extremities of edge j . If G has a vertex cover V' of size at most k then the subset of items x' corresponding to V' has $val(x', s_j) \geq 1$, for any $s_j \in S$ since edge j is covered by V' . Thus, $R_{max}(x') \leq 1$, which implies $opt(I) \leq 1$.

If G has no vertex cover of size at most k then for any $V' \subseteq V$, $|V'| \leq k$, there exists $s_j \in S$, corresponding to an edge j which is not covered by V' , such that the subset of items x' corresponding to V' has $val(x', s_j) = 0$, and thus $R_{max}(x') = 2$, which implies $opt(I) = 2$.

The existence of a polynomial-time algorithm would allow us to decide for VERTEX COVER in polynomial time. □

Observe that the $(2 - \varepsilon)$ non-approximability result that could be derived from this proof is weaker than the result stated in Theorem 4.

We show in the following a non-approximability result for min-max and min-max regret versions of shortest path. For this, we use a reduction from PATH WITH FORBIDDEN PAIRS that is known to be NP -hard [2].

PATH WITH FORBIDDEN PAIRS

Input: A directed graph $G = (V, A)$, where $V = \{1, \dots, n\}$, a collection $C = \{(a_1, b_1), \dots, (a_t, b_t)\}$ of arcs from A .

Question: Is there a path from 1 to n in G containing at most one vertex from each arc of C ?

Theorem 6. MIN-MAX SHORTEST PATH, with an unbounded number of scenarios, is not $(2 - \varepsilon)$ -approximable, for any $\varepsilon > 0$, unless $P = NP$.

Proof. We construct a gap-introducing reduction from PATH WITH FORBIDDEN PAIRS. Let I be an instance of this problem with n vertices and m arcs, and t arcs in collection C . We construct an instance I' of MIN-MAX SHORTEST PATH as follows: consider the same graph $G = (V, A)$, a scenario set $S = \{s_1, \dots, s_t\}$, and costs of arcs defined for each scenario as

$$c_{ij}^{s_h} = \begin{cases} 2 & \text{if arc } (i, j) \text{ corresponds to } (a_h, b_h) \\ 1 & \text{if } i = a_h \text{ or } j = b_h, (i, j) \neq (a_h, b_h) \\ 0 & \text{otherwise} \end{cases}$$

Suppose that I is a positive instance, that is G contains a path p from 1 to n that has at most one extremity from each of the t arcs of C . Then for any scenario s , we have $val(p, s) \leq 1$. Then $\max_{s \in S} val(p, s) \leq 1$, which implies $opt(I') \leq 1$.

If I is a negative instance, then every path p from 1 to n in G contains either an arc or both extremities of an arc (a_h, b_h) from C . Then $val(p, s_h) = 2$ in both cases. Thus $\max_{s \in S} val(p, s) = 2$, which implies $opt(I') = 2$. \square

Theorem 7. MIN-MAX REGRET SHORTEST PATH, with an unbounded number of scenarios, is not $(2 - \varepsilon)$ -approximable, for any $\varepsilon > 0$, unless $P = NP$.

Proof. As for the previous theorem, we construct a similar gap-introducing reduction from PATH WITH FORBIDDEN PAIRS. Let I be an instance of this problem with n vertices and m arcs, and t arcs in the collection C . We construct an instance I'' of MIN-MAX REGRET SHORTEST PATH as follows: consider graph $G' = (V', A')$, where $V' = V \cup \{n + 1, \dots, n + |S|\}$, $A' = A \cup \{(1, i) : i = n + 1, \dots, n + |S|\} \cup \{(i, n) : i = n + 1, \dots, n + |S|\}$, and a scenario set $S = \{s_1, \dots, s_t\}$. The costs of arcs in A are defined for each scenario $s \in S$ as in the previous theorem, and for any $s \in S$

$$c_{1, n+i}^s = c_{n+i, n}^s = \begin{cases} 0 & \text{if } s = s_i \\ 1 & \text{if } s \neq s_i \end{cases}$$

Obviously, $val_{s_i}^* = 0$ since the path $(1, n + i, n)$ has value 0 on scenario s_i . As previously, we can prove that if I is a positive instance, then $opt(I'') \leq 1$, otherwise $opt(I'') = 2$. \square

We show in the following non-approximability results for min-max and min-max regret versions of spanning tree. The first result uses a reduction from MINIMUM DEGREE SPANNING TREE that is known to be not $(\frac{3}{2} - \varepsilon)$ -approximable, for any $\varepsilon > 0$ [2].

MINIMUM DEGREE SPANNING TREE

Input: A graph $G = (V, E)$.

Output: A spanning tree such that its maximum degree is minimum.

Theorem 8. MIN-MAX SPANNING TREE, with an unbounded number of scenarios, is not $(\frac{3}{2} - \varepsilon)$ -approximable, for any $\varepsilon > 0$, unless $P = NP$.

Proof. We construct an approximation preserving reduction from MINIMUM DEGREE SPANNING TREE. Let $G = (V, E)$ be an instance of this problem on n vertices. We construct an instance of MIN-MAX SPANNING TREE on the same graph G , with a set of n scenarios $S = \{s_1, \dots, s_n\}$, and costs of edges in scenario s_h defined by $c_{ij}^{s_h} = 1$ if $h = i$ or $h = j$ and 0, otherwise. Then for any spanning tree T of G , the degree of $i \in V$ in T is the same as $val(T, s_i)$. Thus, the maximum degree of T , that is $\max_{i \in V} d_T(i)$, coincides with the maximum value of T over all scenarios from S , that is $\max_{s \in S} val(T, s)$. \square

Theorem 9. MIN-MAX REGRET SPANNING TREE, with an unbounded number of scenarios, is strongly NP-hard. Moreover, it is not $(\frac{3}{2} - \varepsilon)$ -approximable, for any $\varepsilon > 0$, unless $P = NP$.

Proof. We construct a gap-introducing reduction from 3SAT. Given a set $U = \{u_1, \dots, u_n\}$ of boolean variables and a formula ϕ containing the clauses $\{C_1, \dots, C_m\}$ over U such that each clause depends on exactly 3 variables, we construct an instance I of MIN-MAX REGRET SPANNING TREE defined on a graph $G = (V, E)$ where $V = \{1, \dots, n\} \cup \{\bar{1}, \dots, \bar{n}\} \cup \{n + 1, \dots, 3n\}$. Vertices i, \bar{i} , correspond to variable $u_i, i = 1, \dots, n$. Edge set is $E = \{(i, n + i), (i, 2n + i), (\bar{i}, n + i), (\bar{i}, 2n + i), (i, \bar{i}) : i = 1, \dots, n\} \cup \{(\bar{i}, i + 1) : i = 1, \dots, n - 1\}$. Scenario set $S = S_1 \cup S_2 \cup S_3$ where $S_1 = \{s_1, \dots, s_m\}$ corresponds to clauses and $S_2 = \{s'_{n+1}, \dots, s'_{3n}\}$, $S_3 = \{s'_1, \dots, s'_n, s''_1, \dots, s''_n\}$ correspond to vertices of G . The costs of edges in scenario $s_j \in S_1$ are defined as follows: $c_{i,2n+i}^{s_j} = 1$ if $u_i \in C_j$, $c_{\bar{i},2n+i}^{s_j} = 1$ if $\bar{u}_i \in C_j$, and 0 otherwise. The values of edges in scenario $s'_j \in S_2$ are defined as follows: $c_{i,n+i}^{s'_j} = c_{\bar{i},n+i}^{s'_j} = n$, $c_{i,2n+i}^{s'_j} = c_{\bar{i},2n+i}^{s'_j} = n$, for every $i = 1, \dots, n$ and 0 otherwise. The values of edges in scenario $s'_j \in S_3$ are defined as follows: $c_{i,n+i}^{s'_j} = c_{\bar{i},n+i}^{s'_j} = c_{\bar{i},i}^{s'_j} = 2$, $c_{\bar{i},n+i}^{s'_j} = c_{i,2n+i}^{s'_j} = c_{\bar{i},i}^{s'_j} = 2$, for every $i = 1, \dots, n$ and 0 otherwise.

We compute in the following the optimum costs corresponding to each scenario. For any scenario $s_j \in S_1$, consider the spanning tree containing $\{(\bar{i}, i + 1) : i = 1, \dots, n - 1\}$ and $\{(i, n + i), (\bar{i}, 2n + i), (i, \bar{i})\}$, for every i such that $u_i \in C_j$, or $\{(i, 2n + i), (\bar{i}, n + i), (i, \bar{i})\}$, otherwise. Obviously, this tree has value 0 in scenario s_j . For any scenario $s'_{n+i} \in S_2$, $val_{s'_{n+i}}^* = n$ since any spanning tree contains one of the edges $(i, n + i), (\bar{i}, n + i)$. Similarly, $val_{s'_{2n+i}}^* = n$, for all $s'_{2n+i} \in S_2$. For any scenario $s'_i \in S_3$, $val_{s'_i}^* = 2$ since any spanning tree contains at least one of the edges $(i, n + i), (i, 2n + i), (i, \bar{i})$. Similarly, $val_{s''_i}^* = 2$, for all $s''_i \in S_3$.

A spanning tree in G necessarily contains edges $(\bar{i}, i + 1), i = 1, \dots, n - 1$. We show in the following that every spanning tree T that contains edges $(i, \bar{i}), (i, n + i), (\bar{i}, 2n + i)$ or edges $(i, \bar{i}), (i, 2n + i), (\bar{i}, n + i)$ for every $i = 1, \dots, n$, has $R_{max}(T) \leq 3$. Moreover, any other spanning tree T' in G has $R_{max}(T') \geq 4$. We have $val(T, s_j) \leq 3$, for any $s_j \in S_1$, $val(T, s'_j) = n$, for any $s'_j \in S_2$, and $val(T, s'_j) = 4$, for any $s'_j \in S_3$. Thus, $R_{max}(T) \leq 3$. If T' contains both edges $(i, n + i), (\bar{i}, n + i)$ for some i , then $val(T', s'_{n+i}) = 2n$ and thus $R_{max}(T') = n$. We can also see that if a spanning tree T' contains both edges $(i, 2n + i), (\bar{i}, 2n +$

i) for some i , then $val(T', s'_{2n+i}) = 2n$ and thus $R_{max}(T') = n$. Consider in the following spanning trees T' that contain edges (i, \bar{i}) , $i = 1, \dots, n$. If T' contains both edges $(i, n+i)$, $(i, 2n+i)$ for some i , then $val(T', s'_i) = 6$ and thus $R_{max}(T') = 4$. We can see also that if T' contains both edges $(\bar{i}, n+i)$, $(\bar{i}, 2n+i)$ for some i , then $val(T', s'_{\bar{i}}) = 6$ and thus $R_{max}(T') = 4$. Thus an optimum solution in G is a spanning tree T that contains edges $(\bar{i}, i+1)$, $i = 1, \dots, n-1$, edges (i, \bar{i}) , $i = 1, \dots, n$, and, for every $i = 1, \dots, n$, it contains either edges $(i, n+i)$, $(\bar{i}, 2n+i)$ or edges $(i, 2n+i)$, $(\bar{i}, n+i)$. Such spanning trees are in one-to-one correspondence with assignments of variables u_1, \dots, u_n . More precisely, T contains for some i edges $(i, n+i)$, $(\bar{i}, 2n+i)$ if and only if u_i takes value 1, and it contains edges $(i, 2n+i)$, $(\bar{i}, n+i)$ if and only if u_i takes value 0. If ϕ is satisfiable, then there exists an assignment x for u_1, \dots, u_n that satisfies each clause. Then, consider the spanning tree T associated to x . Every clause C_j is satisfied by x . Therefore, there exists $u_i \in C_j$, such that u_i has value 1 in x or $\bar{u}_i \in C_j$, such that u_i has value 0 in x . In both cases, $val(T, s_j) \leq 2$, for any $s_j \in S_1$. Tree T has also $val(T, s) = n$, for any $s \in S_2$ and $val(T, s) = 4$, for any $s \in S_3$, and thus, $R_{max}(T) = 2$, which implies $opt(I) = 2$.

