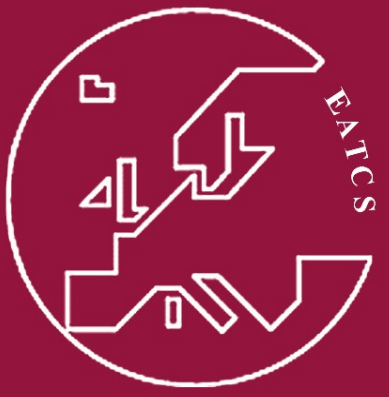
Yossi Azar
Thomas Erlebach (Eds.)

LNCS 4168

Algorithms – ESA 2006

14th Annual European Symposium
Zurich, Switzerland, September 2006
Proceedings



Springer

Lecture Notes in Computer Science 4168

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Yossi Azar Thomas Erlebach (Eds.)

Algorithms – ESA 2006

14th Annual European Symposium
Zurich, Switzerland, September 11 – 13, 2006
Proceedings

Springer

Volume Editors

Yossi Azar
Tel-Aviv University
Department of Computer Science
69978 Tel Aviv, Israel
E-mail: azar@tau.ac.il

Thomas Erlebach
University of Leicester
Department of Computer Science
University Road, Leicester LE1 7RH, UK
E-mail: t.erlebach@mcs.le.ac.uk

Library of Congress Control Number: 2006931490

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

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN 0302-9743
ISBN-10 3-540-38875-3 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-38875-3 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11841036 06/3142 5 4 3 2 1 0

Preface

This volume contains the 70 contributed papers and abstracts of 3 of the 5 invited talks presented at the 14th Annual Symposium on Algorithms (ESA 2006), held at ETH Zurich in Zurich, Switzerland, September 11–13, 2006. The papers in each section of the proceedings are arranged alphabetically. The five distinguished invited speakers were Erik Demaine, Lisa Fleischer, László Lovász, Kurt Mehlhorn, and Ron Shamir.

Since 2002, ESA has consisted of two tracks, with separate Program Committees, which deal 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); Bergen, Norway (2004); and Palma de Mallorca, Spain (2005). The proceedings of these symposia were published as Springer's LNCS volumes 2461, 2832, 3221 and 3669, 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 finance and algorithmic game theory, computational geometry, data structures, databases and information retrieval, external-memory algorithms, graph and network algorithms, graph drawing, machine learning, mobile and distributed 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 network optimization.

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 Committees of both tracks met at ETH Zurich on May 27–28, 2006. The Design and Analysis track selected 52 out of 215 submissions. The Engineering and Applications track selected 18 out of 72 submissions.

ESA 2006 was sponsored by the EATCS (European Association for Theoretical Computer Science). The EATCS sponsorship included an award of euro 500 for the authors of the best student paper. The award was shared by Frederic Dorn for his paper "Dynamic Programming and Fast Matrix Multiplication" and Michal Meyerovitch for her paper "Robust, Generic and Efficient Construction of Envelopes of Surfaces in Three-Dimensional Space."

The Program Committees of the two tracks of ESA 2006 consisted of:

Design and Analysis Track

Pankaj Agarwal	Duke University
Lars Arge	University of Aarhus
Yossi Azar (Chair)	Tel-Aviv University
Nikhil Bansal	IBM T.J. Watson Research Center
Allan Borodin	University of Toronto
Martin Dyer	University of Leeds
Dimitris Fotakis	University of the Aegean
Magnus M. Halldorsson	University of Iceland
Monika Henzinger	Google and ETH Lausanne
Tibor Jordan	Eotvos University, Budapest
Jan Karel Lenstra	CWI, Amsterdam
Yishay Mansour	Tel-Aviv University
Friedhelm Meyer auf der Heide	University of Paderborn
Alessandro Panconesi	La Sapienza University, Rome
Rob van Stee	Karlsruhe University
Mariette Yvinec	INRIA Sophia Antipolis

Engineering and Applications Track

Edoardo Amaldi	Politecnico di Milano
Leah Epstein	University of Haifa
Thomas Erlebach (Chair)	University of Leicester
Lene Favrholdt	University of Southern Denmark
Alexander Hall	ETH Zurich
Dan Halperin	Tel-Aviv University
Ulrich Meyer	MPI-INF Saarbrücken
Rolf Niedermeier	University of Jena
Cliff Stein	Columbia University
Roberto Tamassia	Brown University
Suresh Venkatasubramanian	AT&T

ESA 2006 was held along with the 6th Workshop on Algorithms in Bioinformatics (WABI), the 4th Workshop on Approximation and Online Algorithms (WAOA), the Second International Workshop on Parameterized and Exact Computation (IWPEC), and the 6th Workshop on Algorithmic meThods and Models for Optimization of railwayS (ATMOS) in the context of the combined conference ALGO 2006. The Organizing Committee of ALGO 2006 consisted of, all from ETH Zurich:

Franziska Hefti
Michael Hoffmann (Chair)
Angelika Steger
Emo Welzl
Peter Widmayer

We would like to thank ETH Zurich, in particular Michael Hoffmann and Emo Welzl, for the hospitality at the Program Committee meeting.

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

June 2006 Yossi Azar and Thomas Erlebach

Organization

Referees

Karen Aardal
David Abraham
Dimitris Achlioptas
Divesh Aggarwal
Geir Agnarsson
Kunal Agrawal
Deepak Ajwani
Tatsuya Akutsu
Susanne Albers
Noga Alon
Ernst Althaus
Nir Andelman
Giovanni Andreatta
Spyros Angelopoulos
Jay Aslam
Hagit Attiya
Franz Aurenhammer
Brian Babcock
Mihály Bárász
Gill Barequet
Cindy Barnhardt
Amotz Bar-Noy
Yair Bartal
Ziv Bar-Yossef
Luca Becchetti
Johanna Becker
Rene Beier
Pietro Belotti
Andras Benczur
Boaz Ben-Moshe
Gary Benson
Eric Berberich
Sergey Bereg
Mark de Berg
Piotr Berman
Attila Bernáth
Marcin Bienkowski
Dan Bienstock
Davide Bilo
Johannes Blömer
Jean-Daniel Boissonnat
Vincenzo Bonifaci
Magnus Bordewich
Endre Boros
Ulrik Brandes
Vasco Brattka
Mark Braverman
Gerth S. Brodal
Andrej Brodnik
Hervé Brönnimann
Michael Brudno
Adam Buchsbaum
Alberto Caprara
Ioannis Caragiannis
Jean Cardinal
Alberto Ceselli
Raphaelle Chaine
Amit Chakrabarti
Timothy Chan
Moses Charikar
Ke Chen
Xiaomin Chen
Otfried Cheong
Joseph Cheriyan
Marco Chiarandini
Bogdan Chlebus
George Christodoulou
Marek Chrobak
Fabian Chudak
Serafino Cicerone
Richard Cole
Roberto Cordone
José Correa
Artur Czumaj
Andrew Danner
Mayur Datar
Christophe Delage
Federico Della Croce
Mauro Dell'Amico
Gianluca Della Vedova
Roman Dementiev
Olivier Devillers
Michael Dom
Zhao Dong
Devdatt Dubhashi
Paul Duetting
Adrian Dumitrescu
Keith Edwards
Pavlos Efraimidis
Friedrich Eisenbrand
Michael Elkin
Ran El Yaniv
Amir Epstein
Boris Epstein
Leah Epstein
Thomas Erlebach
Guy Even
Eyal Even-Dar
Rolf Fagerberg
Tomas Feder
Uri Feige
Sandor Fekete
Zsolt Fekete
Paolo Ferragina
Faith Fich
Irene Finocchi
Balázs Fleiner
Tamás Fleiner
Fedor Fomin
Pierre Fraigniaud
Tobias Friedrich
Alan Frieze
Toshihiro Fujito
Hal Gabow
Nicola Galesi
Philippe Galinier
Rajiv Gandhi

Pierre-Marie Gandoin
Bill Gasarch
Leszek Gasieniec
Joachim Gehweiler
Loukas Georgiadis
Bert Gerards
Andrew Goldberg
Leslie Goldberg
Fabrizio Grandoni
Martin Grohe
Andrea Grosso
Prabhakar Gubbala
Joachim Gudmundsson
Sudipto Guha
Jiong Guo
Gregory Gutin
Walter Gutjahr
Shai Gutner
Carsten Gutwenger
Torben Hagerup
Bjarni V. Halldorsson
Horst Hamacher
Sariel Har-Peled
Reza Hashemian
Refael Hassin
Reinhold Heckmann
Pinar Heggernes
Edith Hemaspaandra
Martin Hoefer
Thomas Hofmeister
Han Hoogeveen
Takashi Horiyama
Falk Hüffner
Cor Hurkens
Nicole Immorlica
Piotr Indyk
Robert Irving
Kazuo Iwama
Ravi Janardan
Klaus Jansen
David Johnson
Alpár Jüttner
Volker Kaibel
Kanela Kaligosi
Michael Kaminski
Panagiotis Kanellopoulos
Alexis Kaporis
Menelaos Karavelas
Howard Karloff
Ragnar Karlsson
Irit Katriel
Dimitris Kavvadias
Steven Kelk
Tracy Kimbrel
Tamás Király
Zoltán Király
Stephen Kobourov
Jochen Könemann
Jens Svalgaard Kohrt
Stavros Kolliopoulos
Elisavet Konstantinou
Jan Korst
Guy Kortsarz
Miroslaw Korzeniowski
Arie Koster
Darek Kowalski
Shankar Krishnan
Michael Krivelevich
Nico Kruithoff
Piotr Krysta
Fabian Kuhn
John Langford
Larry Larmore
Hyun Chul Lee
Stefano Leonardi
Hanoch Levi
Retsef Levi
Asaf Levin
Moshe Lewenstein
Leo Liberti
Marie-Colette van Lieshout
Andrea Lodi
Manuel López-Ibáñez
Zvi Lotker
Tamas Lukovski
Francesco Maffioli
Thomas Mailund
Márton Makai
Konstantin Makarychev
Kazuhisa Makino
Vittorio Maniezzo
Fredrik Manne
Carlo Mannino
Shie Mannor
Conrado Martinez
Yossi Matias
Alexander May
Frank McSherry
Abdelkrim Mebarki
Paul Medvedev
Nicole Megow
Kurt Mehlhorn
Alessandro Mei
Vahab Mirrokni
Neeraj Mittal
Michael Molloy
Michele Monaci
Ruggero Morselli
Gabriel Moruz
Hannes Moser
Haiko Müller
Ian Munro
Nabil Mustafa
S. Muthukrishnan
Hiroshi Nagamochi
Gonzalo Navarro
Frank Neumann
Phong Nguyen
Trung Nguyen
Van Nguyen
Rolf Niedermeier
Sortiris Nikoletseas
Zeev Nutov
Anna Östlin Pagh
Rasmus Pagh
Gyula Pap
Júlia Pap
Vicky Papadopoulou
Srinivasan Parthasarathy
Francesco Pasquale
Boaz Patt-Shamir
Christian N. S. Pedersen
Andrzej Pelc
David Peleg

Marco Pellegrini
Michal Penn
Pino Persiano
Seth Pettie
Gabriel Peyré
Ulrich Pferschy
Tomas Philip Runarsson
Jeff Philips
David Phillips
Andrea Pietracaprina
Mustafa C. Pinar
Sylvain Pion
David Pisinger
Yves Pochet
Magda Procopiuc
Guido Proietti
Kirk Pruhs
Tomasz Radzik
R. Ravi
Dror Rawitz
Oded Regev
Gerhard Reinelt
Klaus Reinhardt
Yossi Richter
Giovanni Righini
Laurent Rineau
Romeo Rizzi
Liam Roditty
Dana Ron
Amir Ronen
Stefan Røpke
Adi Rosen
Christian Rössl
Matthias Ruhl
Eytan Ruppin
Daniel Russel
Amin Saberi
Kunihiko Sadakane
Sudheer Sahu
Marie Samozino
Ben Sandbank
Peter Sanders
Christian Scheideler
Heiko Schilling
Christian Schindelhauer
Stefan Schirra
Markus Schmidt
Lex Schrijver
Dominik Schultes
Rüdiger Schultz
Oded Schwartz
Danny Segev
Hadas Shachnai
Nira Shafrir
Micha Sharir
Susan Shortreed
Anastasios Sidiropoulos
Hans Ulrich Simon
Rene Sitters
Michiel Smid
Sagi Snir
Nir Sochen
Troels Bjerre Sørensen
Christian Sohler
Motti Sorani
Gregory Sorkin
Frits Spieksma
Dan Spielman
Andreas Spillner
Yannis Stamatiou
Stamatis Stefanakos
Kostas Stergiou
Leen Stougie
Jonathan Z. Sun
Maxim Sviridenko
Zoltán Szabadka
Jácint Szabó
Tami Tamir
Éva Tardos
Monique Teillaud
Miklos Telek
Kavitha Telikepalli
Moshe Tennenholtz
Dimitrios Thilikos
Robin Thomas
Takeshi Tokuyama
Sivan Toledo
Laura Toma
Eric Torng
Luca Trevisan
Marc Uetz
Takeaki Uno
Patch Uthaisombut
Ugo Vaccaro
Jan Vahrenhold
Kasturi Varadarajan
László Végh
Santosh Vempala
Suresh Venkatasubramanian
Carmine Ventre
Elad Verbin
Berthold Vöcking
Tjark Vredeveld
Dorothea Wagner
Yusu Wang
Osamu Watanabe
Ron Wein
Emo Welzl
Carola Wenk
Renato Werneck
Sebastian Wernicke
Matthias Westermann
Ryan Willams
Gerhard J. Woeginger
Philipp Woelfel
Roberto Wolfler Calvo
Nicola Wolpert
Camille Wormser
Atsuko Yamaguchi
Hande Yaman
Jun Yang
Ke Yi
Hai Yu
Raphael Yuster
Martin Zachariasen
Christos Zaroliagis
Norbert Zeh
Guochuan Zhang
Lisa Zhang
Michele Zito
Uri Zwick

Table of Contents

Invited Lectures

Contributed Papers: Design and Analysis Track

Contributed Papers: Engineering and Applications Track

Origami, Linkages, and Polyhedra: Folding with Algorithms

Erik D. Demaine

MIT Computer Science and Artificial Intelligence Laboratory,
32 Vassar St., Cambridge, MA 02139, USA
edemaine@mit.edu

Abstract. What forms of origami can be designed automatically by algorithms? What shapes can result by folding a piece of paper flat and making one complete straight cut? What polyhedra can be cut along their surface and unfolded into a flat piece of paper without overlap? When can a linkage of rigid bars be untangled or folded into a desired configuration? Folding and unfolding is a branch of discrete and computational geometry that addresses these and many other intriguing questions. I will give a taste of the many results that have been proved in the past few years, as well as the several exciting open problems that remain open. Many folding problems have applications in areas including manufacturing, robotics, graphics, and protein folding.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, p. 1, 2006.

Reliable and Efficient Geometric Computing

Kurt Mehlhorn

Max-Planck-Institut für Informatik

Computing with geometric objects (points, curves, and surfaces) is central for many engineering disciplines and lies at the heart of computer aided design systems. Implementing geometric algorithms is notoriously difficult and most actual implementations are incomplete: they are known to crash or deliver the wrong result on some instances.

In the introductory part of the talk, we illustrate the *pitfalls of geometric computing* [5] and explain for one algorithm in detail where the problem lies and what goes wrong.

In the main part of the talk I discuss approaches to reliable and efficient geometric computing, in particular, the *exact computation paradigm* [3, 9, 7] and *controlled perturbation* [4, 8]. In both cases, I report about recent theoretical advances and the use of the paradigms in systems LEDA [6], CGAL [1], and EXACUS [2]. The research combines techniques from computational geometry, solid modeling, computer algebra, and numerical analysis.

References

1. CGAL (Computational Geometry Algorithms Library). www.cgal.org.
2. EXACUS (EXAct computation with CUrves and Surfaces), 2003. www.mpi-sb.mpg.de/projects/EXACUS.
3. S. Fortune and C. van Wyk. Efficient exact integer arithmetic for computational geometry. In *7th ACM Conference on Computational Geometry*, pages 163–172, 1993.
4. D. Halperin and C. Shelton. A perturbation scheme for spherical arrangements with application to molecular modeling. *CGTA: Computational Geometry: Theory and Applications*, 10, 1998.
5. L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C. Yap. Classroom examples of robustness problems in geometric computations. In *ESA*, volume 3221 of *LNCS*, pages 702–713, 2004. www.mpi-sb.mpg.de/~mehlhorn/ftp/ClassRoomExample.ps.
6. LEDA (Library of Efficient Data Types and Algorithms). www.mpi-sb.mpg.de/LEDA/leda.html.
7. K. Mehlhorn and S. Näher. *The LEDA Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999. 1018 pages.
8. K. Mehlhorn, R. Osbild, and M. Sagraloff. Reliable and Efficient Computational Geometry via Controlled Perturbation To appear in ICALP 06.
9. C. Yap. Towards exact geometric computation. *CGTA: Computational Geometry: Theory and Applications*, 7, 1997.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, p. 2, 2006.

Some Computational Challenges in Today's Bio-medicine

Ron Shamir

School of Computer Science,
Tel Aviv University, Tel Aviv, 69978, Israel
rshamir@tau.ac.il

Abstract. Over the last decade, biology has been rapidly transformed into an information science. Novel high-throughput technologies provide information in a scope that was unimaginable not long ago, and with these data a plethora of computational riddles are emerging. The challenge of deep and integrated computational analysis of diverse biological data, providing meaningful understanding of life processes and principles, is still very far from being answered. If met, this could help improve the quality of life of this and future generations.

We shall discuss several such challenges arising in diverse areas of biology and medicine, including analysis and evolution of genetic regulatory networks, disease association, and genome rearrangements. The tension between elegant algorithmics and useful practical solutions will be a common theme in this story.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, p. 3, 2006.

Kinetic Collision Detection for Convex Fat Objects*

M.A. Abam, M. de Berg, S.-H. Poon, and B. Speckmann

Department of Mathematics and Computing Science,
TU Eindhoven, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{mabam, mdberg, spoon, speckman}@win.tue.nl

Abstract. We design compact and responsive kinetic data structures for detecting collisions between n convex fat objects in 3-dimensional space that can have arbitrary sizes. Our main results are:

(*i*) If the objects are 3-dimensional balls that roll on a plane, then we can detect collisions with a KDS of size $O(n \log n)$ that can handle events in $O(\log n)$ time. This structure processes $O(n^2)$ events in the worst case, assuming that the objects follow constant-degree algebraic trajectories.

(*ii*) If the objects are convex fat 3-dimensional objects of constant complexity that are free-flying in $\mathbb{R}^3$, then we can detect collisions with a KDS of $O(n \log^6 n)$ size that can handle events in $O(\log^6 n)$ time. This structure processes $O(n^2)$ events in the worst case, assuming that the objects follow constant-degree algebraic trajectories. If the objects have similar sizes then the size of the KDS becomes $O(n)$ and events can be handled in $O(1)$ time.

1 Introduction

Collision detection is a basic computational problem arising in all areas of computer science involving objects in motion—motion planning, animated figure articulation, computer-simulated environments, or virtual prototyping, to name a few. Very often the problem of detecting collisions is broken down into two phases: a *broad phase* and a *narrow phase*. The broad phase determines pairs of objects that might possibly collide, frequently using (hierarchies of) bounding volumes to speed up the process. The narrow phase then uses specialized techniques to test each candidate pair, often by tracking closest features of the objects in question, a process that can be sped up significantly by exploiting spatial and temporal coherence. See [19] for a detailed overview of algorithms for such collision and proximity queries.

Algorithms that deal with objects in motion traditionally discretize the time axis and compute or update their structures based on the position of the objects

* M.A. and S.-H.P. were supported by the Netherlands' Organisation for Scientific Research (NWO) under project no. 612.065.307. M.d.B. was supported by the Netherlands' Organisation for Scientific Research (NWO) under project no. 639.023.301.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 4–15, 2006.

at every time step. But since collisions tend to occur rather irregularly it is nearly impossible to choose the perfect time-step: too large an interval between sampled times will result in missed collisions, too small an interval will result in unnecessary computations (and still there is no guarantee that no collisions are missed). Event-driven methods, on the other hand, compute the event times of significant changes to a system of moving objects, store those in a priority queue sorted by time, and advance the system to the event at the front of the queue. The kinetic-data-structure framework initially introduced by Basch et al. [4] presents a systematic way to design and analyze event-driven data structures for moving objects.

A kinetic data structure (KDS) is designed to maintain or monitor a discrete attribute of a set of moving objects, where each object has a known motion trajectory or *flight plan*. A KDS contains a set of *certificates* that constitutes a proof of the property of interest. These certificates are inserted in a priority queue (*event queue*) based on their time of expiration. The KDS then performs an event-driven simulation of the motion of the objects, updating the structure whenever a certificate fails. A KDS for collision detection finds a set of geometric tests (elementary certificates) that together provide a proof that the input objects are disjoint—see the survey by Guibas [11, 12] for more details.

Kinetic data structures and their accompanying maintenance algorithms can be evaluated and compared with respect to four desired characteristics. A good KDS is *compact* if it uses little space in addition to the input, *responsive* if the data structure invariants can be restored quickly after the failure of a certificate, *local* if it can be updated easily when the flight plan for an object changes, and *efficient* if the worst-case number of events handled by the data structure for a given motion is small compared to some worst-case number of "external events" that must be handled for that motion.

Kinetic data structures for collision detection. One of the first papers on kinetic collision detection was published by Basch et al. [3], who designed a KDS for collision detection between two simple polygons in the plane. Their work was extended to an arbitrary number of polygons by Agarwal et al. [1]. Kirkpatrick et al. [17] and Kirkpatrick and Speckmann [18] also described KDS's for kinetic collision detection between multiple polygons in the plane. These solutions all maintain a decomposition of the free space between the polygons into "easy" pieces (usually pseudo-triangles). Unfortunately it seems quite hard to define a suitable decomposition of the free space for objects in 3D, let alone maintain it while the objects move—the main problem being, that all standard decomposition schemes in 3D can have quadratic complexity. Hence, even though collision detection is the obvious application for kinetic data structures, there has hardly been any work on kinetic collision detection in 3D.