Suppose now that ϕ is not satisfiable, that is for any assignment x , there exists a clause C_j that is not satisfied. Therefore, for any spanning tree T associated to x , we have $val(T, s_j) = 3$, and thus $R_{max}(T) = 3$, which implies $opt(I) = 3$. \square

References

1. G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and approximation. Combinatorial optimization problems and their approximability properties*. Springer, 1999.
2. M. Garey and D. Johnson. *Computer and Intractability: A Guide to the theory of NP-completeness*. Freeman, 1979.
3. P. Kouvelis and G. Yu. *Robust Discrete Optimization and its Applications*. Kluwer Academic Publishers, Boston, 1997.
4. M. Mahajan and V. Vinay. Determinants: combinatorics, algorithms, and complexity. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 730–738, New Orleans, USA, 1997.
5. C. H. Papadimitriou and M. Yannakakis. On the approximability of trade-offs and optimal access of web sources. In *IEEE Symposium on Foundations of Computer Science*, pages 86–92, 2000.
6. S. Sahni. General techniques for combinatorial approximation. *Operations Research*, 25(6):920–936, 1977.
7. V. V. Vazirani. *Approximation Algorithms*. Springer, 2001.
8. D. Zuckerman. NP-complete problems have a version that's hard to approximate. In *Proceeding 8th Annual Conference on Structure in Complexity Theory*, pages 305–312, 1993.

Robust Approximate Zeros^{*}

(Extended Abstract)

Vikram Sharma, Zilin Du, and Chee K. Yap

Courant Institute of Mathematical Sciences,
New York University, New York, NY 10012, USA
{sharma, zilin, yap}@cs.nyu.edu

Abstract. Smale's notion of an approximate zero of an analytic function $f : \mathbb{C} \rightarrow \mathbb{C}$ is extended to take into account the errors incurred in the evaluation of the Newton operator. Call this stronger notion a **robust approximate zero**. We develop a corresponding **robust point estimate** for such zeros: we prove that if $z_0 \in \mathbb{C}$ satisfies $\alpha(f, z_0) < 0.02$ then z_0 is a robust approximate zero, with the associated zero z^* lying in the closed disc $\overline{B}(z_0, \frac{0.07}{\gamma(f, z_0)})$. Here $\alpha(f, z), \gamma(f, z)$ are standard functions in point estimates.

Suppose $f(z)$ is an L -bit integer square-free polynomial of degree d . Using our new algorithm, we can compute an n -bit absolute approximation of $z^* \in \mathbb{R}$ starting from a bigfloat z_0 , in time $O[dM(n + d^2(L + \lg d) \lg(n + L))]$, where $M(n)$ is the complexity of multiplying n -bit integers.

1 Introduction

The Newton-Raphson method has been studied extensively in many settings. Given an analytic function $f : \mathbb{C} \rightarrow \mathbb{C}$ and a point $z_0 \in \mathbb{C}$, we consider the iteration $z_{i+1} = N_f(z_i)$ for $i \geq 0$ where $N_f(z) := z - f(z)/f'(z)$. This sequence is well-defined provided $f'(z_i) \neq 0$ for all $i \geq 0$. Kantorovich [KA64] developed convergence criteria for $(z_i)_{i \geq 0}$ that are applicable when points in an entire neighborhood of z_0 satisfy certain bounds. Yamamoto [Yam85, Yam86] gives sharp bounds of this sort. A basic technique in Kantorovich's approach is the use of majorant sequences. Unfortunately, Kantorovich's criteria are sometimes inconvenient to use. Smale [Sma86, BCSS98] developed convergence criteria that are applicable to a single point $z_0 \in \mathbb{C}$. Such criteria are called **point estimates**. Following [BCSS98–p. 155], call z_0 an **approximate zero** of $f(z)$ if the sequence $z_{i+1} = N_f(z_i)$ is well defined for all natural numbers i and there exists a root z^* of $f(z)$ such that for all $i \geq 0$,

$$|z_i - z^*| \leq 2^{1-2^i} |z_0 - z^*|;$$

* This research is supported by NSF Grant #CCF-043836. The work of Yap is partially carried out at the Korea Institute for Advanced Study (KIAS).

z^* is called the associated zero. One such point estimate [DF95] says that if $\alpha(f, z_0) < 3 - 2\sqrt{2} \sim 0.17157$ then z_0 is an approximate zero. Here $\alpha(f, z)$ is an easily computed function defined in the next Section.

Variations, improvements and extensions are known. Kim [Kim86, Kim88] derived comparable point estimates for slightly different¹ notions of approximate zeros than the one defined above. Shub and Smale [SS85, SS93] and Malajovich [Mal93, Mal94] have developed such criteria for multivariate Newton methods in affine and projective spaces. Malajovich further extended this to pseudo Newton iteration, i.e., Newton iteration using the Moore-Penrose inverse. Wang and Zhao [DF95] improved Smale's point estimate using Kantorovich's approach, and extended it to the Weierstrass method [Dur60, Ker66]. Petkovic and others [PCT95, PHI98, Bat98] also obtained point estimates for the Weierstrass method.

The above results are developed in a setting where the operations are assumed to be exact, i.e., $N_f(z)$ can be computed without error. Even when this is possible, such as the case where z is rational and $f(z)$ an integer polynomial, it may be undesirable because of inefficiency. In practice, the z_i 's will be represented by floating point numbers. In this paper, we assume the use of **bigfloats**, i.e., floating point numbers whose exponent and mantissas are arbitrary precision integers. Since $N_f(z)$ involves division, the use of approximation is essential in bigfloat computation. Indeed, Newton iteration is uniquely suited for approximation because of its known self-correcting behavior. Bigfloat arithmetic is basically the multiple-precision arithmetic of Brent [Bre76a, Bre76b]. The fundamental results in this model have been achieved by Brent 30 years ago; but we shall point out new issues in this paper.

Although there is a large literature on the error analysis of Newton iteration [Ypm83, Ypm84, Tis01, Hig96], these results do not address the point estimate setting. To our best knowledge, the only such results are by Malajovich [Mal94]. There are several differences between our work and Malajovich's.

- We focus on the univariate case while Malajovich address the more general case of multi-variate Newton. Consequently, our complexity bounds for the univariate case are much stronger than Malajovich's bound (when specialized to the univariate case). Indeed, Malajovich's complexity statements (see [Mal93–p. 2, Main Theorem and p. 79-80] or [Mal94–p. 2, and p. 8, Theorem 10]) contain terms that have no explicit complexity bounds.
- Malajovich assumes that each Newton step is computed to a fixed precision s . In contrast, we follow Brent's approach of doubling the precision at each iteration. This has the advantage that the overall complexity is essentially determined by the last iteration step (see [Bre76a, Bre76b]).
- Finally, Malajovich's robust point estimate involves an extra parameter s (the precision of the bigfloat computations steps above). In particular, he shows that z_0 and s should satisfy $\alpha(f, z_0) < 0.05$ and $\gamma(f, z_0)s < 1/384$. Since s has to be at least the precision with which we want to approxi-

¹ For that matter, Smale has used more than one variant in his papers.

mate the zero, this criterion imposes additional constraints on the procedure for finding z_0 . In contrast, our robust point estimate only requires $\alpha(f, z_0) \leq 0.02$, which is independent of the desired final precision. Our approach guarantees convergence to the root, unlike Malajovich’s approach where the distance between the iterates and the root can only be bounded above by 2^{-6s} .

Error Notation. If $z, \tilde{z} \in \mathbb{C}$ and $t \in \mathbb{R}$, then \tilde{z} is an **absolute t -bit approximation** of z if $|z - \tilde{z}| \leq 2^{-t}$. Similarly, \tilde{z} is a **relative t -bit approximation** of z if $|z - \tilde{z}| \leq 2^{-t}|z|$. We use two convenient notations for error bounds: we shall write

$$[z]_t \quad (\text{resp., } \langle z \rangle_t) \tag{1}$$

for *any* relative (resp., absolute) t -bit approximation of z . Furthermore, the symbol “ \pm ” in this paper has a specialized meaning: when we write “ $x \pm y$ ”, it stands for some number $x + \theta y$ where θ satisfies $|\theta| \leq 1$. E.g., $y = x \pm \varepsilon$ is equivalent to $y \in [x - |\varepsilon|, x + |\varepsilon|]$.

BigFloat Model of Computation. As in Brent [Bre76b, Bre76a], we use bigfloat numbers to approximate real or complex numbers. If f is an integer, write $\langle f \rangle$ for the value $f2^{-\lfloor \lg |f| \rfloor}$; thus $\langle f \rangle \in [1, 2)$. A (binary) **bigfloat** is a rational number of the form $x = n2^m$ where $n, m \in \mathbb{Z}$. We say x has **precision t** if $|n| < 2^t$. We represent $x = n2^m$ by a pair (e, f) of binary integers. Given an arbitrary pair (e, f) the associated bigfloat number, denoted $\langle e, f \rangle$, is

$$\langle e, f \rangle = f2^{e - \lfloor \lg f \rfloor} = \langle f \rangle 2^e.$$

Thus $n2^m \equiv \langle e, f \rangle$, where $f = n$ and $e = m + \lfloor \lg |n| \rfloor$ is the **exponent**. The **bit size** of a representation (e, f) is the pair $(1, 1)$ when $f = 0$; $(1, \lg(2|f|))$, when $e = 0$; otherwise, $(\lg(2|e|), \lg(2|f|))$, where $\lg = \log_2$.

We distinguish two **modes** of using bigfloats. In the **weak (bigfloat) mode**, one fixes some precision bound which is used by all the bigfloats in a computation. Thus a weak mode computation can be regarded as a generalization of the IEEE model implemented in hardware in modern computers. Malajovich’s algorithms operate in this weak mode. In the **strong (bigfloat) mode**, we use bigfloats without a priori precision bounds, and the algorithms can actively manage the precision of each computation step. Brent’s complexity results (as well as ours) are achieved in this strong mode. Although our complexity model is essentially Brent’s, our treatment deviates from Brent in three ways:

- Brent’s complexity analysis applies to floating point numbers in a bounded range. For a floating point number $\langle e, f \rangle$, “bounded range” means $|e| = O(1)$. For unbounded floating point numbers, our complexity bounds depends on $\lg(2+|e|)$. This dependence can range from polynomial (e.g. Lemma 7 below) to exponential (e.g., Lemma 8 below), and it may not be obvious when this happens. Our complexity results apply to unbounded bigfloats. See also [CSY97].

- Brent uses the big-Oh notation in two ways: in error analysis and in complexity estimates. Unfortunately, when implementing such algorithms, a big-Oh error analysis does not tell us important constants needed in various places of an algorithm. Therefore, we will use **non-asymptotic error analysis** although our complexity analysis will continue to use asymptotics.
- Finally, our complexity model is based on Schönhage’s pointer machine model [Sch80], rather than the standard multi-tape Turing machines. This is because Turing machines are not robust enough for our complexity estimates involving unbounded bigfloats. E.g., if a bigfloat $\langle e, f \rangle$ is represented in the obvious way on a Turing tape, we cannot read f without scanning e . This causes unbounded distortion of the complexity of basic operations such as truncation. Other conventions also have problems. Note that for pointer machines, we can multiply n -bit numbers in time $M(n) = O(n)$. We expressed complexity bounds in terms of $M(n)$ so that even if suboptimal multiplication algorithms are used, we can gauge their effects on complexity.