There are only a few papers that deal directly with (specialized versions of) kinetic 3D collision detection. Guibas et al. [13], extending work by Erickson et al. [10] in the plane, show how to certify the separation of two convex polyhedra moving rigidly in 3D using certain outer hierarchies. Basch et al. [5] describe a structure for collision detection among multiple convex *fat* objects that have

almost the same size. The structure of Basch et al. uses $O(n \log^2 n)$ storage and events can be processed in $O(\log^3 n)$ time. Coming and Staadt [9] kineticize the sweep-and-prune approach to find candidate pairs of objects that might collide. Their method has a quadratic worst-case bound and they give only experimental evidence for its performance. If all objects are spheres of similar sizes Kim et al. [15] present an event-driven approach that subdivides space into cells and processes events whenever a sphere enters or leaves a cell. This approach was later extended [16] to accommodate spheres with unknown trajectories but still similar sizes. There is only experimental evidence for the performance of this method. Finally, Guibas et al. [13] use the power diagram of a set of arbitrary balls in 3D to kinetically maintain the closest pair among them. The worst-case complexity of this structure is quadratic and it might undergo more than cubically many changes.

Results. The main goal of our paper is to develop KDS's for 3D collision detection that have a *near-linear number of certificates* for *multiple* convex fat objects of *varying sizes*. As discussed above, none of the existing solutions achieves all these goals simultaneously. Our KDS's can be viewed as structures that perform the broad phase of the global collision-detection approach sketched above; one still has to detect collisions between the candidate pairs of objects produced by the KDS. Assuming the objects have constant complexity, this can trivially be done in constant time per pair; how to do this for complex objects is beyond the scope of this paper. Thus the challenge is to get a near-linear number of certificates, so that the number of candidate pairs is reduced from quadratic to near-linear.

We start in Section 2 with the special case of n balls of arbitrary sizes rolling on a plane. Here we present an elegant and simple KDS that uses $O(n \log n)$ storage and processes $O(n^2)$ events in the worst case if the objects follow constant-degree algebraic[1] trajectories. Processing an event takes $O(\log n)$ time.

In Section 3 we turn our attention to free-flying convex fat objects. Note that we do not assume the objects to be polyhedral. We first study fat objects that have similar sizes. We give an almost trivial KDS that has $O(n)$ size and processes $O(n^2)$ events; handling an event takes $O(1)$ time. This improves both the storage and the event-handling time of the KDS of Basch et al. [5] by several logarithmic factors. Next we consider the much more difficult general case, where the fat objects can have vastly different sizes. Here we present a KDS that uses $O(n \log^6 n)$ storage and processes $O(n^2)$ events; handling an event takes $O(\log^6 n)$ time. This is the first collision-detection KDS for multiple objects in 3D that has a near-linear number of certificates and does not require the

[1] In fact, the bound on the number of events holds in a more general setting: We maintain lists of certain x- and y-coordinates—for instance the coordinates of the tangency points of the disks with the plane on which they roll—whose values change according to the motions of the objects. The number of events is bounded by the number of changes (swaps) in these sorted lists. The $O(n^2)$ bound thus holds if we assume that any pair of coordinates swaps $O(1)$ times (which is for example the case if the motions are constant-degree algebraic). A similar remark holds for the other KDS's that we develop.

objects to have similar sizes. Even though our KDS for this case uses $O(n \log^6 n)$ storage, it maintains only a linear number of candidate pairs of objects to test for collisions; the additional storage is used in various supporting data structures. Our structure is based on the following idea: we put a number of points—we call them guards—around each object in such a way that if two objects collide, one must contain a guard from the other. Because the objects are fat, we can show that a constant number of guards per object suffices. The idea of reducing problems on fat objects to problems on suitably chosen points has been used before. In our context, however, it is far from straightforward to apply since detecting collisions between objects and guards is nearly as difficult as detecting collisions between the objects themselves. Nevertheless, using several additional ideas, we show how to make this approach work.

In the reminder of the paper we present the main idea behind our results. For lack of space, most of the proofs are omitted.

2 Balls Rolling on a Plane

Assume that we are given a set $\mathcal{B}$ of n 3-dimensional balls which are rolling on a 2-dimensional plane T, that is, the balls in $\mathcal{B}$ move continuously while remaining tangent to T. In this section we describe a responsive and compact KDS that detects collisions between the balls in $\mathcal{B}$.

The basic idea behind our KDS is to construct a *collision tree* recursively as follows:

- If $|\mathcal{B}| = 1$, then there are obviously no collisions and the collision tree is just a single leaf.
- If $|\mathcal{B}| > 1$, then we partition $\mathcal{B}$ into two subsets, $\mathcal{B}_S$ and $\mathcal{B}_L$. The subset $\mathcal{B}_S$ contains the $\lfloor n/2 \rfloor$ smallest balls and the subset $\mathcal{B}_L$ contains the $\lceil n/2 \rceil$ largest balls from $\mathcal{B}$, where ties are broken arbitrarily. The collision tree now consists of a root node that has an associated structure to detect collisions between any ball from $\mathcal{B}_S$ and any ball from $\mathcal{B}_L$, and two subtrees that are collision trees for the sets $\mathcal{B}_S$ and $\mathcal{B}_L$, respectively.

To detect all collisions between the balls in $\mathcal{B}$ it suffices to detect collisions between the two subsets maintained at every node of the collision tree. Let $\mathcal{B}_S$ and $\mathcal{B}_L$ denote the two subsets maintained at a particular node. The remainder of this section focusses on detecting collisions between the balls in $\mathcal{B}_S$ and those in $\mathcal{B}_L$. In particular, we describe a KDS of size $O(|\mathcal{B}_S| + |\mathcal{B}_L|)$ that can handle events in $O(1)$ time—see Lemma 4. The structure processes $O((|\mathcal{B}_S| + |\mathcal{B}_L|)^2)$ events in the worst case, assuming that the balls follow constant-degree algebraic trajectories. Since the same event can occur simultaneously at $O(\log n)$ nodes of the collision tree, we obtain the following theorem:

Theorem 1. *For any set $\mathcal{B}$ of n 3-dimensional balls that roll on a plane, there is a KDS for collision detection that uses $O(n \log n)$ space and processes $O(n^2)$ events in the worst case, assuming that the balls follow constant-degree algebraic trajectories. Each event can be handled in $O(\log n)$ time.*

2.1 Detecting Collisions Between Small and Large Balls

As mentioned above, we can restrict ourselves to detecting collisions between balls from two disjoint sets $\mathcal{B}_S$ and $\mathcal{B}_L$ where the balls in $\mathcal{B}_L$ are at least as large as the balls in $\mathcal{B}_S$. Recall that all balls are rolling on a plane T. Our basic strategy is the following: we associate a region D_i on T with each $B_i \in \mathcal{B}_L$ such that if the point of tangency of a ball $B_j \in \mathcal{B}_S$ and T is not contained in D_i, then B_j cannot collide with B_i. The regions associated with the balls in $\mathcal{B}_L$ need to have two important properties: (i) each point in T is contained in a constant number of regions and (ii) we can efficiently detect whenever a region starts or stops to contain a tangency point when the balls in $\mathcal{B}_L$ and $\mathcal{B}_S$ move. We first deal with the first requirement, that is, we consider $\mathcal{B}_L$ to be static. For a ball B_i let r_i denote its radius and let t_i be the point of tangency of B_i and T.

The threshold disk. We define the distance of a point q in the plane T to a ball B_i as follows. Imagine that we place a ball $B(q)$ of initial radius 0 at point q. We then inflate $B(q)$ while keeping it tangent to T at q, until it collides with B_i. We define the distance of q and B_i, which we denote by $\text{dist}(q, B_i)$, to be the radius of $B(q)$. More precisely, $\text{dist}(q, B_i)$ is the radius of the unique ball that is tangent to T at q and tangent to B_i. It is easy to show that $\text{dist}(q, B_i) = d(q, t_i)^2/4r_i$ where $d(q, t_i)$ denotes the Euclidean distance between q and t_i.

Since we have to detect collisions only with balls from $\mathcal{B}_S$ we can stop inflating when $B(q)$ is as large as the smallest ball in $\mathcal{B}_L$. Based on this, we define the threshold disk D_i of a ball $B_i \in \mathcal{B}_L$ as follows: a point $q \in T$ belongs to D_i if and only if $\text{dist}(q, B_i) \leq r_{\min}$ where $r_{\min}$ is the radius of the smallest ball in $\mathcal{B}_L$. It is straightforward to show that D_i is a disk whose radius is $2\sqrt{r_i \cdot r_{\min}}$ and whose center is t_i.

Clearly a ball $B_j \in \mathcal{B}_S$ cannot collide with a ball $B_i \in \mathcal{B}_L$ as long as t_j is outside D_i. In the following, we prove that a point $q \in T$ can be contained in at most a constant number of threshold disks. We start by proving a more general result, which we will need later when we replace the threshold disks by threshold boxes. For a given constant $c \geq 0$, let $c \cdot D_i$ denote the disk with radius $c \cdot \text{radius}(D_i)$ and center t_i.

Lemma 1. *The number of threshold disks D_j that are at least as large as a given threshold disk D_i and for which $c \cdot D_i \cap c \cdot D_j \neq \emptyset$, is at most $(8\,c^2 + 2\,c + 1)^2 + 1$.*

Proof. Let $\mathcal{D}(i)$ be the set of all threshold disks D_j that are at least as large as D_i and for which $c \cdot D_i \cap c \cdot D_j \neq \emptyset$. First we prove that there are no two balls B_j and B_k such that $r_k \geq r_j > 16\,c^2\,r_i$ and $D_j, D_k \in \mathcal{D}(i)$. Assume, for contradiction, that there are two balls B_j and B_k such that $r_k \geq r_j > 16\,c^2\,r_i$ and $D_j, D_k \in \mathcal{D}(i)$. Since B_j and B_k are disjoint, we have $d(t_j, t_k) \geq 2\sqrt{r_j \cdot r_k} > 8\,c\sqrt{r_k \cdot r_i}$. We also know that $d(t_j, t_k) \leq d(t_j, t_i) + d(t_i, t_k) \leq 8\,c\sqrt{r_k \cdot r_i}$ which is a contradiction. Hence, there is at most one ball B_j such that $r_j > 16\,c^2\,r_i$ and $D_j \in \mathcal{D}(i)$.

It remains to show that the number of balls B_j whose radii are not greater than $16\,c^2\,r_i$ and for which $D_j \in \mathcal{D}(i)$ is at most $(8\,c^2+2\,c+1)^2$. Let B_j be one of these balls and let x be a point in $c \cdot D_j \cap c \cdot D_i$. Since

$$d(t_i,t_j) \le d(t_i,x) + d(t_j,x) \le 2\,c\sqrt{r_i \cdot r_{\min}} + 2\,c\sqrt{r_j \cdot r_{\min}} \le (2\,c+8\,c^2)\,r_i\ ,$$

t_j must lie in a disk whose center is t_i and whose radius is $(2\,c+8\,c^2)\,r_i$. We also know that $d(t_j,t_k) \ge 2\sqrt{r_j \cdot r_k} \ge 2\,r_i$ for any two such balls B_j and B_k. Thus the set $\mathcal{D}'(i)$ of disks centered at t_j with radius r_i for all $D_j \in \mathcal{D}(i)$ are disjoint. Note that any disk in $\mathcal{D}'(i)$ lies inside the disk centered at t_i with radius $((2\,c+8\,c^2)+1)\,r_i$. Thus $|\mathcal{D}(i)| = |\mathcal{D}'(i)| \le (2\,c+8\,c^2+1)^2$. □

Lemma 2. *Each point $q \in T$ is contained in at most a constant number of threshold disks.*

The threshold box. The threshold disks have the important property that each point in T is contained in a constant number of disks. But unfortunately, as the balls in $\mathcal{B}_L$ and $\mathcal{B}_S$ move, it is difficult to detect efficiently whenever a tangency point enters or leaves a threshold disk. Hence we replace each threshold disk by its axis-aligned bounding box. The bounding box of a threshold disk D_i associated with a $B_i \in \mathcal{B}_L$ is called a *threshold box* and is denoted by $\mathrm{TB}(B_i)$. The following lemma states that the threshold boxes retain the crucial property of the threshold disks, namely, that each point $q \in T$ is contained in at most a constant number of threshold boxes. It follows fairly easily from Lemma 1.