Contributions of This Paper. Our main results are as follows:

1. In Section 2 we introduce a notion of **robust approximate zero** of an analytic function $f : \mathbb{C} \rightarrow \mathbb{C}$ and give a corresponding **robust point estimate** for $z_0 \in \mathbb{C}$ to be a such a zero.
2. Section 3 derives explicit bounds on the precision necessary to carry out the steps of a robust Newton iteration.
3. In Section 5, we give explicit complexity bounds for approximating a zero of a square-free integer polynomial starting from a robust approximate zero. This can be viewed as an extension of Brent’s complexity bound (for algebraic roots) to the case of unbounded bigfloats.
4. Our introduction of non-asymptotic error analysis for bigfloat computation is motivated by implementations needs. In the full version of this paper, implementations and comparisons in Core Library [KLPY99] will be reported.

The Core Library [KLPY99], and also LEDA [BFMS99, MS01], provides a number system with guaranteed *a priori* precision bounds, unlike the guaranteed *a posteriori* precision bounds of interval analysis. Furthermore, this precision guarantee is global in nature, as the bound is relative to an arbitrary sequence of computational steps. In contrast, conventional IEEE arithmetic gives local precision guarantees (being relative to each operation). See [Yap04] for discussion of such “guaranteed precision mode” of computation.

2 Robust Newton Iteration

Let $f : \mathbb{C} \rightarrow \mathbb{C}$ be any analytic function with a simple root at z^* . We may assume f is fixed in this paper and $N_f(z) = z - \frac{f(z)}{f'(z)}$ is its Newton iterator. Given $z \in \mathbb{C}$ and $C \in \mathbb{R}$, let

$$N_{f,i,C}(z) := \langle N_f(z) \rangle_{2^i + C}. \quad (2)$$

Equation (2) uses our error notation of (1): this means $|N_{f,i,C}(z) - N_f(z)| \leq 2^{-2^i - C}$. For any $z_0 \in \mathbb{C}$ and $C \in \mathbb{R}$, a **robust iteration sequence of z_0 (relative to C and f)** is an infinite sequence

$$(\tilde{z}_i)_{i \geq 0} \tag{3}$$

such that $\tilde{z}_0 = z_0$, and for all $i \geq 1$,

$$\tilde{z}_i = N_{f,i,C}(\tilde{z}_{i-1}). \tag{4}$$

We assume each $\tilde{z}_i \in \mathbb{C} \cup \{\infty\}$, and the relation (4) must be understood in the following way: if $\tilde{z}_{i-1} = \infty$ or \tilde{z}_{i-1} is a critical point of f (i.e., $f'(\tilde{z}_{i-1}) = 0$), then $\tilde{z}_i = \infty$. We call the iteration sequence **finite** if each $\tilde{z}_i \neq \infty$.

Our key definition is as follows: z_0 is a **robust approximate zero** of f if, there exists a zero z^* of f , such that for all C satisfying

$$2^{-C} \leq |z_0 - z^*|, \tag{5}$$

whenever $(\tilde{z}_i)_{i \geq 0}$ is any robust iteration sequence of z_0 (relative to C and f), then the sequence is finite and for all $i \geq 0$,

$$|\tilde{z}_i - z^*| \leq 2^{1-2^i} |z_0 - z^*|. \tag{6}$$

Call z^* the **associated zero** of z_0 .

We now recall several functions used in Smale’s analysis of approximate zeros:

- $\gamma(f, z) := \sup_{k \geq 2} \left| \frac{f^{(k)}(z)}{k!f'(z)} \right|^{1/(k-1)}$.
- $\beta(f, z) := \left| \frac{f(z)}{f'(z)} \right|$.
- $\alpha(f, z) := \beta(f, z)\gamma(f, z)$.
- $\psi(x) := 1 - 4x + 2x^2$. The roots of ψ are $(2 \pm \sqrt{2})/2$.
- $u(z, w) := \gamma(f, z)|z - w|$. For the special case where $z = z^*$, a root of f , we use the succinct notation u_w .

Smale et al. [BCSS98–p. 156, Thm. 1] have shown the following:

Proposition 1. *If z^* is a simple zero of $f(z)$, then $z_0 \in \mathbb{C}$ is an approximate zero of f with associated zero z^* if*

$$|z_0 - z^*| \leq \frac{3 - \sqrt{7}}{2\gamma(f, z^*)}.$$

Here is our robust analogue:

Theorem 1. *If z^* is a simple zero of $f(z)$, then $z_0 \in \mathbb{C}$ is a robust approximate zero of f with associated zero z^* if*

$$|z_0 - z^*| \leq \frac{4 - \sqrt{14}}{2\gamma(f, z^*)}.$$

Proof. Let $u_z = \gamma(f, z^*)|z - z^*|$ as above. We prove (6) by induction on $i \geq 0$. The result is clearly true for $i = 0$. Inductively, assume that \tilde{z}_i satisfies (6). Then $u_{\tilde{z}_i} \leq 2^{1-2^i} u_{z_0}$. Since $u_{z_0} \leq \frac{4-\sqrt{14}}{2}$, it is smaller than the both roots of $\psi(x)$. Hence

$$\psi(u_{\tilde{z}_i}) \geq \psi(u_{z_0}). \tag{7}$$

Thus,

$$\begin{aligned} |\tilde{z}_{i+1} - z^*| &= |N_{f,i+1,C}(\tilde{z}_i) - z^*| \\ &\leq |N_f(\tilde{z}_i) - z^*| + 2^{-2^{i+1}} |z_0 - z^*| \quad (\text{from (5)}) . \end{aligned}$$

From [BCSS98–p. 157, Prop. 1] we further get

$$\begin{aligned} |N_f(\tilde{z}_i) - z^*| &\leq \frac{\gamma(f, z^*)}{\psi(u_{\tilde{z}_i})} |\tilde{z}_i - z^*|^2 \\ &\leq \frac{\gamma(f, z^*)}{\psi(u_{z_0})} |\tilde{z}_i - z^*|^2 \quad (\text{from (7)}) . \end{aligned}$$

From the inductive hypothesis we thus get,

$$\begin{aligned} |\tilde{z}_{i+1} - z^*| &\leq \frac{\gamma(f, z^*)}{\psi(u_{z_0})} 2^{2-2^{i+1}} |z_0 - z^*|^2 + 2^{-2^{i+1}} |z_0 - z^*| \\ &= \frac{u_{z_0}}{\psi(u_{z_0})} 2^{2-2^{i+1}} |z_0 - z^*| 2^{1-2^{i+1}} |z_0 - z^*| \\ &\leq 2^{1-2^{i+1}} |z_0 - z^*|, \end{aligned}$$

since the assumption $u_{z_0} \leq \frac{4-\sqrt{14}}{2}$ implies $\frac{u_{z_0}}{\psi(u_{z_0})} \leq \frac{1}{4}$. **Q.E.D.**

Let the continuous function $\Gamma : S \rightarrow S$ be a contraction map on $S \subseteq \mathbb{C}$ with contraction constant $K < 1$; this implies that there is a unique fixed point $z^* \in S$ of Γ such that for all $z \in S$, the sequence $(\Gamma^n(z))_{n \geq 0}$ converges to z^* . We consider the inexact analogue of $\Gamma^n(z)$:

Lemma 1. *Let $\Gamma_{i,C}(z) := \langle \Gamma(z) \rangle_{i+C}$ (for $C \in \mathbb{R}$ and $i \geq 0$). If $C \geq -\lg(|z - z^*|)$, then the sequence*

$$\tilde{z}_{i+1} := \Gamma_{i+1,C}(\tilde{z}_i),$$

starting from $\tilde{z}_0 := z_0$, converges to $z^ \in S$, assuming $\tilde{z}_i \in S$ for each i .*

The following shows that under suitable restrictions on z_0 the robust iteration sequence defined in (4) converges to a root z^* of f . Let $\overline{B}(z, R)$ denote the closed disc with center $z \in \mathbb{C}$ and radius R .

Lemma 2. *Suppose there exist constants α_0, u_0 and $C_0 := \frac{2(\alpha_0 + u_0)}{\psi(u_0)^2}$ which satisfy the following criteria:*

1. $0 \leq u_0 < 1 - 1/\sqrt{2}$,
2. $C_0 < \frac{3}{4}$,
3. $\alpha_0 \leq (\frac{3}{4} - C_0)u_0$, and
4. $\frac{u_0}{\psi(u_0)(1-u_0)} \leq \frac{4-\sqrt{14}}{2}$.

If $z_0 \in \mathbb{C}$ is such that $\alpha(f, z_0) < \alpha_0$ we have the following:

- (a) N_f is a contraction map on $\overline{B}(z_0, \frac{u_0}{\gamma(f, z_0)})$ with contraction constant C_0 .
- (b) z_0 is a robust approximate zero of f , the associated zero $z^* \in \overline{B}(z_0, \frac{u_0}{\gamma(f, z_0)})$.

One choice of constants that satisfy the above criteria is $u_0 = 0.07$ and $\alpha_0 = 0.02$.

Theorem 2 (Point estimate for robust approximate zero). Any $z_0 \in \mathbb{C}$ for which $\alpha(f, z_0) < 0.02$ is a robust approximate zero of f , with the associated zero $z^* \in \overline{B}(z_0, \frac{0.07}{\gamma(f, z_0)})$.

3 Approximate Evaluation of Newton Iterator

Let $f(z)$ be a square-free integer polynomial. In this section we determine the absolute precision with which to evaluate f and f' , and the relative precision with which to carry out the division at each iteration step; let these be e_i , E_i , and ϱ_i , respectively.

We will have recourse to the next two lemmas which apply to an analytic f .

Lemma 3. Let $u = \gamma(f, z)|z - w| < 1 - \frac{1}{\sqrt{2}}$. Then we have

$$\frac{\psi(u)}{(1 - u)^2} \leq \frac{|f'(w)|}{|f'(z)|} \leq \frac{1}{(1 - u)^2}.$$

Lemma 4. Let z be such that $u_z = \gamma(f, z^*)|z - z^*| < 1$, where z^* is a simple root of f . Then

$$\frac{|z - z^*|(1 - 2u_z)}{1 - u_z} \leq \left| \frac{f(z)}{f'(z^*)} \right| \leq \frac{|z - z^*|}{1 - u_z}.$$

Let $z_0 \in \mathbb{C}$ such that $\alpha(f, z_0) < 0.02$; from Thm. 2 we know that z_0 is a robust approximate zero with an associated root z^* and $u_{z_0} \leq \frac{4 - \sqrt{14}}{2}$, and hence $\psi(u_{z_0}) \geq \frac{1}{2}$. Let $(\tilde{z}_i)_{i \geq 0}$ be a robust approximate sequence starting from z_0 , relative to a constant C satisfying (5). Writing $\delta_i = \tilde{z}_i - z^*$, we know $|\delta_i| \leq 2^{-2^i + 1} |\delta_0|$.

The main result of this section is:

Theorem 3. To compute an absolute approximation $\langle N_f(\tilde{z}_i) \rangle_{2^i + C}$ it suffices to

- (i) evaluate $f(\tilde{z}_i)$ to $(\kappa + 2^{i+1} + 4 + C)$ absolute bits,
- (ii) evaluate $f'(\tilde{z}_i)$ to $(\kappa' + 2^i + 3 + C)$ absolute bits,
- (iii) and perform the division in N_f to $(\kappa'' + 2^i + 1 + C)$ relative bits.

Here, $\kappa \geq -\lg |f'(z_0)|$, $\kappa' \geq -\lg |f'(z_0)|\gamma(f, z_0)$ and $\kappa'' \geq 3 - \lg \gamma(f, z_0)$.

4 Estimating the Distance between an Approximate Zero and its Associated Root

Let z_0 be a robust approximate zero with the associated zero z^* . To construct a robust iteration sequence (4) converging to z^* , we need to determine a $C \in \mathbb{Z}$ satisfying (5), or equivalently, $C \geq -\lg |z_0 - z^*|$. In this section we compute tight bounds on $|z_0 - z^*|$ where z_0 is an approximate zero (not just a robust approximate zero). We assume that $\alpha(f, z_0) < 0.03$. Then from [BCSS98–p. 160, Thm. 2] and [BCSS98–p. 166, Remark 6], we know that z_0 is an approximate zero satisfying Prop. 1.