Lemma 3. *Each point $q \in T$ is contained in at most a constant number of threshold boxes.*

Kinetic maintenance. Recall that to detect collisions between $\mathcal{B}_S$ and $\mathcal{B}_L$, for each ball $B_j \in \mathcal{B}_S$ we determine which threshold boxes of balls in $\mathcal{B}_L$ contain the tangency point t_j. Note that according to Lemma 3, t_j is contained in a constant number of threshold boxes. For each $B_j \in \mathcal{B}_S$ we maintain the set of threshold boxes that contain t_j and certificates that guarantees disjointness of B_j and the balls from $\mathcal{B}_L$ whose threshold boxes contain t_j.

To maintain our structure we only need to detect when a tangency point t_j enters or leaves a threshold box. To do so, we maintain two sorted lists on the x- and y-coordinates of the tangency points of $\mathcal{B}_S$ and the extremal points of the threshold boxes associated with the balls in $\mathcal{B}_L$. If the objects follow constant-degree algebraic trajectories, the number of events processed by our structure is quadratic in the size of of $\mathcal{B}_S$ and $\mathcal{B}_L$. Moreover, each event can be processed in constant time.

Lemma 4. *Let $\mathcal{B}_S$ and $\mathcal{B}_L$ be two disjoint sets of balls that roll on a plane where the balls in $\mathcal{B}_L$ are at least as large as the balls in $\mathcal{B}_S$. There is a KDS for collision detection between the balls of $\mathcal{B}_S$ and those of $\mathcal{B}_L$ that uses $O(|\mathcal{B}_S|+|\mathcal{B}_L|)$ space, and that processes $O((|\mathcal{B}_S|+|\mathcal{B}_L|)^2)$ events if the balls follow constant-degree algebraic trajectories. Each event can be handled in $O(1)$ time.*

3 Free-Flying Fat Objects in 3-Space

We now turn our attention to collision detection for a set $\mathcal{K}$ of n free-flying objects in 3-space. We will show how to obtain a compact and responsive KDS when $\mathcal{K}$ consists of convex, constant-complexity ρ-fat objects. Note that we do not require the objects to be polyhedral.

We will use the following definition of fatness [14]. An object K is called *ρ-fat*, for some $\rho \geq 1$, if there are two concentric balls $B^-(K)$ and $B^+(K)$ such that $B^-(K) \subset K \subset B^+(K)$ and

$$\text{radius}(B^+(K))/\,\text{radius}(B^-(K)) \leq \rho\,.$$

Since we are dealing with convex objects, this definition is equivalent up to constant factors to other definitions of fatness that have been used [7]. We call $\text{radius}(B^-(K))$ and $\text{radius}(B^+(K))$ the *inner radius* and *outer radius* of K, respectively, and we call the common center of $B^-(K)$ and $B^+(K)$ the *center* of K. We say that an object K is *larger* than another object K' if the inner radius of K is larger than the inner radius of K'.

Unfortunately the approach of the previous section does not work for free-flying objects, not even if we are dealing with balls. The problem is that the radius of the threshold ball of a ball B_i will now be $r_i + r_{\min}$ instead of $2\sqrt{r_i \cdot r_{\min}}$ and this invalidates the proof of Lemma 1 for $c > 1$ and thus invalidates Lemma 3.

3.1 Similarly Sized Objects

We first consider the case where the objects have similar sizes. More precisely, let σ be the *scale factor* of the scene, that is, the ratio between the sizes of the largest and the smallest inner ball:

$$\sigma = \frac{\max_{K\in\mathcal{K}} \text{radius}(B^-(K))}{\min_{K\in\mathcal{K}} \text{radius}(B^-(K))}$$

It follows from the results of Zhou and Suri [21] that the number of pairs of intersecting bounding boxes of the objects in $\mathcal{K}$ is at most $O(\rho\sqrt{\rho^3\sigma^3 n}) = O(\rho^2\sigma\sqrt{\rho\sigma n})$. (A similar but slightly weaker result also follows directly from results in Van der Stappen's thesis [20].) Hence, if σ is a constant, we can simply maintain the set of pairs of intersecting bounding boxes, and for each such pair add a certificate to test for disjointness of the corresponding objects.

To maintain the pairs of intersecting bounding boxes, we maintain three sorted lists: one on the minimum and maximum x-coordinates of the boxes, one on the minimum and maximum y-coordinates of the boxes, and one on the minimum and maximum z-coordinates of the boxes. Whenever there is a swap in one of these lists, two boxes may intersect or become apart. If two boxes intersect, we add a certificate for the corresponding objects. If they become apart, we remove the corresponding certificate. This leads to the following theorem.

Theorem 2. *For any set $\mathcal{K}$ of n convex, constant-complexity ρ-fat objects with scale factor σ, there is a KDS for collision detection that uses $O(\rho^2\sigma\sqrt{\rho\sigma}n)$ storage and processes $O(n^2)$ events in the worst case, assuming the objects follow constant-degree algebraic trajectories. Each event can be handled in $O(1)$ time.*

3.2 Arbitrarily Sized Objects

When the sizes of the objects vary greatly, then there can be a quadratic number of intersecting bounding boxes even when the objects are fat. Hence, a more sophisticated approach is needed. Our global strategy for this case is as follows. We place a number of so-called *guarding points*—or *guards*, for short—around each object $K \in \mathcal{K}$. The guards for K are defined in a local reference frame for K, so they follow the motion of K. We choose the guards in such a way that when two objects collide, the larger object must contain at least one guard from the smaller object. This reduces the collision-detection problem to maintaining for each guard which object contains it. The next lemma states that we can always find a small guarding set because the objects are fat.

Lemma 5. *For any ρ-fat object K, there is a set $G(K)$ of $O(\rho^6)$ guarding points such that any ρ-fat object K' that collides with K and is at least as large as K contains a point from $G(K)$.*

Our KDS for collision detection thus works as follows. For each object $K \in \mathcal{K}$ we compute a set $G(K)$ of guards according to Lemma 5. Our goal is now to maintain for each $g \in G(K)$ the object $K(g)$ containing g (if such an object exists). Let $\text{Cand}(K) := \{K(g) : g \in G(K)\}$; the set $\text{Cand}(K)$ contains the candidates with which we check for collisions. More precisely, for each object $K(g) \in \text{Cand}(K)$, our KDS has a certificate testing for the disjointness of K and $K(g)$.

Unfortunately, it seems difficult to maintain the set $\text{Cand}(K)$ directly. This would require us to detect when an object K' starts to contain a guard g, which is difficult to do efficiently. Hence, we replace the objects by their bounding boxes. Because the bounding boxes are axis-aligned, it will be easier to check whether any of them starts (or stops) to contain a guard of some other object. This introduces a new problem, however; a guard can be contained in many bounding boxes—see Figure 1. Clearly, we cannot afford to maintain for each guard g all the bounding boxes that contain it. Next we describe how to deal with this problem.

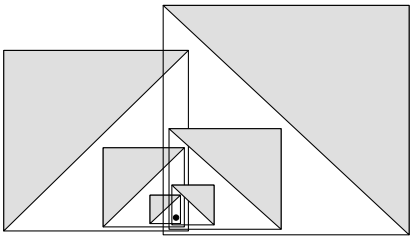


Fig. 1. A guard can be contained in many bounding boxes

Consider a guard g. As noted earlier, there can be many disjoint objects whose bounding boxes contain g. When this happens, however, the objects must become larger and larger, as shown in Figure 1, with the larger objects being "behind" the

smaller ones. Thus the smaller objects are the candidates for containing g. Hence, the idea is to maintain for g not all objects whose bounding boxes contain g, but only the smallest k such objects for some suitable constant k.

To make this idea work, we first partition the space around g into cones, as follows. Let U be the unit cube, centered at the origin. Draw a grid on each face of U, such that the grid cells have edge length $1/(2\sqrt{2}\rho)$. Triangulate each grid cell. We have now partitioned the surface of U into $O(\rho^2)$ triangles. Each triangle defines, together with the origin, an (infinite) cone γ. The set of cones for a guard g is obtained by translating these cones such that their apices—the origin in the construction—coincide with g. We denote this set by $\Gamma(g)$.

The next lemma implies that we can indeed restrict our attention to the smallest objects whose bounding box contains a guard g. For an object K, let $\mathrm{bb}(K)$ denote its (axis-aligned) bounding box.

Lemma 6. *Let $\mathcal{K}(\gamma)$ be the set of all objects K whose centers lie in a cone γ and such that $\mathrm{bb}(K)$ contains the apex g of the cone. Suppose that one of these objects, $K(g)$, contains g. Then $K(g)$ must be among the $8\rho^3(\sqrt{3}\rho+1)^3$ smallest objects in $\mathcal{K}(\gamma)$.*

To summarize, our KDS works as follows. For each object $K \in \mathcal{K}$ we compute a set $G(K)$ of guards according to Lemma 5. For each guard g we construct a collection $\Gamma(g)$ of infinite cones with apex g. For each cone $\gamma \in \Gamma(g)$ we maintain the $8\rho^3(\sqrt{3}\rho+1)^3$ smallest objects whose center is inside γ and whose bounding box contains g, and we have a certificate testing for disjointness for each such object with the object for which g is a guard. Next we describe a KDS that maintains all this information efficiently.

Details of the KDS. Let $G(\mathcal{K}) := \{G(K) : K \in \mathcal{K}\}$ denote the set of all guards over all objects, let $\Gamma(\mathcal{K})$ denote the collection of all the cones, that is, $\Gamma(\mathcal{K}) := \{\Gamma(g) : g \in G(\mathcal{K})\}$, and let $\mathrm{bb}(\mathcal{K})$ denote the set of bounding boxes of the objects.

Detecting events. We wish to maintain for each $\gamma \in \Gamma(g)$ the collection $\mathcal{K}^*(\gamma)$ of the $8\rho^3(\sqrt{3}\rho+1)^3$ smallest objects whose centers are inside γ and whose bounding boxes contain g. Such a collection can change only when one of the following two events happens:

Box event: a bounding box starts or stops to contain a guard.
Center event: a center moves into or out of a cone.

To detect box events, we maintain three sorted lists. The first list is sorted on x-coordinate and contains the guards in $G(\mathcal{K})$ as well as the bounding boxes, where each bounding box occurs twice (according to its maximum and minimum x-coordinates). We have similar lists sorted on y- and z-coordinates.

To detect center events, we observe that each cone is a translate of one of the $O(\rho^2)$ cones defined for the unit cube. Hence, the facets of the cones have only $O(\rho^2)$ distinct orientations. In fact, it is not difficult to see that there are only $O(\rho)$ orientations, because many orientations are re-used. Hence, we

can detect center events using $O(\rho)$ sorted lists, each containing the object centers and the guards according to one of those orientations. Since we have $O(\rho^6)$ guards per object, we get the following lemma.

Lemma 7. *The box and center events can be detected with a KDS that uses $O(\rho^7 n)$ storage and that processes $O(\rho^{13} n^2)$ events in total, assuming the objects follow constant-degree algebraic trajectories. Updating the KDS at such an event takes $O(1)$ time.*

Handling events. When we have detected a center event, we may have to update the set $\mathcal{K}^*(\gamma)$ of at most two cones. Next we describe how to handle the event involving an object K and some cone γ defined for a guard g.

When $\mathrm{bb}(K)$ starts to contain g, or when the center of K moves into γ, things are easy: When $\mathcal{K}^*(\gamma)$ contains less than $8\rho^3(\sqrt{3}\rho + 1)^3$ objects, we add K to $\mathcal{K}^*(\gamma)$; otherwise, we check whether K is smaller than the largest object in $\mathcal{K}^*(\gamma)$ and, if so, let it replace that object.

Handling the case where $\mathrm{bb}(K)$ stops to contain g, or when the center of K moves out of γ, is more difficult. For this we need a supporting data structure that can answer the following query:

> Given a cone γ with apex g, report the k smallest objects whose centers are in γ and whose bounding boxes contain g, where $k := 8\rho^3(\sqrt{3}\rho+1)^3$.

Recall that the set of cones can be partitioned into $O(\rho^2)$ subsets, where the cones in each subset are translates of some "standard" cone. We construct a data structure for each subset separately. Because the facets of the cones in a subset have only three distinct orientations, we can find all centers inside a query cone in with a three-level range tree. Finding the bounding boxes containing the apex of the query cone can be done with a three-level segment tree, and filtering out the k smallest objects requires a sorted list on the size of the objects. Hence, our total data structure will be a seven-level tree. Answering a query can be done in $O(\log^6 n + k)$ time—the query time is not $O(\log^7 n + k)$ because in the last level we only need to report the k smallest objects—and the amount of storage is $O(n \log^6 n)$. To kinetize the structure, we simply use the kinetic variants of range trees [5] and segment trees [6]. The number of events processed is $O(n^2)$.

Lemma 8. *When a center or box event occurs, we can update the collections $\mathcal{K}^*(\gamma)$ in $O(\log^6 n + \rho^6)$ time, using a supporting KDS that uses $O(\rho^2 n \log^6 n)$ storage. The supporting KDS processes $O(n^2)$ events in the worst case, assuming the objects follow constant-degree algebraic trajectories.*

This leads to our main result.

Theorem 3. *For any set $\mathcal{K}$ of n convex, constant-complexity ρ-fat objects, there is a KDS for collision detection that uses $O(\rho^2 n \log^6 n + \rho^7 n)$ storage and that processes $O(\rho^{13} n^2)$ events in the worst case, assuming the objects follow constant-degree algebraic trajectories. Each event can be handled in $O(\log^6 n + \rho^6)$ time.*

Our KDS is compact and responsive, but unfortunately it is not local: a large object K with many small objects around can be involved in many certificates, because it may contain guards for each of the small objects. However, we can show that the locality of our KDS depends on the ratio of the size of the biggest object and the smallest object in $\mathcal{K}$.

Lemma 9. *Each object in the KDS of Theorem 3 is involved in* $O(\rho^{14} + \rho^9\sigma^3)$ *certificates, where* σ *is the ratio of the largest inner radius to the smallest inner radius of the objects in* $\mathcal{K}$.

4 Conclusion

We presented the first KDS's for collision detection between multiple convex fat 3D objects that use a near-linear number of certificates and do not require the objects to have similar sizes. We believe that this is an important step forward in the theoretical investigation of KDS's for 3D collision detection. Our KDS for balls rolling on a plane is simple, and may perform well in practice. Our general KDS for free-flying objects of varying sizes, however, is complicated and the dependency on the fatness parameter ρ is large. Thus our result should be seen as a proof that good bounds are possible in theory—whether a simple and practical solution exists that achieves similar worst-case bounds is still open.

As remarked above, our structures are not local: a single object can be involved in a linear number of certificates. Unfortunately, this seems very hard (if not impossible) to avoid if there is a single large object that is closely surrounded by many tiny objects. Thus we do not expect to see a local KDS that can deal with arbitrarily sized objects. (We have shown though that a local KDS is possible for convex fat objects when their sizes are similar.)

Finally, a challenging open problem is to obtain results on non-convex and/or non-fat objects.

Acknowledgements. The last author would like to thank David Kirkpatrick for valuable discussions on the presented subject.

References

1. P. K. Agarwal, J. Basch, L. J. Guibas, J. Hershberger, and L. Zhang. Deformable free space tilings for kinetic collision detection. *International Journal of Robotics Research*, 21:179–197, 2002.
2. F. Aurenhammer and H. Edelsbrunner. An optimal algorithm for constructing the weighted Voronoi diagram in the plane. *Pattern Recognition*, 17(2):251–257, 1984.
3. J. Basch, J. Erickson, L. J. Guibas, J. Hershberger, and L. Zhang. Kinetic collision detection for two simple polygons. In *Proc. 10th ACM-SIAM Symposium on Discrete Algorithms*, pages 102–111, 1999.
4. J. Basch, L. Guibas, and J. Hershberger. Data structures for mobile data. *Journal of Algorithms*, 31:1–28, 1999.

5. J. Basch, L. Guibas, and L. Zhang. Proximity problems on moving points. In *Proc. 13th Symposium on Computational Geometry*, pages 344–351, 1997.
6. M. de Berg, J. Comba, and L. Guibas. A segment-tree based kinetic bsp. In *Proc. 17th Symposium on Computational Geometry*, pages 134–140, 2001.
7. M. de Berg, M. Katz, F. van der Stappen, and J. Vleugels. Realistic input models for geometric algorithms. *Algorithmica*, 34:81–97, 2002.
8. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer Verlag, Berlin, Germany, 2nd edition, 2000.
9. D. Coming and O. Staadt. Kinetic Sweep and Prune for Collision Detection. In *Proc. Workshop on Virtual Reality Interactions and Physical Simulations*, pages 81-90, 2005.
10. J. Erickson, L. Guibas, J. Stolfi, and L. Zhang. Separation-sensitive collision detection for convex objects. In *Proc. 10th ACM-SIAM Symposium on Discrete Algorithms*, pages 327–336, 1999.
11. L. Guibas. Kinetic data structures: A state of the art report. In *Proc. 3rd Workshop on Algorithmic Foundations of Robotics*, pages 191–209, 1998.
12. L. Guibas. Motion. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 1117–1134. CRC Press, 2nd edition, 2004.
13. L. Guibas, F. Xie, and L. Zhang. Kinetic collision detection: Algorithms and experiments. In *Proc. International Conference on Robotics and Automation*, pages 2903–2910, 2001.
14. M. Katz. 3-D vertical ray shooting and 2-D point enclosure, range searching, and arc shooting amidst convex fat objects. *Computational Geometry: Theory and Applications*, 8:299–316, 1998.
15. D. Kim, L. Guibas, and S.Y. Shin. Fast collision detection among multiple moving spheres. *IEEE Transactions on Visualization and Computer Graphics*, 4(3):230–242, 1998.
16. H.K. Kim, L. Guibas, and S.Y. Shin. Efficient collision detection among moving spheres with unknown trajectories. *Algorithmica*, 43:195–210, 2005.
17. D. Kirkpatrick, J. Snoeyink, and B. Speckmann. Kinetic collision detection for simple polygons. *International Journal of Computational Geometry and Applications*, 12(1&2):3–27, 2002.
18. D. Kirkpatrick and B. Speckmann. Kinetic maintenance of context-sensitive hierarchical representations for disjoint simple polygons. In *Proc. 18th ACM Symposium on Computational Geometry*, pages 179–188, 2002.
19. M. Lin and D. Manocha. Collision and proximity queries. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 787–807. CRC Press, 2nd edition, 2004.
20. A. van der Stappen. *Motion planning amidst fat obstacles*. PhD thesis, Utrecht University, Utrecht, the Netherlands, 1994.
21. Y. Zhou and S. Suri. Analysis of a bounding box heuristic for object intersection. In *Proc.10th ACM-SIAM Symposium on Discrete Algorithms*, pages 830–839, 1999.

Dynamic Connectivity for Axis-Parallel Rectangles

Peyman Afshani and Timothy M. Chan*

School of Computer Science
University of Waterloo
Waterloo, Ontario, N2L 3G1, Canada
{pafshani, tmchan}@uwaterloo.ca

Abstract. In this paper we give a fully dynamic data structure to maintain the connectivity of the intersection graph of n axis-parallel rectangles. The amortized update time (insertion and deletion of rectangles) is $O(n^{10/11} \operatorname{polylog} n)$ and the query time (deciding whether two given rectangles are connected) is $O(1)$. It slightly improves the update time ($O(n^{0.94})$) of the previous method while drastically reducing the query time (near $O(n^{1/3})$). Our method does not use fast matrix multiplication results and supports a wider range of queries.

1 Introduction

Dynamic connectivity for undirected graphs is one of the most basic problems in data structure design and has been extensively studied [7, 9, 10, 13, 14, 15]. Currently the best method, due to Thorup [15], supports insertions and deletions of edges in $O(\log n \log^3 \log n)$ randomized amortized time and can determine whether two vertices are connected in $O(\log n / \log \log \log n)$ time, where n denotes the number of vertices in the graph.

In this paper, we investigate geometric versions of the problem. Perhaps the simplest, and certainly one of the most naturally appealing, version concerns intersection graphs of orthogonal (horizontal and vertical) line segments. Such graphs arise in applications from VLSI design, geographic information systems, and other areas. In the dynamic setting, we want to answer connectivity queries between any two segments, while supporting insertions and deletions of segments.

Surprisingly, this simple-sounding dynamic geometric problem turns out to be quite difficult, more so than the original graph problem. For one thing, we cannot afford to maintain the intersection graph explicitly, because the insertion or deletion of a single object can bring forth as many as $\Omega(n)$ edge updates in the graph every time.

In STOC 2002, the second author [2] discovered the only nontrivial fully dynamic result for the problem known to date: a data structure that has $O(n^{0.939})$ amortized update time and $\tilde{O}(n^{1/3})$ query time. The $\tilde{O}$ notation hides

* This work has been supported by an NSERC grant.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 16–27, 2006.

polylogarithmic factors throughout this paper. This data structure more generally works for connectivity queries for axis-parallel rectangles or boxes in any fixed dimension.

The approach in the previous paper was to use so-called "bi-clique covers" to compactify the intersection graph. This process reduces the geometric problem to a new dynamic graph problem: how to maintain an undirected graph under not only edge updates but also vertex updates—namely, turning a vertex "on" or "off"—so that connectivity queries can be answered in the subgraph induced by the "on" vertices. The paper calls this the *dynamic subgraph connectivity* problem. This graph problem was then solved by a combination of techniques, including the use of Coppersmith and Winograd's fast matrix multiplication algorithm [5].

It was observed [2] that connectivity for axis-parallel line segments or boxes in any fixed dimension $d \geq 3$ is equivalent (up to polylogarithmic factors) to dynamic subgraph connectivity, and that dynamic subgraph connectivity is related to matrix multiplication. Thus, the approach taken is the "right" one in higher dimensions. However, the possibility of a different approach that exploits specifically the geometry of the two-dimensional case was left open.

The main result of this paper is a new fully dynamic data structure for connectivity queries among n axis-parallel line segments, or more generally, axis-aligned rectangles in two dimensions. The amortized update time is $\tilde{O}(n^{10/11}) = O(n^{0.910})$, and the query time is $O(1)$.