We can use an inequality from Kalantari [Kal05]: for any $z_0 \in \mathbb{C}$,

$$|z_0 - z^*| \geq \frac{1}{2\gamma_2(f, z_0)} \tag{8}$$

where

$$\gamma_2(f, z_0) := \sup_{k \geq 1} \left| \frac{f^{(k)}(z_0)}{k!f(z_0)} \right|^{1/k}. \tag{9}$$

Hence it suffices to choose any C satisfying

$$C \geq 1 + \lg \gamma_2(f, z_0). \tag{10}$$

The Kalantari function $\gamma_2(f, z_0)$ is easily approximated in practice.

Since C controls the number of bits used in our robust iteration, it is desirable for C to be as small as possible. We pose the problem of computing C up to some additive constant $K > 0$. More precisely, compute any C which satisfies

$$0 \leq C + \lg |z_0 - z^*| \leq K. \tag{11}$$

Kalantari’s estimate (10) is not known to satisfy (11). In short, we want a tight estimate of the distance $|z_0 - z^*|$ between z_0 and its associated zero z^* . We could use Turan’s proximity test [Pan97] to approximate the minimum and maximum distances from any complex number to the zeros of a polynomial $f(z)$ within a constant factor, at the cost of $O(d \lg d)$ arithmetic operations, where $d = \deg f(z)$. We do not use this test because it is limited to polynomials, and also it does not leverage the fact that z_0 is an approximate zero.

Our solution exploits the property of approximate zeros, based on a tight relationship between $\delta := \left| \frac{f(z_0)}{f'(z_0)} \right|$ ($= \beta(z_0)$) and $|z_0 - z^*|$:

Lemma 5. *Let $z \in \mathbb{C}$ satisfy $u = \gamma(f, z^*)|z - z^*| < 1 - \frac{1}{\sqrt{2}}$, where z^* is a simple root of f . Then*

$$|z - z^*|(1 - 2u)(1 - u) \leq \left| \frac{f(z)}{f'(z)} \right| \leq \frac{|z - z^*|(1 - u)}{\psi(u)}.$$

We now describe our algorithm:

ALGORITHM D	
INPUT:	f, z_0 where $\alpha(f, z_0) < 0.03$
OUTPUT:	n such that $ f(z_0)/f'(z_0) = C' \cdot 2^{-n}$ for some $0.5 \leq C' \leq 3$.
1	$n = 0$.
2	Do
3	$w \leftarrow \left\langle \frac{f(z_0)}{f'(z_0)} \right\rangle_n$
4	$n \leftarrow n + 1$
5	while ($ w \leq 2^{-n+1}$)
6	Return ($n - 1$)

Note that $\alpha(f, z_0) < 0.03$ implies $u_{z_0} \leq \frac{3-\sqrt{7}}{2}$. Hence $\psi(u_{z_0}) \geq \frac{1}{2}$, and the above lemma gives us

$$\frac{\delta}{2} \leq |z_0 - z^*| \leq 2\delta. \tag{12}$$

We then conclude that Algorithm D produces the necessary constant C for robust iteration:

Lemma 6. *Let $C := n + 2$, where $n - 1$ is the value returned by Algorithm D on an approximate zero z_0 , $\alpha(f, z_0) < 0.03$, with z^* as the associated root. Then*

$$2^{-C} \leq |z_0 - z^*| \leq 6.2^{-C+2}.$$

Basically, Algorithm D is converting absolute precision into relative precision. Algorithm D takes $(-\lg \delta) + O(1)$ steps of evaluation. But using the geometric searching method in [AKY04], we can further reduce the number of evaluation steps to $2 \lg \lg(1/\delta) + O(1)$. For the purposes of this exposition we present the simpler version, however the complexity result below is based upon the geometric search method.

5 Complexity of Approximating a Zero of a Polynomial

Let $f(z)$ be a degree d square-free polynomial with L -bit integer coefficients. Furthermore, let $\text{sep}(f, z^*)$ be the distance between z^* and the nearest root of f different from z^* . Suppose z_0 is a robust approximate zero of f , with associated zero z^* and satisfying $\alpha(f, z_0) < 0.02$. Note that there are well-known methods for computing such a z_0 (either real or complex). Starting from such a z_0 , our goal is to compute an n -bit absolute approximation $\langle z^* \rangle_n$ for z^* .

Our analysis here will focus will be on the case when $z^* \in \mathbb{R}$ and z_0 is a bigfloat. It is easy to check that $|\tilde{z}_i - z^*| \leq 2^i$ provided $i \geq \lg(n + 1 + \lg |z_0 - z^*|)$. From Cauchy's bound we may assume that $|z_0| \leq 2^L$. Thus $|z_0 - z^*| \leq 2^{L+1}$ which means we require at most $\lg(n + L + 2)$ steps of Newton iteration. The complete algorithm which returns $\langle z^* \rangle_n$, given z_0 , is as follows:

- Compute C satisfying (5) using Algorithm D above. Let $\tilde{z}_0 := z_0$.
- For $i = 0, \dots, \lg(n + L + 2)$ do the following:

1. $x := \langle f(\tilde{z}_{i-1}) \rangle_{2^i + C + 1 - \lg \kappa}$.
 2. $y := \langle f'(\tilde{z}_{i-1}) \rangle_{2^i + 2 - \lg \kappa}$.
 3. $\tilde{z}_i := \tilde{z}_{i-1} - \left\lfloor \frac{x}{y} \right\rfloor_{2^i + 1}$.
- Return \tilde{z}_i .

To carry out the complexity estimates, we need some basic complexity bounds for unbounded bigfloats:

Lemma 7. *Let $x = \langle e_x, f_x \rangle, y = \langle e_y, f_y \rangle$ be bigfloats, and n be a positive natural number. Also, $f_x f_y \neq 0$.*

1. We can compute $[x]_n$ in $O(n + \lg(2 + |e_x|))$ time.
2. We can compute $[xy]_n$ in $O(M(n) + \lg(2 + |e_x e_y|))$ time.
3. We can compute $[x + y]_n$ in $O(n + \lg(2 + |e_x e_y|))$ time provided $xy \geq 0$ or $|x| > 2|y|$ or $|x| < |y|/2$. In general, computing $[x + y]_n$ can be done in time $O(\lg(2 + |f_x f_y e_x e_y|))$.
4. An analogous statement holds for $[x - y]_n$, where we replace $xy \geq 0$ by $xy \leq 0$.

Next consider the evaluation of polynomial to arbitrary absolute precision: let $f(x) = \sum_{i=0}^d a_i x^i$ and suppose the a_i 's and x are bounded by e_i 's and e_x as follows:

$$2^{e_i} \leq |a_i| < 2^{e_i+1}, \quad 2^{e_x} \leq |x| < 2^{e_x+1}.$$

Also, let $e := \max\{e_0, e_1, \dots, e_d\}$.

Lemma 8. *We can evaluate $f(x)$ to absolute precision n in time*

$$O(dM(n + |e| + d|e_x|)).$$

We now bound the complexity of Algorithm D. Let $f(z)$ be a degree d integer polynomial with L -bit coefficients. Further assume that z_0 is a rational number with s -bit numerator and denominator. Then we have the following:

Lemma 9. *Let the bigfloat z_0 be an approximate zero, $\alpha(f, z_0) < 0.03$, whose exponent has bit size s . Then the geometric version of Algorithm D has complexity $O(M(d(L + s)))$.*

Finally, we show:

Theorem 4. *Let $f(z) = \sum_{i=0}^d a_i z^i$ be a polynomial such that $|a_i| \leq 2^L$. Suppose we are given a bigfloat z_0 satisfying $\alpha(f, z_0) \leq 0.02$. So z_0 is a robust approximate zero and let its associated root be z^* . Then we can compute an n -bit absolute approximation $\langle z^* \rangle_n$ of z^* in time*

$$O[dM(n + d^2(L + \lg d) \lg(n + L))]. \tag{13}$$

If d, L are bounded then the complexity is $O(M(n))$.

This result may be regarded as a generalization of Brent's bounded precision bound [Bre76a–Lem. 3.1].

6 Conclusion and Future Work

The key contribution of this paper is the development of the concept of robust approximate zero and robust point estimates. We improve on Malajovich's work by obtaining explicit complexity bounds and a stronger point estimate in the univariate case. We plan to implement the robust Newton iteration in Core library; the current implementation is in the weak mode.

We plan to extend the above work in the following directions: to multi-variate Newton iteration, and to multiple zeros. For the latter problem, Yakoubsohn [Yak03] has obtained results under the exact arithmetic setting. Brent has given the complexity of approximating a simple zero of a non-linear equation in the bounded bigfloat setting. We also plan to extend this to the unbounded robust setting.

Acknowledgements. The authors would like to thank an anonymous referee for meticulous and invaluable feedback.

References

- [AKY04] Tetsuo Asano, David Kirkpatrick, and Chee Yap. Pseudo approximation algorithms, with applications to optimal motion planning. *Discrete and Computational Geometry*, 31(1):139–171, 2004. Special Conference Issue from 18th ACM Symp. of Comput. Geom., 2002.
- [Bat98] Prashant Batra. Improvement of a convergence condition for the Durand-Kerner iteration. *J. of Comp. and Appl. Math.*, 96:117–125, 1998.
- [BCSS98] Lenore Blum, Felipe Cucker, Michael Shub, and Steve Smale. *Complexity and Real Computation*. Springer-Verlag, New York, 1998.
- [BFMS99] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. Efficient exact geometric computation made easy. In *Proc. 15th ACM Symp. Comp. Geom.*, pages 341–450, New York, 1999. ACM Press.
- [Bre76a] Richard P. Brent. Fast multiple-precision evaluation of elementary functions. *J. of the ACM*, 23:242–251, 1976.
- [Bre76b] Richard P. Brent. Multiple-precision zero-finding methods and the complexity of elementary function evaluation. In J. F. Traub, editor, *Proc. Symp. on Analytic Computational Complexity*, pages 151–176. Academic Press, 1976.
- [CSY97] J. Choi, J. Sellen, and C. Yap. Approximate Euclidean shortest paths in 3-space. *Int'l. J. Comput. Geometry and Appl.*, 7(4):271–295, 1997. Also: 10th ACM Symp. on Comp. Geom. (1994)pp.41–48.
- [DF95] Wang Deren and Zhao Fengguang. The theory of Smale's point estimation and its applications. *J. of Comp. and Appl. Math.*, 60:253–269, 1995.
- [Dur60] E. Durand. *Solutions Numériques des Équations Algébriques, Tome I: Equations du Type $F(x) = 0$. Racines d'un Polyôme*, Masson, Paris, 1960.
- [Hig96] Nicholas J. Higham. *Accuracy and stability of numerical algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, 1996.

- [KA64] L.V. Kantorovich and G.P. Akilov. *Functional Analysis in Normed Spaces*. New York, MacMillan, 1964.
- [Kal05] Bahman Kalantari. An infinite family of bounds on zeros of analytic functions and relationship to Smale's bound. *Mathematics of Computation*, 74(250):841–852, 2005.
- [Ker66] I.O. Kerner. Ein Gesamtschrittverfahren zur Berechnung der Nullstellen von Polynomen. *Numer. Math.*, 8:290–294, 1966.
- [Kim86] Myong-Hi Kim. *Computational Complexity of the Euler Type Algorithms for the Roots of polynomials*. PhD thesis, City University of New York, January 1986.
- [Kim88] Myong-Hi Kim. On approximate zeroes and root finding algorithms for a complex polynomial. *Math. Comp.*, 51:707–719, 1988.
- [KLPY99] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A Core library for robust numerical and geometric computation. In *15th ACM Symp. Computational Geometry*, pages 351–359, 1999.
- [Mal93] Gegorio Malajovich. *On the complexity of path-following Newton algorithms for solving systems of polynomial equations with integer coefficients*. PhD thesis, Berkeley, 1993.
- [Mal94] Gregorio Malajovich. On generalized Newton algorithms: Quadratic convergence, path-following and error analysis. *Theoretical Computer Science*, 133:65–84, 1994.
- [MS01] Kurt Mehlhorn and Stefan Schirra. Exact computation with `leda_real` – theory and geometric applications. In G. Alefeld, J. Rohn, S. Rump, and T. Yamamoto, editors, *Symbolic Algebraic Methods and Verification Methods*, volume 379, pages 163–172, Vienna, 2001. Springer-Verlag.
- [Pan97] Victor Y. Pan. Solving a polynomial equation: some history and recent progress. *SIAM Review*, 39(2):187–220, 1997.
- [PCT95] Miodrag S. Petković, Carsten Carstensen, and Miroslav Trajković. Weierstrass formula and zero-finding methods. *Numer. Math.*, 69:353–372, 1995.
- [PHI98] Miodrag S. Petković, Dorde Herceg, and Snežana Ilić. Safe convergence of simultaneous methods for polynomial zeros. *Numerical Algorithms*, 17:313–331, 1998.
- [Sch80] A. Schönhage. Storage modification machines. *SIAM J. Computing*, 9:490–508, 1980.
- [Sma86] S. Smale. Newton's method estimates from data at one point. In R. Ewing, K. Gross, and C. Martin, editors, *The Merging of Disciplines: New Directions in Pure, Applied, and Computational Mathematics*. Springer-Verlag, 1986.
- [SS85] Mike Shub and Steven Smale. Computational Complexity: On the Geometry of Polynomials and a Theory of Cost. I. *Annales Scientifiques De L'É.N.S.*, 4(18):107–142, 1985.
- [SS93] Mike Shub and Steve Smale. Complexity of Bezout's Theorem I: Geometric aspects. *J. of Amer. Math. Soc.*, 6(2):459–501, 1993.
- [Tis01] Françoise Tisseur. Newton's method in floating point arithmetic and iterative refinement of generalized eigenvalue problems. *SIAM J. on Matrix Anal. and Appl.*, 22(4):1038–1057, 2001.
- [Yak03] Jean-Claude Yakoubsohn. Numerical Elimination, Newton Method and Multiple Roots. In Frédéric Chyzak, editor, *Algorithms Seminar, 2001-2002*, number 5003, Rapport de recherche, INRIA, pages 49–54. Nov. 2003.

- [Yam85] T. Yamamoto. A unified derivation of several error bounds for Newton's process. *Journal of Comp. and Appl. Mathematics*, 12&13:179–191, 1985.
- [Yam86] T. Yamamoto. Error bounds for Newton's method under the Kantorovich assumptions. In R. Ewing, K. Gross, and C. Martin, editors, *The Merging of Disciplines: New Directions in Pure, Applied, and Computational Mathematics*. Springer-Verlag, 1986.
- [Yap04] Chee K. Yap. On guaranteed accuracy computation. In Falai Chen and Dongming Wang, editors, *Geometric Computation*, chapter 12, pages 322–373. World Scientific Publishing Co., Singapore, 2004.
- [Ypm83] T.J. Ypma. The effect of rounding errors on Newton-like methods. *IMA J. of Numerical Analysis*, 3:109–118, 1983.
- [Ypm84] T.J. Ypma. Local convergence of inexact Newton methods. *SIAM J. of Numer. Anal.*, 21(3):583–590, 1984.

Optimizing a 2D Function Satisfying Unimodality Properties

Erik D. Demaine¹ and Stefan Langerman^{2,*}

¹ MIT Computer Science and Artificial Intelligence Laboratory,
32 Vassar Street, Cambridge, MA 02139, USA
`edemaine@mit.edu`

² Département d’informatique, Université Libre de Bruxelles,
ULB CP212, Bruxelles, Belgium
`Stefan.Langerman@ulb.ac.be`

Abstract. The number of probes needed by the best possible algorithm for locally or globally optimizing a bivariate function varies substantially depending on the assumptions made about the function. We consider a wide variety of assumptions—in particular, global unimodality, unimodality of rows and/or columns, and total unimodality—and prove tight or nearly tight upper and lower bounds in all cases. Our results include both nontrivial optimization algorithms and nontrivial adversary arguments depending on the scenario.

1 Introduction

Many problems in geometry, in particular problems about the set of distances among geometric objects (diameter, closest pairs, farthest pairs, etc.) can be seen as finding a maximum in a two-dimensional array. This abstraction is used by many algorithms, but one of the most remarkable results is probably the $O(n)$ -time algorithm for optimizing “totally monotone” $n \times n$ matrices of Aggarwal et al. [1] and the application of this algorithm to many geometric proximity problems. This work later found many applications as a general technique for speeding up dynamic-programming algorithms. A survey of these applications can be found in [4]. The motivation for our work came from the desire to understand what matrix properties enable speeding a search from linear time down to a polylogarithmic number of probes. For example, we want to know the weakest properties that would have to be expressed in order to find the closest pair of points between two given convex polygons in logarithmic time [3]. Such an understanding could lead to many generalizations, for example to other metric spaces or variants of convexity.

The most general formulation of this discrete optimization problem is to maximize a given function $f : D \rightarrow \mathbb{R}$ over a discrete (finite) domain D . In general, of course, this problem may require $|D|$ probes to f . One approach to making optimization more tractable is to be satisfied with finding a *local* maximum, i.e., a point at which f attains a value larger than all “neighboring” points, for some definition

* Chercheur qualifié du FNRS.

of neighborhoods. In particular, for the standard 1D domain $D = \{1, 2, \dots, n\}$, Fibonacci search [6] finds a local maximum using $\log_\phi n + O(1)$ probes, where $\phi = (1 + \sqrt{5})/2$ is the golden ratio. Surprisingly, the problem complexity grows exponentially in 2D, even for a square domain $D = \{1, 2, \dots, n\} \times \{1, 2, \dots, n\}$: independently, Llewellyn et al. [7,8], Althöfer and Koschnick [2], and Mityagin [9] proved that $\Theta(n)$ probes to a function f are sufficient and sometimes necessary to find any local optimum in an $n \times n$ array D . (Unless otherwise specified, we use the 4-neighborhood $\{(i-1, j), (i+1, j), (i, j-1), (i, j+1)\}$ of a point (i, j) in the square grid.) Thus weakening the optimization problem to finding local maxima does not provide an exponential speedup in higher dimensions like it did in 1D. See also [10] for a survey on local optimization methods.

Another approach to making optimization more tractable is to add assumptions about the function f . Other than the monotonicity assumptions mentioned above, the main example in the literature of which we are aware is a kind of Lipschitz condition: if f is *integral Lipschitz* in the sense that, between two neighboring points x and y , $f(x)$ and $f(y)$ are integral and differ by at most L , then it is possible to find a local maximum in $O(L \log n)$ probes [2]. Another simple example is that, if we assume that f is *unimodal* (denoted “ \odot unimodal”), i.e., it has exactly one local maximum, then finding local maxima and finding global maxima are equivalent. One could hope that having this structural information about the function would also help in finding that maximum. Unfortunately, a careful reading of the construction in [9] of 2D functions f requiring $\Theta(n)$ probes reveals they are in fact \odot unimodal.

We study the related condition that the 2D function f is unimodal in every column (\downarrow *unimodal*) and/or in every row (\leftrightarrow *unimodal*). These properties are satisfied by e.g. convex functions, but are more general: for example, the distance function between a point on a convex chain and a point on a monotone chain satisfies one of these properties (in fact, it is \downarrow convex) but not the other. While seemingly weaker than \odot unimodality, these properties are incomparable to unimodality, and in fact result in exponential speedup for finding local maxima. We also study the stronger condition that a function is *totally unimodal* in the sense that every submatrix is \odot unimodal. This property is the first that allows us to find the (unique) local maximum of the array in $O(\log n)$ probes. Table 1 summarizes all of our results.

A notion related to a totally unimodal matrix is a *unique-sink orientation* of the $m \times n$ grid graph, as considered for arbitrary-dimensional grids in [5]. However, the latter notion is less restrictive: essentially, unique-sink orientations capture only relative comparisons between adjacent vertices, whereas total unimodality captures comparisons between arbitrary vertices in a total order. The relative comparisons of unique-sink orientations may not even be realizable by a total order because of directed cycles in the orientation. When restricted to the two-dimensional case, the algorithms of [5] have running time $O(m+n)$; our totally unimodal algorithms are exponentially faster (but less general).

In the next section, we show how total properties of matrices can be expressed as a set of forbidden partial orders in submatrices. This characterization allows

Table 1. Worst-case bounds on the number of probes required to maximize a function $f : \{1, 2, \dots, m\} \times \{1, 2, \dots, n\} \rightarrow \mathbb{R}$. In the bounds, $max = \max\{m, n\}$ and $min = \min\{m, n\}$.

Assumption	Local optimization	Global optimization
None	$\leq min \cdot (\lg \frac{max}{min} + 4) + O(\lg max)$ $\geq \min\{min, max/2\}$	[9] $\leq m \cdot n$ [obvious] [9] $\geq m \cdot n$ [obvious]
Totally +j monotone	$\leq min \cdot (\lg \frac{max}{min} + 4) + O(\lg max)$ $\geq \log_\phi max$ [Lem. 7]	[9] $O(\min(1 + \lg \frac{max}{min}))$ [1] [9] $\Omega(\min(1 + \lg \frac{max}{min}))$ [1]
⊙ unimodal	$\leq min \cdot (\lg \frac{max}{min} + 4) + O(\lg max)$ $\geq \min\{min, max/2\}$	[9] same as local [9]
↓ unimodal	$\leq \log_\phi m \log_\phi n + O(\lg n)$ [Lem. 8] $\geq \frac{1}{4} \lg m \lg n - O(\lg m \lg \lg n)$ if $m \leq n$ [Thm. 1] $\geq \frac{1}{4} \lg^2 n - O(\lg n \lg \lg n)$ if $m \geq n$ [Thm. 1]	$\leq n \log_\phi m + O(n)$ [Lem. 12] $\geq n \log_\phi m - O(n)$ [Lem. 13]
⊙, ↓ unimodal	same as ↓ unimodal	same as local
↑, ↔ unimodal	$\leq \frac{3}{\lg \phi} \lg^2 min + O(\lg max)$ [Thm. 2] $\geq \frac{1}{4} \lg^2 min - O(\lg max \lg \lg max)$ [Thm. 3] $\geq \log_\phi max$ [Lem. 7]	$\leq \min(\log_\phi max + O(1))$ [Lem. 12] $\Omega(min)$ [Lem. 14]
⊙, ↑, ↔ unimodal	same as ⊙, ↓ unimodal	same as local
⊙, ↑, ↔ unimodal & totally +i, +j monotone	same as ⊙, ↑, ↔ unimodal	same as local
Totally unimodal	$O(\lg max)$ [Thm. 4] $\geq \log_\phi max$ [Lem. 7]	same as local

us to determine easily which combinations of properties imply which others. We then proceed to present nearly tight bounds for finding a local or global maximum for most combinations of properties: in ↓ unimodal functions (Section 4), in ↑, ↔ unimodal functions (Section 5), and in totally unimodal functions (Section 6). Finally, in Section 7, we analyze a natural random probing strategy and show that it falls in between the last two strategies.

2 Forbidden Submatrix Partial Orders

In this section we show how several properties of real-valued functions on the $m \times n$ grid, or equivalently a real $m \times n$ matrix, can be expressed by finite forbidden substructures.

For any matrix property \mathcal{P} , we say that a matrix is *totally* \mathcal{P} if every of its submatrices has property \mathcal{P} . In this section, we show how many total properties for matrices can be expressed as a finite set of constant-size partial orders that are forbidden to occur in any submatrix. This characterization of total properties gives an easy way to determine which combination of properties imply which other.