Although this update time is still fairly large and the improvement may not seem dramatic, we believe that the result is important for several reasons. First, the new method does not use overly complicated matrix multiplication algorithms and is entirely based on "elementary" techniques, and is thus actually implementable. (In the previous method, if Coopersmith and Winograd's algorithm is replaced by Strassen's, the update time would increase to $O(n^{0.984})$.) Second, the new method supports queries in addition to connectivity between two objects; for example, we can decide whether the entire intersection graph is connected, or count the number of connected components. The previous method inherently cannot deal with such queries of a global nature. Third, our significantly lower query time is attractive, especially in applications where queries are more frequent than updates. Finally, the geometric techniques we use are interesting and original; in particular, we introduce a simple but crucial combinatorial lemma about disjoint curves in the plane. This lemma appears new, to the best of our knowledge. (If not, its algorithmic significance has certainly been overlooked; for instance, it leads to at least one new intersection-searching result that cannot be obtained by previous techniques in the area.) We start with this lemma in the next section and then proceed to describe the main algorithm.

2 A Simple Combinatorial Lemma

In this section we prove a combinatorial lemma which later will be used in the analysis of the final algorithm. Consider a set R of n disjoint regions with simply connected boundaries and a set C of disjoint simple curves in the plane. We say

two curves are *equivalent* if they cross the exact same set of regions from R. Generally, there could be up to 2^n curves such that no two are equivalent (one for each subset of R). However, we aim to show that if the curves are disjoint, then we cannot have too many such curves. In fact, there can be at most $\Theta(n^3)$ curves such that no two are equivalent.

We mention that a slightly weaker bound can be obtained by using *VC-dimension* techniques [12]. It is possible to show (by a K_5-avoidance argument) that the set system defined by the curves (where the ground set is R and each curve defines the set of regions it intersects) has VC-dimension four. This implies that the number of curves is $O(n^4)$. We omit the details, as the approach we now give produces a better bound:

Lemma 1. *Assume C is a set of pairwise non-crossing curves with common endpoints p and q, $p \neq q$. If no two curves are equivalent in C, then $|C| = O(n)$.*

Proof. Let $c_1, c_2, \ldots, c_m$ be the given curves, ordered clockwise around p. For every i, consider two adjacent curves c_i and c_{i+1}. These two curves are not equivalent, so they do not pass through the same set of regions. This implies that there is at least one region r which intersects exactly one of them. We charge c_i to r. Obviously, we have charged $m-1$ units in total. In the clockwise ordering of the curves, consider the first curve c_i and the last curve c_j which pass through r. All the curves from c_i to c_j intersect r and all the curves from c_j to c_i do not intersect r. Thus the total charge of r is at most two since only c_{i-1} and c_j can be charged to r. This proves $|V| = O(n)$. □

Lemma 2. (The Main Lemma) *If C is a set of disjoint curves containing no pairwise equivalent curves, then $|C| = O(n^3)$.*

Proof. Consider one curve $c \in C$. Begin from one endpoint of c and start erasing (or shrinking) the curve from that endpoint. This process can be viewed as moving the endpoint along the curve. We shrink the curve until the endpoint of the curve lies on a boundary point p of a region r such that r and c only intersect at p. We repeat the same process for the other endpoint of c and call the resulting curve c'. In other words, the curve c' is a minimal sub-curve of c which intersects the exact same regions as c; thus this operation preserves the equivalence relation. (See Figure 1.)

We do this operation on all the curves in C. Let C' be the set of shrunk curves. Every curve $c \in C'$ has the property that it starts from (and ends at) the boundary of a region r and never passes through that region again. For two regions r_i and r_j, let C_{ij} be the set of curves that have one endpoint in r_i and another endpoint in r_j. If we consider only the curves in C_{ij}, then we can contract the regions r_i and r_j to two points and apply Lemma 1. This implies that C_{ij} contains $O(n)$ curves. (For the special case $i = j$, we have $|C_{ii}| \leq 1$, since at most one curve is entirely contained in r_i.) There are $O(n^2)$ different sets of C_{ij} and thus the total number of curves in C' and C is $O(n^3)$. □

The above lemma is good enough to lead to new results for dynamic rectangle connectivity, but the following refinement will lead to a slightly better result:

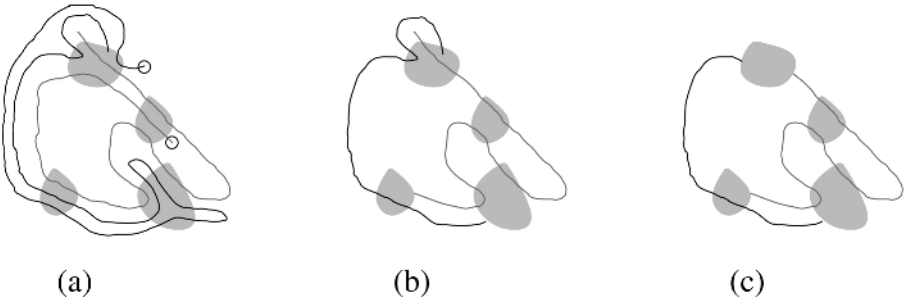


Fig. 1. (a) Erasing from the endpoints marked with circle. (b) Erasing from the other endpoints. (c) The final curves.

Lemma 3. *If C is a set of disjoint curves containing no pairwise equivalent curves and each curve intersects at most k regions, then $|C| = O(nk^2)$.*

Proof. We adapt a random sampling idea by Clarkson and Shor [4] that was originally used for the "$(\leq k)$-set" problem.

Take a random sample $Q \subseteq R$ where each region is included with probability $1/k$. We define a planar (multi)graph G_Q where vertices are the contracted regions of Q, as follows. Assume the curves have been shrinked as in the earlier proof. Fix two regions r_i and r_j. Let $c_1, \ldots, c_m$ be the curves in C_{ij} in clockwise order around r_i. For $m = 1$, if r_i and r_j are in Q and all of the $\leq k$ regions intersecting c_1 are not in Q, then add c_1 to G_Q as an edge between r_i and r_j. Observe that the probability that c_1 is added is at least $1/k^2(1-1/k)^k = \Omega(1/k^2)$. For $m > 1$, take each consecutive pair (c_t, c_{t+1}) and let $r(c_t, c_{t+1})$ be a region intersected by c_{t+1} but not c_t, or a region intersected by c_t but not c_{t+1}. Color the pair *red* in the former case, and *blue* otherwise. Without loss of generality, assume that at least half of all pairs are red. For a red pair (c_t, c_{t+1}), if r_i, r_j, and $r(c_t, c_{t+1})$ are in Q and all of the $\leq k$ regions intersecting c_t are not in Q, then add the curve c_t to the graph G_Q as an edge between r_i and r_j. Observe that the probability that c_t is added is at least $1/k^3(1-1/k)^k = \Omega(1/k^3)$. Thus, $E[|E(G_Q)|] = \Omega(|C|/k^3)$.

On the other hand, $E[|V(G_Q)|] = O(n/k)$. By Euler's formula, every planar graph with all face lengths at least 3 (in particular, every simple planar graph) has a linear number of edges. Our graph G_Q is planar but not simple. However, between any 2 parallel edges, there is at least one vertex in G_Q: namely, if the red pairs (c_t, c_{t+1}) and (c_u, c_{u+1}), $t < u$, define 2 edges between r_i and r_j in G_Q, then $r(c_t, c_{t+1})$ would lie entirely between c_t and c_u. Because of this property, we may assume that G_Q has no faces of length 2 (by adding extra edges to isolated vertices if necessary). Thus, $E[|E(G_Q)|] = O(n/k)$.

We can conclude that $|C|/k^3 = O(n/k)$. □

We apply Lemmas 2 and 3 to the case of connected components formed by a set of axis-parallel rectangles, or more generally, polygons. Assume we have a set of polygons which form m connected components and a set R of n disjoint regions

as before. We say two connected components are equivalent iff the set of regions they cross is identical.

Corollary 1. *Consider a set S of simple polygons forming a set C of connected components. If C contains no pairwise equivalent components, then $|C| = O(n^3)$. Furthermore, if X is the total number of crossings of the polygons' boundaries with the regions, then $|C| = O(n + n^{1/3}X^{2/3})$.*

Proof. If a component c_i completely contains a region r_j, then we delete both c_i and r_j. Since no other component can intersect r_j, this operation preserves the equivalence relation. By repeating this operation, we remove a total of $O(n)$ components and at the end no region is completely contained in a connected component. Thus, a region r_j intersects c_i iff r_j intersects the boundary of a polygon in c_i. So, it suffices to consider a set S of line segments rather than polygons.

Pick one connected component c_i and look at the arrangement created by the line segments of c_i. Pick one cell except the outer cell of the arrangement and cut one bounding segment of this cell at some arbitrary point (as in Figure 2). This operation connects this cell to its neighboring cell. We repeat this for all the other remaining cells until only one cell, the outer cell, is left in the arrangement. Define the curve c_i' as the Eulerian tour of this arrangement.

The resulting curves are disjoint and they cross the same set of regions as their corresponding connected component. Using Lemma 2 we conclude that $|C| = O(n^3)$.

To get a bound sensitive to X, observe that there are at most X/k components intersecting more than k regions. Using Lemma 3 we conclude that $|C| = O(nk^2 + X/k)$. We can set $k = (X/n)^{1/3}$. □

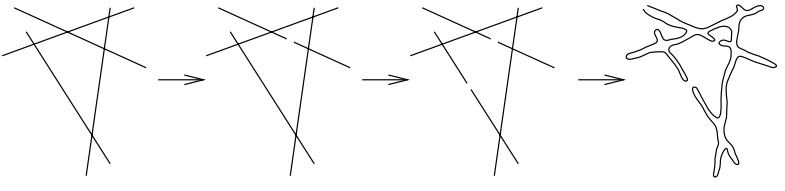


Fig. 2. Changing a connected component into a closed curve

Remark: The bounds in Lemmas 2 and 3 and Corollary 1 are all tight, as we can see from the following example of a set R of $\Theta(n)$ regions and a set C of $\Theta(nk^2)$ curves with $X = \Theta(nk^3)$, for any given $k \leq n$: Let R contain the $2k$ vertical segments $\{i\} \times [0, n+1]$ for $i = -k, \ldots, -1$ and $i = 1, \ldots, k$, as well as n short vertical segments $\{0\} \times [t-\varepsilon, t+\varepsilon]$ for $t = 1, \ldots, n$. Let C contain nk^2 horizontal segments $[i, j] \times \{t\}$ for all $i = -k, \ldots, -1$, $j = 1, \ldots, k$, and $t = 1, \ldots, n$. Small perturbations can ensure that the segments in C are disjoint. No two segments in C are equivalent.

3 Dynamic Connectivity for Rectangles

We begin with a few preliminaries which will act as building blocks for the final fully dynamic algorithm. Let S be a set of n axis-parallel rectangles in the plane. Fix a parameter q and build a $q \times q$ grid by drawing vertical (and similarly horizontal) lines at every $4n/q$-th corner point of S. This construction ensures that each vertical or horizontal slab contains $O(n/q)$ corners. We call this grid a q-grid. Let the set of regions R be the set of $\Theta(q^2)$ non-crossing (vertical and horizontal) line segments which form this grid and let C be the set of connected components of S. We define equivalence as before and give subroutines that help in computing and maintaining the corresponding equivalence classes.

3.1 Equivalence-Class Management and Decremental Connectivity

Let c be a connected component of the rectangles. We begin by defining a canonical *representation* for the set of regions (grid segments) intersected by c, with the intention that two components are equivalent iff their representations are identical. (A naive bit-vector representation of size $\Theta(q^2)$ would be too long for our purposes.)