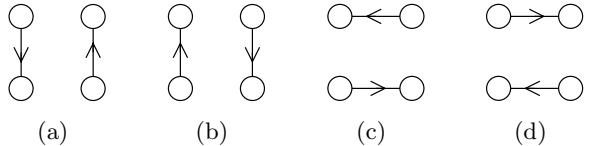


Fig. 1. Forbidden 2×2 submatrices for total monotonicity. Arrows point to larger elements.

Monotone. Let $i(j)$ be the row index of the maximum in column j . A matrix is $+j$ monotone if $j \leq j'$ implies $i(j) \leq i(j')$. A matrix is *totally $+j$ monotone* if every submatrix is $+j$ -monotone. It can be shown [1] that it is sufficient to consider only 2×2 matrices. Thus to obtain the class of totally $+j$ monotone matrices, we just have to forbid the configuration shown in Figure 1(a).

Total monotonicity can be defined in all four directions: $+j$, $-j$, $-i$, and $+i$. The corresponding four forbidden configurations are shown in Figure 1(a–d).

A matrix is *totally monotone* if it forbids any one of these four configurations.

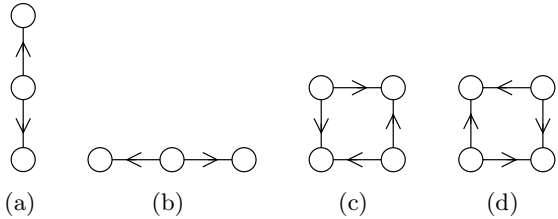


Fig. 2. Forbidden submatrices for total unimodality. (a) \updownarrow unimodality; (b) \leftrightarrow unimodality.

\updownarrow or \leftrightarrow *Unimodal.* Note that \updownarrow or \leftrightarrow unimodality are total properties. Each property has

a single forbidden configuration, as shown in Figure 2(a, b). Of course, $\updownarrow, \leftrightarrow$ unimodality is given by forbidding both of these configurations.

Totally Unimodal. A matrix is *totally unimodal* if every submatrix is \odot unimodal, i.e., every submatrix has a unique local maximum. This property has four forbidden configurations, shown in Figure 2(a, b, c, d).

Lemma 1. *If a matrix is $\updownarrow, \leftrightarrow$ unimodal and totally $+j, -i$ monotone (or totally $-j, +i$ monotone), then it is totally unimodal.*

Lemma 2. *A matrix is totally unimodal if and only if it is $\updownarrow, \leftrightarrow$ unimodal, and every 2×2 submatrix is \odot unimodal.*

Corollary 1. *If a matrix is totally unimodal, then it is $\odot, \updownarrow, \leftrightarrow$ unimodal.*

3 Elimination Lemmas

This section develops a battery of lemmas for guaranteeing that the solution we desire is not in a particular region, or more precisely, that at least one desired solution is in the remaining region. Different lemmas apply to different scenarios of assumptions made on the function, while one lemma is generic.

We consider the following more general (non-matrix) setting. A *discrete domain* D is a finite set along with a notion of adjacency (defined by a graph on the finite set). As mentioned, we mainly focus on the square grid domain $D = \{1, 2, \dots, m\} \times \{1, 2, \dots, n\}$, primarily with 4-adjacency—two points are adjacent if their ℓ_1 distance is 1—but several of our definitions and basic results apply more generally.

A *local maximum* of a function $f : D \rightarrow \mathbb{R}$ is a point p of the domain D such that all points adjacent to p have strictly smaller f values than p . In other

words, a point is a local maximum if all incident edges (adjacencies) are *downhill* (in f). In this paper we assume that adjacent points have distinct f values; otherwise, a constant function f satisfies all (nonstrict) unimodality properties but is impossible to optimize in less than $|D|$ probes.

The following lemmas allow us to restrict the region in which we must search for a local maximum. In particular, given various configurations and/or unimodality assumptions on f , our goal is to identify elements that are effectively *eliminated* by a constant number of probes, in the sense that the remaining uneliminated region contains a local maximum.

The first lemma is useful in particular when the region in which we are searching disconnects into multiple components. In general, a *region* R of a discrete domain D is a subset of D . The *skin* of a region R is the set of points in the domain D that are not in R but are adjacent to points in R .

Lemma 3. *For a function $f : D \rightarrow \mathbb{R}$, if the maximum f value over a region R of the domain D is larger than the f values of all points on the skin of R , then R contains a local maximum of f .*

The next lemma shows that, in the \uparrow -unimodal case, whenever an algorithm makes a probe it can probe a vertically adjacent point (losing at most a factor of 2 in probe count) and eliminate either the top or bottom “half” of the column, depending on which of the two points has a larger f value. Define $(\leq i, j) = \{(i', j) \mid i' \leq i\}$ and similarly for $(?i, j)$ and $(i, ?j)$ for $? \in \{\leq, \geq, <, >\}$.

Lemma 4. *Suppose $f : \{1, 2, \dots, m\} \times \{1, 2, \dots, n\} \rightarrow \mathbb{R}$ is \uparrow unimodal and suppose that region R contains a local maximum. If $f(i, j) > f(i + 1, j)$, then $R \setminus (> i, j)$ contains a local maximum. Similarly, if $f(i, j) < f(i + 1, j)$, then $R \setminus (\leq i, j)$ contains a local maximum.*

The next lemma shows an analogous result for $\odot, \updownarrow, \leftrightarrow$ unimodality, except that the constant factor loss is now at most 3, and the eliminated elements are nearly an entire quadrant. (The entire quadrant can be eliminated at a cost of at most a factor of 5.) Define $(\leq i, \leq j) = \{(i', j') \mid i' \leq i, j' \leq j\}$ and similarly for $(?i, ?j)$ for $?, i, j \in \{\leq, \geq, <, >\}$.

Lemma 5. *Suppose $f : \{1, 2, \dots, m\} \times \{1, 2, \dots, n\} \rightarrow \mathbb{R}$ is $\odot, \updownarrow, \leftrightarrow$ unimodal and suppose that region R contains a local maximum. Consider a point (i, j) in R . If $f(i, j) > f(i + 1, j)$ and $f(i, j) > f(i, j + 1)$, then $R \setminus [(\geq i, \geq j) - (i, j)]$ contains a local maximum. (Unless (i, j) is also a local maximum, i.e., we also have $f(i, j) > f(i - 1, j)$ and $f(i, j) > f(i, j - 1)$, even $R \setminus (\geq i, \geq j)$ contains a local maximum.) Similarly, if $f(i, j) > f(i + 1, j)$ and $f(i, j) < f(i, j + 1)$, then $R \setminus (\geq i, \leq j)$ contains a local maximum; if $f(i, j) < f(i + 1, j)$ and $f(i, j) > f(i, j + 1)$, then $R \setminus (\leq i, \geq j)$ contains a local maximum; and if $f(i, j) < f(i + 1, j)$ and $f(i, j) < f(i, j + 1)$, then $R \setminus (\leq i, \leq j)$ contains a local maximum.*

Finally, we prove a more powerful quadrant elimination lemma for totally unimodal functions, where we can compare to nonadjacent points because total unimodality allows us to consider induced submatrices.

Lemma 6. *Suppose $f : \{1, 2, \dots, m\} \times \{1, 2, \dots, n\} \rightarrow \mathbb{R}$ is totally unimodal and suppose that region R contains a local maximum. Consider two points (i, j) and (i', j') with $i < i'$ and $j < j'$, suppose that R is already disjoint of the cornerless quadrants $(\leq i, \leq j) - (i, j)$ and $(\geq i', \geq j') - (i', j')$, and suppose that $f(i - 1, j) < f(i, j)$ and $f(i, j - 1) < f(i, j)$. If $f(i', j) > f(i, j)$ and $f(i', j) > f(i', j')$, then $R \setminus (\leq i, \geq j')$ contains a local maximum. If $f(i, j') > f(i, j)$ and $f(i, j') > f(i', j')$, then $R \setminus (\geq i', \leq j)$ contains a local maximum. If neither of these conditions hold, then $R \setminus ((\leq i, \geq j') \cup (\geq i', \leq j))$ contains a local maximum.*

Before proceeding to more difficult upper and lower bounds, we prove a simple logarithmic lower bound in the most specific case of totally unimodal functions:

Lemma 7. *Any comparison-based algorithm for finding a local maximum in a totally unimodal function must make at least $\log_\phi \max\{m, n\} - O(1)$ probes in the worst case.*

4 \updownarrow Unimodal

4.1 Local Optimization

Lemma 8. *There is an algorithm that, given a \updownarrow unimodal $m \times n$ matrix, finds a local optimum after $\leq \log_\phi n \log_\phi m + O(\log n)$ probes.*

Theorem 1. *For every algorithm that correctly finds a local optimum in an $m \times n \updownarrow$ unimodal matrix, there is an adversary that (a) generates a \updownarrow unimodal function with a unique local optimum, and (b) forces the algorithm to make $\geq \frac{1}{4} \lg m \lg n - O(\lg m \lg \lg n)$ probes if $m \leq n$, and $\geq \frac{1}{4} \lg^2 n - O(\lg n \lg \lg n)$ if $m > n$.*

Proof. The adversary gives the algorithm extra information, which can only help. Whenever the algorithm probes the value at a particular point (i, j) , the adversary reveals not only that value, but also the slope of that value in that column, i.e., whether the mode in that column j is above or below that point (i, j) . Furthermore, if the mode of column j is above the probe point (i, j) , then the adversary reveals all values in the column j below the point (i, j) ; symmetrically, if the mode is below the probe point, the adversary reveals all values above the point in its column. If the algorithm discovers the mode of column j , the adversary reveals all values in the column j . Thus we maintain the invariant that every column that is not totally revealed has some revealed values in the topmost few rows, some revealed values in the bottommost few rows, and the algorithm knows that the mode of the column is somewhere in between.

If the unrevealed region ever becomes disconnected, the adversary reveals all values in all connected components except the largest connected component. Thus we maintain the invariant that the unrevealed region is connected. We also maintain the invariant that the algorithm cannot discover the unique local

optimum until every value has been revealed. Together these two invariants make the goal of the algorithm to disconnect the unrevealed region; otherwise, the algorithm must make at least one probe per column, for a total of at least n probes.

The main task of the adversary is to decide whether a probe point is above or below the mode of that column, and then to choose the revealed values below or above the probe point. The adversary bases its decision on matching the “nearest” previous decision, according to the ℓ_1 distance function. Naturally, the distance between a point (i, j) and the top horizontal wall is i , and the distance to the bottom wall is $m + 1 - i$.

Suppose that the algorithm probes the point (i, j) . If point (i, j) is closer to a horizontal wall than every revealed point, then the adversary reveals all values in column j between (i, j) and the nearest wall, specifying that the mode is in the other direction. Otherwise, the adversary specifies (i, j) to be above or below the mode in its column j according to whether the revealed point (i^*, j^*) nearest to (i, j) is above or below the mode in its column j^* . Then the adversary reveals all unrevealed values starting from (i, j) in the opposite direction to the mode in column j . (In the special case described below that the algorithm discovers the mode among these revealed values, the specification that the mode is above or below (i, j) is false; in this case the adversary reveals all values in column j .)

The adversary chooses the revealed values as follows. Suppose that the algorithm probes (i, j) and say that the adversary decides that probe point (i, j) is below the mode in its column j . If the to-be-revealed points keep the unrevealed region connected, then the adversary repeatedly reveals that the bottommost unrevealed value in column j is one more than the largest previously revealed value, until reaching point (i, j) . In this way the revealed values increase in an integer sequence from the bottommost unrevealed value to (i, j) . Equivalently, the adversary reveals every unrevealed point (i', j) below (i, j) in column j to have value $m - d$ more than the largest previously revealed value, where $d = i - i'$ is the Manhattan distance between the unrevealed point (i', j) and the probe point (i, j) .