First, if a rectangle in c entirely contains a region r, then c is the only component of its class. In this case we store r as the representation for c. If this is not the case, then we proceed to find a representation for the set of regions intersected by b, the union of all line segments bounding the rectangles in c. Consider i-th row of the grid (a horizontal slab) and let $r_{i,0}, \cdots, r_{i,q}$ be the regions (vertical grid segments) contained in this row, ordered from left to right. We represent the regions intersected by b in this row by a list of *intervals*, $(r_{i,j_1}, r_{i,j'_1}), \cdots, (r_{i,j_k}, r_{i,j'_k})$, where $j_1 \leq j'_1 \leq \cdots \leq j_k \leq j'_k$. Here, an interval $(r_{i,j}, r_{i,j'})$ indicates that b intersects regions $r_{i,j+1}$ to $r_{i,j'-1}$ but not the regions $r_{i,j}$ and $r_{i,j'}$. We build a similar representation for the columns of the grid (vertical slabs) and define the representation of c to be the concatenation of these lists for all the rows and columns of the grid. Note that the size of this representation is $O(\min\{|c|, q^2\})$, where $|c|$ denotes the number of rectangles in c.

Lemma 4. *Given a q-grid and a set of n axis-parallel rectangles, we can find the connected components and the set of equivalence classes in $\tilde{O}(n)$ time.*

Proof. The connected components can be found in $O(n \log n)$ time by a sweep-line algorithm [11]. To build the classes, we need to compute the representation for each connected component c. Within each row, the key subproblem is to compute the union of the x-intervals of the horizontal segments contained in the row. By sorting and scanning, the union of any m given (one-dimensional) intervals can be constructed in $O(m \log m)$ time. Within each column, we have a similar subproblem. The total time to compute the representation for c is therefore $O(|c| \log |c|)$.

If we interpret the representation of the components as long strings, we can compare the representation of two components c_1 and c_2 in $O(\min\{|c_1|, |c_2|\})$ time using the lexicographical ordering of strings. Since the total size of these

strings is $O(n)$, we can sort the strings lexicographically in $\tilde{O}(n)$ time, for example, by mergesort. This will put all the elements of the same class in consecutive order. Finally, a linear scan can be used to separate the components into classes. □

Lemma 5. *Given a connected component c and a set of classes L sorted lexicographically by their representations, in time $O(|c| \log n)$ we can find a class $\ell \in L$ corresponding to c or conclude that no such class exists.*

Proof. We can use binary search on the sorted list of representations to find the proper component. Just note that each comparison takes $O(|c|)$ time. □

We need one more subroutine for our final algorithm: a *decremental* data structure that maintains connectivity of a set of rectangles under deletions. In [2] it was noted that by using a compact representation (bi-clique covers) for the intersection graph of the rectangles, we can obtain a decremental algorithm with $\tilde{O}(1)$ amortized update time. This is achieved by applying a known decremental connectivity algorithm for graphs, e.g., [14], which can explicitly maintain the connected components and in particular answer queries in constant time. Thus we have the following lemma.

Lemma 6. *Given a set of n axis-parallel rectangles, we can maintain the connected components under any sequence of deletions in $\tilde{O}(n)$ total time and answer queries in constant time.*

3.2 The Fully Dynamic Method

Our overall strategy is similar to the overall strategy of the previous method [2] (which in turn is based on an idea from [3]): we will be "lazy" about insertions but periodically rebuild the data structure to limit the "damage" caused by these insertions. Let r be a parameter which will be determined later. After every r updates we rebuild the whole data structure so that we can assume at any given time there have been less than r updates.

The preprocessing and data structure: We build a q-grid and compute the connected components and the corresponding set of classes according to Lemma 4. The amortized cost of the preprocessing over r updates is $\tilde{O}(n/r)$.

Let M be the maximum number of equivalence classes. According to Corollary 1, $M = O(|R|^3) = O(q^6)$. Noting that the number of crossings of the rectangles' boundaries with the q-grid is $X = O(nq)$, we also get an alternative bound $M = O(|R| + |R|^{1/3}X^{2/3}) = O(q^2 + q^{4/3}n^{2/3})$.

The data structure maintains the connectivity of a graph H that contains three types of vertices:

1. Rectangle vertex, i.e., a vertex corresponding to a rectangle inserted after the latest rebuild.
2. Class vertex, i.e., a vertex corresponding to an equivalent class for some subset of the connected components of the rectangles not represented by the rectangle vertices.
3. Component vertex, i.e., a vertex corresponding to a connected component not represented by the class vertices.

For each class vertex, we store a list of its connected components. The components (in both class and component vertices) are maintained in a decremental connectivity data structure by Lemma 6. For each component, we also keep a data structure for orthogonal intersection search, e.g., [6].

With a slight abuse of notation we use vertices of H to refer to their corresponding geometric objects as well. The graph H is defined as the intersection graph of the three types of geometric objects listed above and is stored in a connectivity data structure that supports polylogarithmic edge updates, e.g., [10]. We will ensure the following invariant: whenever there is an edge between a rectangle vertex s_i and a class vertex ℓ_j, the rectangle s_i intersects *all* the connected components in ℓ_j.

Initially, the graph has isolated class vertices and no rectangle or component vertices. An exception is the class corresponding to components intersecting no regions; we break down this class into $O(q^2)$ class vertices, one vertex for all the components lying inside a cell.

We will ensure that each update deletes at most $O(n/q)$ vertices of H and that the number of component vertices at the end is bounded by $O(rn/q)$. We already know that the number of class vertices at the end is at most M. Thus the total number of vertices of H created during the r updates is $O(M + rn/q)$. Now, we describe the update and query algorithms.

Insertion of a rectangle s: Consider the two horizontal slabs and two vertical slabs of the grid containing the corners of s. These *special* slabs contain $O(n/q)$ corners and thus there are $O(n/q)$ components with a corner in these slabs. We go through each such component c_i and remove them from their corresponding class ℓ_j and add one component vertex c_i to the graph H. Since c_i was previously a part of ℓ_j, we add an edge from c_i to each vertex of H adjacent to ℓ_j. We delete all the empty class vertices.

Now we claim that the invariant still holds, i.e., if a connected component c_1 of a class vertex ℓ intersects s, then any other member c_2 of ℓ intersects s as well. We consider two cases. The first case is when s entirely contains c_1. Note that if c_1 does not intersects any region then its cell must be contained by S since we have removed all the components from the special slabs of s. So we can assume c_1 intersects at least one region. Pick any region r_1 intersected by c_1. Since c_1 does not have any corners in the special slabs (otherwise it would be removed), s entirely contains r_1. Since c_1 and c_2 are equivalent, c_2 must also intersect r_1, and thus s. The second case is when c_1 intersects a bounding segment s' of s; say s' is horizontal. Since c_1 does not have any corners in the special slabs, we can find a grid cell (rectangle) such that c_1 cuts through the cell vertically while s' cuts through it horizontally. Since c_1 and c_2 are equivalent and c_2 does not have any corners in the special slabs either, c_2 must cut through the same cell vertically and must intersect s', and thus s.

Next we add a rectangle vertex s to H, and add an edge to each vertex it intersects. We can decide whether a component intersects s in polylogarithmic time by querying an orthogonal intersection search structure [6]. According to

the invariant, for a class vertex, we only need to examine one of its components. Hence insertion can be performed in $\tilde{O}(|V(H)|) = \tilde{O}(M + rn/q)$ time. Notice that each insertion deletes $O(n/q)$ vertices and adds $O(n/q)$ new component vertices to H, which is acceptable.

Deletion of a rectangle s: For deletion, we use a "weighted split" strategy (like in [2]). We consider two cases.

- s is a rectangle vertex in H: In this case we simply remove the vertex s and at most $O(M + rn/q)$ incident edges.
- s is a rectangle inside a connected component c (either located inside a class vertex or a component vertex): In this case we remove s from the component by the decremental connectivity data structure. The amortized cost of this operation is $\tilde{O}(n/r)$ over r updates. This operation splits c into smaller connected components, $c_1, \ldots, c_z$ sorted in decreasing order according to size. Let $m = |c_2| + \cdots + |c_z|$. We can rebuild the classes corresponding to all the rectangles in $c_2, \ldots, c_z$ in $\tilde{O}(m)$ time according to Lemma 4 and insert them into the class vertices of H within the same time bound according to Lemma 5. Components that have a corner in the same slab as one of the $O(r)$ corners of the rectangle vertices, however, are moved to new component vertices; the earlier argument implies that this preserves the invariant. Note that the number of such component vertices at the end is $O(rn/q)$, which is acceptable. We can obtain intersection-search structures for $c_1, \ldots, c_z$ from the structure for c by performing $O(m)$ update operations in $\tilde{O}(m)$ additional time.

  For each new class/component vertex created in this phase, we add an edge to each of the $O(r)$ rectangle vertices it intersects. Since the total number of vertices created during the r updates is $O(M + rn/q)$, the total cost of this step is $\tilde{O}(rM + r^2 n/q)$, which yields an amortized cost of $\tilde{O}(M + rn/q)$.

  The size of each c_i, $2 \leq i \leq z$ is at most half the size of c. Thus the sum of all the m values encountered during the entire update sequence is $O(n \log n)$. This implies that the amortized time for processing the smaller components is $\tilde{O}(n/r)$. To take care of c_1 we simply remove it from its corresponding class vertex and add a single component vertex to H and add edges to all rectangle vertices it intersects, in $\tilde{O}(r)$ additional time.

After each update we regenerate the connected components of the graph H in $\tilde{O}(M + rn/q)$ time. We conclude that the overall amortized update time is $\tilde{O}(n/r + q^2 + q^{4/3}n^{2/3} + rn/q)$, which is asymptotically minimized by picking $q = r^2$ and $r = n^{1/11}$.

Connectivity query between rectangles u and v: Given pointers to u and v, we want to determine whether u and v are connected. We first find the components containing u and v from the decremental structure in constant time (by Lemma 6). If u and v are inside the same component (either in a component vertex or class vertex), we return "yes". If they are in different components but inside the same class vertex ℓ, we return "yes" iff ℓ is not an isolated vertex in H.

Otherwise, we return "yes" iff the vertex containing u and the vertex containing v are connected in the graph H. Since we know all the connected components of H, the query time is $O(1)$.

Thus we have proved the following theorem.

Theorem 1. *Given n axis-parallel rectangles, there exists a deterministic fully dynamic data structure which performs updates in $\tilde{O}(n^{10/11}) = O(n^{0.910})$ amortized time and answers connectivity queries in constant time.*

Remarks: Note that we can determine the connectivity between two points in the plane in polylogarithmic time, simply by performing orthogonal range search queries to find two rectangles containing the two points.

Our method enables us to answer many other types of queries by storing additional information. For example, the following queries cannot be handled by the previous method [2].

- We can determine the number of connected components in the given set of rectangles in $O(1)$ time: we just record the number of components that are in isolated class vertices of H, as well as the number of connected components in the graph H itself, after each update.
- We can determine whether the entire set of rectangles is connected in $O(1)$ time, just by checking whether the number of connected components is 1.
- We can list all rectangles in the same connected component as a given rectangle s in time proportional to the output size: if s is in an isolated class vertex, we just report all rectangles in the component of s from the decremental structure; otherwise, we report all rectangles that appears in the vertices of the connected component of the graph H containing s.

4 More Results

4.1 An Offline Method

It is possible to improve the update time of the algorithm if the list of insertions is known in advance. In this case queries and deletions may be online. Let I be the set of rectangles of the r future insertions. Instead of using a q-grid, we let the set of regions R be the set of all $O(r^2)$ non-crossing line segments in the arrangement of the bounding segments in I. We build equivalence classes with respect to this set of regions. We use the bound $M = O(r^6)$ on the number of equivalence classes.