On the other hand, if the to-be-revealed points disconnect the unrevealed region, then we either keep unrevealed the component left of column j or the component right of column j , whichever has the largest number of unrevealed columns. Assume the component left of column j is to be kept unrevealed, and let j' be the rightmost unrevealed column in the matrix. We reveal the entries in the columns from column j' to column j as follows: when revealing the entries of column j'' , $j' \geq j'' \geq j$, we identify an entry (i'', j'') adjacent to an unrevealed entry in column $j'' - 1$. We set (i'', j'') to be the mode of column j'' , and reveal all entries in that column, by repeatedly revealing the topmost entry with one more than the previously revealed value until $(i'' - 1, j'')$ is revealed, then repeatedly revealing the bottommost entry with one more than the previously revealed value until (i'', j'') is revealed, then proceed to reveal the entries of column $j'' - 1$ in the same manner, until column j is completely revealed. This strategy ensures that whenever a point is revealed, it is connected to a yet unrevealed point, and

so there is an increasing path from any entry in the table to the unique local optimum which is the last value to be revealed. Thus we obtain:

Lemma 9. *The only point to become a local maximum according to the adversary is the mode of the final column to become completely revealed.*

Lemma 10. *The algorithm must make $\min\{n, \lg m\}$ probes before the unrevealed region first disconnects into multiple connected components.*

Lemma 11. *The nearest point or horizontal wall to a point (i, j) is in a column j' such that $|j - j'| \leq m$.*

Finally we conclude the proof of Theorem 1. Consider an algorithm that makes fewer than $\lg n \lg m$ probes. As mentioned above, the algorithm must disconnect the unrevealed region or else it is doomed to make at least n probes. Lemma 10 says that the algorithm must make at least $\min\{n, \lg m\}$ probes for the first disconnection. Consider the final probe that caused the disconnection. By the pigeon-hole principle, the $(\lg n \lg m)m$ consecutive columns including and to the right of this final probe must have a gap of at least m consecutive empty columns, because there are at most $\lg n \lg m$ probes total. We remove columns starting from the final probe up to but not including this gap of m consecutive empty columns. Similarly, we remove at most $(\lg n \lg m)m$ columns to the left of the final probe up to but not including a gap of m consecutive empty columns. Thus we obtain two subproblems (one left and one right) that by Lemma 11 act completely independently from each other and from the probes causing the disconnection, as far as probes made so far. We recursively consider the subproblem corresponding to the larger connected component that remains. This recursive subproblem is a rectangle with m rows and $n' \geq \lfloor n/2 \rfloor - (\lg n \lg m)m$ columns. The recursive subproblem may have already been probed, but we can consider such probes as happening after this subproblem. Thus the recursion applies until $n'/2 < (\lg n \lg m)m$.

Therefore we obtain the lower bound of $\min\{n', \lg m\}$ probes, where $n' \geq 2(\lg n \lg m)m$, at each of $\lg(n/(2(\lg n \lg m)m))$ levels of recursion. In total we obtain a lower bound of $(\lg m)(\lg(n/m) - 1 - \lg \lg n - \lg \lg m) \geq \lg m \lg(n/m) - O(\lg m \lg \lg n)$. If $m \leq \sqrt{n}$, then the lower bound is $\geq \frac{1}{2} \lg m \lg n - O(\lg m \lg \lg n)$. If $m > \sqrt{n}$, we perform the same argument on a submatrix with $m' = \sqrt{n}$ rows. The lower bound then becomes $\geq \lg m' \lg(n/m') - O(\lg m' \lg \lg n) \geq \frac{1}{4} \lg^2 n - O(\lg n \lg \lg n)$. In particular, if $\sqrt{n} \leq m \leq n$, we obtain the lower bound $\geq \frac{1}{4} \lg m \lg n - O(\lg m \lg \lg n)$. \square

4.2 Global Optimization

Lemma 12. *There is an algorithm that, given a \uparrow unimodal $m \times n$ matrix, finds its global optimum after at most $n \log_\phi m + O(n)$ probes.*

Lemma 13. *Any algorithm that, given a \uparrow unimodal $m \times n$ matrix finds its global optimum must perform at least $n \log_\phi m - O(n)$ probes.*

5 $\updownarrow, \leftrightarrow$ Unimodal

Theorem 2. *There is an algorithm that finds a local optimum in a $\updownarrow, \leftrightarrow$ unimodal $m \times n$ matrix after at most $(3/\lg \phi) \lg^2 \min + O(\lg \max)$ probes.*

Proof. Assume without loss of generality that $m \leq n$. First find the maximum element on row $m/2$, among elements in columns $in/m, i = 1, \dots, m$, in $\lg_\phi m$ time. (Ratios are implicitly rounded to integers, affecting only lower-order terms.) This finds two elements jn/m and $(j + 1)n/m$ on columns separated by n/m elements. We can now eliminate from the search one of the two quadrants left of $(m/2, jn/m)$ and one of the two quadrants right of $(m/2, (j + 1)n/m)$. Then find the maximum on columns jn/m and $(j + 1)n/m$ using Fibonacci search, and evaluate the right and left neighbors of those two maxima. We now know a local max is either (I) to the left of column jn/m , (II) between columns jn/m and $(j + 1)n/m$ or (III) to the right of column $(j + 1)n/m$. Since one quadrant has been eliminated to the left of column jn/m and one quadrant has been eliminated to the right of column $(j + 1)n/m$, the size of the submatrix to recurse in is $(m/2) \times n$ in cases (I) and (III), or $m \times (n/m)$ in case (II). Thus, cases (I) and (III) can only happen $\lg m$ times, and case (II) can only happen $\lg n/\lg m$ times. Each step performs $3 \lg_\phi m + O(1)$ probes, so the total number of probes is $(3/\lg \phi)(\lg^2 m + \lg n) + O(1)(\lg m + \lg n/\lg m)$. \square

Theorem 3. *For every algorithm that correctly finds a local maximum in an $m \times n \updownarrow, \leftrightarrow$ unimodal, totally $+j, +i$ monotone matrix, there is an adversary that (a) generates such a function with a unique local maximum, and (b) forces the algorithm to make $\frac{1}{4} \lg^2 \min - O(\lg \min \lg \lg \min)$ probes.*

Lemma 14. *Any algorithm that, given a $\updownarrow, \leftrightarrow$ unimodal $m \times n$ matrix finds its global maximum must perform at least $\min\{n, m\}$ probes.*

6 Totally Unimodal

Theorem 4. *There is an algorithm that, given a totally unimodal $m \times n$ matrix, finds its global maximum after $O(\lg n + \lg m)$ probes.*

Proof. The algorithm performs successive probes and eliminates regions of the matrix known not to contain the local maximum. At every step of the algorithm, the unrevealed region will be a cross inside a submatrix, i.e., the algorithm maintains four indices i_1, i_2, j_1, j_2 , with $i_1 + 1 < i_2$ and $j_1 + 1 < j_2$ such that the unique local maximum is known not to be in the quadrants $(\leq i_1, \leq j_1), (\geq i_2, \leq j_1), (\leq i_1, \geq j_2), (\geq i_2, \geq j_2)$. Furthermore, we maintain the invariant that the apex of each of those four quadrants is the maximum value in the quadrant, e.g. for the first quadrant, that $f(i_1 - 1, j_1) < f(i_1, j_1)$ and $f(i_1, j_1 - 1) < f(i_1, j_1)$. We call this the *apex invariant*. The rectangular area with corners $(i_k + 1, j_l + 1)$ for $k, l \in \{1, 2\}$ is called the *center* of the cross, and is surrounded by four *legs*. We will sometimes refer to T shapes or L shapes instead of the cross, those are just crosses for which one or two of the legs is empty, respectively.

The algorithm then performs a constant number of probes which will reduce the unrevealed area of the matrix by a constant factor. This will be done in one of four ways: (a) by reducing the width of the cross ($i_2 - i_1$ and $j_2 - j_1$) by half, (b) by transforming the cross into an L shaped region which is a constant fraction smaller than the original cross, but whose center might not be contained in the original center, or (c) by transforming the cross into an L shaped region whose center (which might not be contained in the original center) has area at least one quarter of the total unrevealed area.

Let $i_m = \lfloor (i_1 + i_2)/2 \rfloor$ and $j_m = \lfloor (j_1 + j_2)/2 \rfloor$. We first probe (i_m, j_m) and its four neighbors. Either (i_m, j_m) is the local maximum, or by Lemma 5, one of its four quadrants can be eliminated. Assume that the eliminated quadrant is $(\geq i_m, \geq j_m)$, the other cases are handled symmetrically. Next, we apply the Lemma 6 on entries (i_1, j_1) and (i_m, j_m) . For this, we probe the entries (i_1, j_m) and (i_m, j_1) . Assume that $f(i_1, j_m) > f(i_m, j_1)$, the other case is handled symmetrically. Then the quadrant $(\geq i_m, \leq j_1)$ can be eliminated from the search. Furthermore, if the apex (i_m, j_1) is the minimum of that quadrant, then the quadrant satisfies the apex invariant. If it does not, then we still know that $f(i_m, j_1)$ is smaller than one of $f(i_1, j_1)$ and $f(i_m, j_m)$, otherwise rows i_1 and i_m and columns j_1 and j_m form a 2×2 matrix with 2 local optima. If $f(i_m, j_1) < f(i_1, j_1)$, then $f(i_m + 1, j_1) < f(i_m, j_1)$ but $f(i_m, j_1 - 1) > f(i_m, j_1)$ since the apex invariant is not satisfied. This implies $f(i_m, j_1 + 1) < f(i_m, j_1)$ and applying Lemma 5, quadrant $(\geq i_m, \geq j_1)$ can be eliminated. Those two eliminated quadrants together remove all rows $\geq i_m$. Likewise, if $f(i_m, j_1) < f(i_m, j_m)$, then the quadrant $(\leq i_m, \leq j_1)$ can be eliminated and so all columns $\leq j_1$ can be removed.

At this point, we have eliminated at least two quadrants, and the left, right and bottom legs have had their width divided by 2, and the bottom or left leg might have been eliminated. We now probe the four neighbors of (i_1, j_m) (whose value is known from the previous step), and apply Lemma 5 to eliminate one of its four quadrants. We now have four cases to consider.

If quadrant $(\leq i_1, \geq j_m)$ is eliminated, then we have achieved goal (a): we have a new cross of half the width (where one of the legs may have been eliminated), so the area of all four legs is multiplied by $\frac{1}{2}$, and the area of the center is multiplied by $\frac{3}{4}$.

If quadrant $(\leq i_1, \leq j_m)$ is eliminated, then we apply Lemma 6 to quadrants $(\leq i_1, \leq j_m - 1)$ and $(\geq i_m, \geq j_m)$. This eliminates either $(\leq i_1, \geq j_m - 1)$ in which case all rows $\leq i_1$ can be removed, or $(\geq i_m, \leq j_m)$, in which case all rows $\geq i_m$ can be removed. In both cases, the unrevealed region is a T shape, a cross with one leg cut, and all legs have had their width multiplied by a factor $\frac{1}{2}$, so we have again achieved goal (a).

If quadrant $(\geq i_1, \geq j_m)$ is eliminated, then all columns $\geq j_2$ are eliminated. We then apply Lemma 6 to quadrants $(\leq i_1, \leq j_1)$ and $(\geq i_1 + 1, \geq j_m)$. This further eliminates either $(\leq i_1, \geq j_m)$ or $(\geq i_1 + 1, \leq j_1)$. In the first case, all rows $\geq j_m$ can be removed and we obtain a T shaped unrevealed region, which is a cross with one leg cut off, and all legs have had their width multiplied by a factor

$\frac{1}{2}$, so we have again achieved goal (a). In the second case, the unrevealed region becomes L shaped: the left leg has been removed with quadrant $(\geq i_1 + 1, \leq j_1)$, the right leg was already removed, so what remains is the top leg, the center of the cross divided in two vertically and the bottom leg divided by two vertically. So unless the top leg contained more than half of the area unrevealed at the beginning of this step, we have eliminated at least one quarter of the total unrevealed area, and so we have achieved goal (b). Otherwise, if the top leg contained more than half of the area unrevealed at the beginning of this step, then the center of the new L shape contains more than half of the top leg, and so more than one quarter of the total unrevealed area, reaching goal (c).

Finally, if quadrant $(\geq i_1, \leq j_m)$ is eliminated, then all columns $\geq j_1$ are eliminated. We then apply Lemma 6 to quadrants $(\geq i_1 + 1, \leq j_m)$ and $(\leq i_1, \geq j_2)$. This further eliminates either $(\leq i_1, \leq j_m)$ or $(\geq i_1 + 1, \geq j_2)$. In the first case, all rows $\leq j_m$ can be removed and we obtain an L shaped unrevealed region, which is a cross with two legs cut off, and all legs have had their width multiplied by a factor $\frac{1}{2}$, so we have again achieved goal (a). In the second case, the unrevealed region becomes L shaped, containing just the top leg and a quarter of the original center. As in the previous case, if the top leg contained less than half of the area unrevealed at the beginning of this step, we have eliminated at least one quarter of the total unrevealed area, and so we have achieved goal (b). Otherwise, if the top leg contained more than half of the area unrevealed at the beginning of this step, then the center of the new L shape contains more than half of the top leg, and so more than one quarter of the total unrevealed area, and we have reached goal (c).

To conclude, note that every step performs a constant number of probes. After each elimination step, if goals (a) or (b) are reached, then one quarter of the unrevealed area has been eliminated. If goal (c) is attained, then the first set of probes of the next step eliminates one quarter of the area of the center of the cross, which is in this case at least one sixteenth of the total unrevealed area. So in all cases, two consecutive steps eliminate a constant fraction of the area, so the total number of steps is $O(\lg(mn))$. \square

7 Random Probing Algorithm

In this section we analyze a natural family of *uniform probing* strategies for finding a local optimum in an $\uparrow, \leftrightarrow$ unimodal function. We specify and analyze the strategy only in the case of totally unimodal functions, where of course the algorithm finds the global maximum. Our lower bound on the strategy's performance also applies to any generalization of this algorithm to $\uparrow, \leftrightarrow$ unimodal functions. Our upper bound is specific to totally unimodal functions.

The uniform probing algorithm for totally unimodal functions works as follows. Initially, we set the region R to the entire domain $D = \{1, 2, \dots, m\} \times \{1, 2, \dots, n\}$ of the function f . At each step, the algorithm chooses a point (i, j) uniformly at random from the remaining region R . Then the algorithm makes three samples to eliminate a quadrant except for its corner (i, j) , according to

Lemma 5. If the remaining region R' contains just one point, then it is the unique maximum; otherwise the algorithm continues.

Theorem 5. *Uniform probing makes $\Theta(\ln^2 \max)$ expected probes in a totally unimodal function.*

8 Conclusion

We expect many of our results to generalize to several other scenarios. In particular, we expect similar bounds in d dimensions, at least in the case of an $n \times n \times \cdots \times n$ matrix, where logarithmic bounds remain logarithmic and squared-logarithmic bounds grow to \log^d . We also believe that our results generalize to local maxima defined in terms of size-8 neighborhoods instead of the 4-neighborhoods we use. For example, as with 4-neighborhoods, total unimodality with 8-neighborhoods can be characterized as forbidding a finite set of partial orders on constant-size submatrices. More generally, it would be interesting to characterize the complexities achievable for all possible forbidden submatrices. In our work, we have seen how combining various unimodality conditions with the total monotonicity conditions of [1] yields surprising results. In particular, combining $\uparrow, \leftrightarrow$ unimodality with total $(+i, -j)$ or $(-i, +j)$ monotonicity implies total unimodality (Lemma 1), and so an $O(\log n)$ optimization algorithm, while combining $\uparrow, \leftrightarrow$ unimodality with total $(+i, +j)$ or $(-i, -j)$ monotonicity has an $\Omega(\log^2 \min)$ lower bound (Theorem 3).

References

1. A. Aggarwal, M. M. Klawe, S. Moran, P. W. Shor, and R. Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2:195–208, 1987.
2. I. Althöfer and K.-U. Koschnick. On the deterministic complexity of searching local maxima. *Discrete Appl. Math.*, 43(2):111–113, 1993.
3. H. Edelsbrunner. Computing the extreme distances between two convex polygons. *J. Algorithms*, 6:213–224, 1985.
4. Z. Galil and K. Park. Dynamic programming with convexity, concavity and sparsity. *Theoret. Computer Sci.*, 92(1):49–76, Jan. 1992.
5. B. Gärtner, W. D. Morris, and L. Rüst. Unique sink orientations of grids. In *Proc. 11th Internat. IPCO Conf. on Integer Prog. and Combinat. Opt.*, volume 3509 of *Lecture Notes in Computer Science*, pages 210–224, Berlin, Germany, June 2005.
6. J. Kiefer. Sequential minimax search for a maximum. *Proc. Amer. Math. Soc.*, 4:502–506, 1953.
7. D. C. Llewellyn, C. Tovey, and M. Trick. Local optimization on graphs. *Discrete Appl. Math.*, 23(2):157–178, 1989.
8. D. C. Llewellyn and C. A. Tovey. Dividing and conquering the square. *Discrete Appl. Math.*, 43(2):131–153, 1993.
9. A. Mityagin. On the complexity of finding a local maximum of functions on discrete planar subsets. *Theoret. Computer Sci.*, 310(1–3):355–363, Jan. 2004.
10. C. A. Tovey. Local improvement on discrete structures. In *Local Search in Combinatorial Optimization*, pages 57–89. John Wiley and Sons, 1997.

Author Index

- Agarwal, Pankaj K. 355
Aissi, Hassene 862
Alfieri, Arianna 283
Arge, Lars 355
Ausiello, Giorgio 532
Azar, Yossi 484
- Bar-Yehuda, Reuven 714
Batra, Garima 35
Bazgan, Cristina 862
Bejerano, Yigal 702
Benkert, Marc 143
Benkoczi, Robert 271
Berberich, Eric 155
Berenbrink, Petra 746
Berger, André 472
Bergeron, Anne 779
Bhattacharya, Binay 271
Bienkowski, Marcin 815
Bilò, Vittorio 460
Björklund, Andreas 839
Bodlaender, Hans L. 95, 391
Boissonnat, Jean-Daniel 367
Buchbinder, Niv 689
Byrka, Jaroslaw 815
- Cabello, Sergio 131, 520
Caragiannis, Ioannis 460
Chaudhry, Geeta 317
Chauve, Cedric 779
Chen, Zhi-Zhong 179
Christodoulou, George 59
Cicalese, Ferdinando 664
Codenotti, Bruno 83
Cormen, Thomas H. 317
Crochemore, Maxime 426
Czumaj, Artur 472
- Daskalakis, Konstantinos 71
de Berg, Mark 508
de Kok, Thierry 343
de Montgolfier, Fabien 779
Delage, Christophe 367
Demaine, Erik D. 887
Dementiev, Roman 640
- Díaz, J. 215
Dorn, Frederic 95
Du, Zilin 874
- Eigenwillig, Arno 155
Epstein, Leah 604
Ergun, Funda 746
- Farach-Colton, Martín 827
Farshi, Mohammad 556
Farzan, Arash 305
Faye, Alain 850
Fernandes, Rohan J. 827
Ferragina, Paolo 305
Fiat, Amos 803
Finocchi, Irene 1
Fischer, Johannes 415
Fishkin, Aleksei V. 580
Fleischer, Rudolf 11
Fomin, Fedor V. 95
Frahling, Gereon 758
Fraigniaud, Pierre 791
Franceschini, Gianni 305
Franciosa, Paolo G. 532
Frank, András 249
Friedetzky, Tom 746
- Garg, Naveen 35
Giannopoulos, Panos 520
Gidenstam, Anders 329
Ginzinger, Simon W. 415
Grammatikopoulos, G. 215
Grandoni, Fabrizio 1
Grigni, Michelangelo 472
Grigoriev, Alexander 391
Gudmundsson, Joachim 556
Gupta, Garima 35
- Hassin, Refael 167, 726
Haverkort, Herman 508
Hay, David 496
Hayrapetyan, Ara 191
Heggernes, Pinar 403
Hemmer, Michael 155

- Hermelin, Danny 426
Hert, Susan 155
Holzer, Martin 628
Hu, T.C. 226
- Italiano, Giuseppe F. 1, 532
Ito, Hiro 119
Iwama, Kazuo 119
- Jansen, Klaus 580
- Kaklamanis, Christos 460
Kanellopoulos, Panagiotis 460
Kaporis, A.C. 215
Karpinski, Marek 238
Kempe, David 191
Kettner, Lutz 155, 640
Khuller, Samir 259
Király, Zoltán 249
Kirosis, L.M. 215
Kliwer, Georg 47
Knauer, Christian 520
Koster, Arie M.C.A. 391
Kotnyek, Balázs 249
Koutsoupas, Elias 59
Kovács, Annamária 616
Krokowski, Jens 758
Krommidas, Ioannis 544
Kučera, Luděk 203
- Laber, Eduardo Sany 664
Landau, Gad M. 426
Langerman, Stefan 887
Larmore, Lawrence L. 226
Lauther, Ulrich 293
Lee, Kwangil 259
Levin, Asaf 726
Löfller, Maarten 343
Lukovszki, Tamás 293
- Manku, Gurmeet Singh 438
Marx, Dániel 448
McCune, Benton 83
Mehlhorn, Kurt 155
Mitchell, Joseph S.B. 143
Mohar, Bojan 131
Moore, Cristopher 10
Morgenthaler, J. David 226
Mosteiro, Miguel A. 827
Munagala, Kamesh 677
- Munro, J. Ian 305
Muthukrishnan, S. 734
- Nagoya, Takayuki 179
Naor, Joseph 9, 689, 702
Nekrich, Yakov 238
- Or, Einat 167
Osumi, Tsuyoshi 119
- Pál, Martin 191
Papadimitriou, Christos H. 71
Papatriantafyllou, Marina 329
Paul, Christophe 379
Penninx, Eelko 95
Pérez, X. 215
Prasinos, Grigorios 628
- Raffinot, Mathieu 779
Raman, Rajiv 83
Rawitz, Dror 714
Reichel, Joachim 155
Reinbacher, Iris 143
Rote, Günter 520
Roupin, Frédéric 850
Rührup, Stefan 23
- Saia, Jared 803
Sanders, Peter 568, 640
Sankowski, Piotr 770
Sawada, Joe 438
Scalosub, Gabriel 496
Schindelbauer, Christian 23
Schmitt, Susanne 155
Schömer, Elmar 155
Schultes, Dominik 568
Schulz, Frank 628
Sevastyanov, Sergey V. 580
Sgall, Jiří 592
Shachnai, Hadas 592
Sharma, Vikram 874
Shayman, Mark 259
Sitters, René 580
Sotiropoulos, D.G. 215
Sprintson, Alexander 702
Strauss, M. 734
Streppel, Micha 508
Suchan, Karol 403
Svitkina, Zoya 191
- Tam, Yiu-Cheong 652
Tamir, Tami 592

- Telle, Jan Arne 379
Timajev, Larissa 47
Todinca, Ioan 403
Trippen, Gerhard 11
Tsigas, Philippas 329
- van de Velde, Steef L. 283
Vanderpooten, Daniel 862
van Kreveld, Marc 143, 343
Varadarajan, Kasturi 83
Vialette, Stéphane 426
Villanger, Yngve 403
- Wagner, Dorothea 628
Wahlström, Magnus 107
- Woeginger, Gerhard J. 283
Wolff, Alexander 143
Wolpert, Nicola 155
Wong, Chi-Him 652
- Yang, Jun 677
Yap, Chee K. 874
Yi, Ke 355
Young, Maxwell 803
Yu, Hai 677
- Zachut, Rafi 484
Zaroliagis, Christos 544, 628
Zhao, Hairong 472
Zheng, X. 734