The rest of the method is mostly the same, except that the invariant is more easily satisfied: when inserting a rectangle s, we know that each of its bounding segments s' is covered exactly by regions of R, so obviously if one connected component $c_1 \in \ell$ intersects s', then all other members of ℓ intersect s' as well. For this reason we can skip the creation of $O(rn/q)$ component vertices. Thus the number of vertices of H is reduced to $O(r^6)$. This implies that the amortized cost of an update is $\tilde{O}(r^6 + n/r)$, which is asympotically minimized for $r = n^{1/7}$ and yields an update time of $\tilde{O}(n^{6/7}) = O(n^{0.858})$.

4.2 Intersection Searching for Disjoint Rectilinear Polygons

We can also prove the following result by our techniques:

Theorem 2. *Given a set of disjoint simple rectilinear polygons in the plane of total complexity n, we can build a data structure in $\tilde{O}(n)$ time and space, so that we can count the number of polygons intersecting a query rectangle in $\tilde{O}(n^{6/7})$ time.*

Proof. We build a q-grid and the $M = O(q^6)$ equivalence classes for the polygons as in the previous section. For each polygon, we keep an intersection-search data structure. For each equivalence class ℓ, we record the number of its components. As before, we break down the class of components intersecting no regions into $O(q^2)$ smaller classes, one for each cell of the grid.

Given a query rectangle s, we go through the $O(n/q)$ polygons with a corner in one of the special slabs containing the corners of s and increase the counter iff it intersects s. This takes $\tilde{O}(n/q)$ time. We also mark these components to prevent double counting in the next step.

By an earlier argument, we know that if an unmarked component of a class ℓ intersects s, then any other unmarked member of ℓ intersects s as well. So, we go through each class ℓ, and if an arbitrary unmarked component in ℓ intersects s, we increase the counter by the number of unmarked components in ℓ. This takes $\tilde{O}(q^6)$ additional time. The total query time $\tilde{O}(q^6 + n/q)$ is asymptotically minimized for $q = n^{1/7}$. □

Although the above intersection-searching result is somewhat specialized, we think it is of theoretical interest in view of previous work. For example, by standard results on orthogonal range/intersection searching [1], we can count the number of intersections of the polygons with an orthogonal line segment. However, for our problem, segments from the same polygon should be counted only once. On the other hand, by simple known techniques for *colored* intersection searching [8], we can report (the labels of) the polygons intersecting a rectangle in time $O(k \operatorname{polylog} n)$, where k is the number of output polygons. These techniques do not require disjointness of the given rectilinear polygons; however, they do not yield results that are independent of k for the counting problem.

5 Conclusion

We have presented a data structure with sublinear update time and constant query time which has the additional advantage of being able to answer global connectivity queries. Our work leaves a few open questions. The obvious one is reducing the update time, but probably the most challenging open problem is achieving a meaningful lower bound as it seems unlikely that this problem can be solved in polylogarithmic time.

Another direction is to solve the dynamic connectivity problem for other classes of geometric graphs, for instance the intersection graph of arbitrary line

segments. Neither the previous technique [2] nor the new technique can be directly applied to this class of graphs (because the bi-clique cover of these graphs does not have linear size, and the management of the equivalent classes also becomes more difficult).

References

1. P. K. Agarwal and J. Erickson. Geometric range searching and its relatives. In *Advances in Discrete and Computational Geometry*, pages 1–56. AMS Press, 1999.
2. T. M. Chan. Dynamic subgraph connectivity with geometric applications. In *Proc. 34th ACM Sympos. on Theory of Comput.*, pages 7–13, 2002.
3. T. M. Chan. Semi-online maintenance of geometric optima and measures. *SIAM J. Comput.*, 32:700–716, 2003.
4. K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete Comput. Geom.*, 4:387–421, 1989.
5. D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *J. Symbolic Comput.*, 9:251–280, 1990.
6. H. Edelsbrunner and H. A. Maurer. On the intersection of orthogonal objects. *Inform. Process. Lett.*, 13:177–181, 1981.
7. M. Fredman and M. Henzinger. Lower bounds for fully dynamic connectivity problems in graphs. *Algorithmica*, 22:351–362, 1998.
8. P. Gupta, R. Janardan, and M. Smid. Computational geometry: generalized intersection searching. In *Handbook of Data Structures and Applications*, pages 64–1–64–17. Chapman & Hall/CRC, Boca Raton, FL, 2005.
9. M. R. Henzinger and V. King. Randomized dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46:76–103, 2000.
10. J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48:723–760, 2001.
11. H. Imai and T. Asano. Finding the connected components and a maximum clique of an intersection graph of rectangles in the plane. *J. Algorithms*, 4(4):310–323, 1983.
12. J. Matoušek. *Lectures on Discrete Geometry*. Springer-Verlag, 2002.
13. M. Pătraşcu and E. D. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM J. Comput.*, 35(4):932–963, 2006.
14. M. Thorup. Decremental dynamic connectivity. *J. Algorithms*, 33(2):229–243, 1999.
15. M. Thorup. Near-optimal fully-dynamic graph connectivity. In *Proc. 32nd ACM Sympos. on Theory of Comput.*, pages 343–350, 2000.

Single Machine Precedence Constrained Scheduling Is a Vertex Cover Problem

Christoph Ambühl[1] and Monaldo Mastrolilli[2]

[1] University of Liverpool - Great Britain
christoph@csc.liv.ac.uk
[2] IDSIA - Switzerland
monaldo@idsia.ch

Abstract. In this paper we study the single machine precedence constrained scheduling problem of minimizing the sum of weighted completion time. Specifically, we settle an open problem first raised by Chudak & Hochbaum and whose answer was subsequently conjectured by Correa & Schulz.

The most significant implication of our result is that the addressed scheduling problem is a special case of the vertex cover problem. This will hopefully be an important step towards proving that the two problems behave identically in terms of approximability.

As a consequence of our result, previous results for the scheduling problem can be explained, and in some cases improved, by means of vertex cover theory. For example, our result implies the existence of a polynomial time algorithm for the special case of two-dimensional partial orders. This considerably extends Lawler's result from 1978 for series-parallel orders.

1 Introduction

We address the problem of scheduling a set $N = \{1, \ldots, n\}$ of n jobs on a single machine. The machine can process at most one job at a time. Each job j is specified by its length p_j and its weight w_j, where p_j and w_j are nonnegative integers. We only consider *non-preemptive* schedules, in which all p_j units of job j must be scheduled consecutively. Jobs have precedence constraints between them that are specified in the form of a directed acyclic graph $G = (N, P)$, where $(i, j) \in P$ implies that job i must be completed before job j can be started. We assume that G is transitively closed, i.e., if $(i, j), (j, k) \in P$ then $(i, k) \in P$. The goal is to find a schedule which minimizes the sum $\sum_{j=1}^{n} w_j C_j$, where C_j is the time at which job j completes in the given schedule. In standard scheduling notation (see e.g. Graham et al. [8]), this problem is known as $1|prec|\sum w_j C_j$. The general version of $1|prec|\sum w_j C_j$ was shown to be strongly NP-hard by Lawler [12] and Lenstra & Rinnooy Kan [14].

Nevertheless, special cases are known to be polynomial-time solvable. In 1956, Smith [25] showed that, in absence of precedence constraints, an optimal solution

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 28–39, 2006.

could be found by sequencing the jobs in non-increasing order of the ratio w_i/p_i. Afterwards, several other results for special classes of precedence constraints were proposed. All of them culminated in the work by Lawler [12], who gave an $O(n \log n)$ time algorithm for solving $1|prec|\sum w_j C_j$ when the given precedence constraints are series-parallel. Goemans & Williamson [7] provided a nice alternative proof for the correctness of Lawler's algorithm by using a two-dimensional Gantt chart. The series-parallel precedence constraints represent the most important class known to be polynomially solvable to date. Several authors worked on finding larger classes of polynomially solvable instances, mainly by considering precedence constraints which are lexicographic sums [27] of polynomially solvable classes (see [13] for a survey).

Two-dimensional partial orders represent an important generalization of the series-parallel case [16]. However, determining its complexity is a long-standing open problem. A first attempt [26] dates back to 1984, and the currently best known approximation factor is 3/2 [5]. Other restricted classes of precedence constraints, such as interval orders and convex bipartite precedence constraints, have been studied (see e.g. [16] for a survey, and [5, 11, 28] for more recent results). Woeginger [28] proved that the general case of $1|prec|\sum w_j C_j$ is not harder to approximate than some fairly restricted special cases, among them for example the case of bipartite precedence constraints where all jobs on the first partition class have processing time 1 and weight 0, and all jobs on the second partition class have weight 1 and processing time 0.

For the general version of $1|prec|\sum w_j C_j$, closing the approximability gap is considered an outstanding open problem in scheduling theory (see e.g. [23]). While currently no inapproximability result is known (other than the problem does not admit a fully polynomial time approximation scheme), there are several polynomial time 2-approximation algorithms. Pisaruk [20] claims to have obtained the first of such 2-approximation algorithms. Schulz [22] and Hall, Schulz, Shmoys & Wein [9] gave 2-approximation algorithms by using a linear programming relaxation in completion time variables. Chudak & Hochbaum [4] gave another algorithm based on a relaxation of the linear program studied by Potts [21]. Independently, Chekuri & Motwani [3] and Margot, Queyranne & Wang [15], provided identical, extremely simple 2-approximation algorithms based on Sidney's decomposition theorem [24] from 1975.

A *Sidney decomposition* partitions the set N of jobs into sets $S_1, S_2, \ldots, S_k$ by using a generalization of Smith's rule [25], such that there exists an optimal schedule where jobs from S_i are processed before jobs from S_{i+1}, for any $i = 1, \ldots, k-1$. Lawler [12] showed that a Sidney decomposition can be computed in polynomial time by performing a sequence of min-cut computations. Chekuri & Motwani [3] and Margot, Queyranne & Wang [15] actually proved that every schedule that complies with a Sidney decomposition is a 2-approximate solution. Correa & Schulz [5] subsequently showed that all known 2-approximation algorithms follow a Sidney decomposition, and therefore belong to the class of algorithms described by Chekuri & Motwani [3] and Margot, Queyranne & Wang [15].

This result is rather demoralizing, as it shows that despite of many years of active research, all the approximation algorithms rely on the Sidney decomposition dating back to 1975. It should be emphasized that the Sidney decomposition does not impose any ordering among the jobs within a set S_i, and any ordering will do just fine for a 2-approximation. This shows that we basically have no clue how to order the jobs within the sets S_i.

The current state of $1|prec|\sum w_jC_j$, in terms of approximation, is very similar to one of the most famous and best studied NP-hard problems: the vertex cover problem (see [19] for a survey). Despite considerable efforts, the best known approximation algorithm still has a ratio of $2 - o(1)$. Improving this ratio is generally considered one of the most outstanding open problems in theoretical computer science. Hochbaum [10] conjectured that it is not possible to obtain a better factor.

In 1973, Nemhauser & Trotter [17, 18] used the following integer program to model the minimum vertex cover problem in a weighted graph (V, E) with weights w_i on the vertices.

$$\begin{array}{lll} \text{[VC-IP]} \qquad \min & \displaystyle\sum_{i\in V} w_i x_i & \\ \text{s.t.} & x_i + x_j \geq 1 & \{i,j\} \in E \\ & x_i \in \{0,1\} & i \in V \end{array}$$

They also studied the linear relaxation [VC-LP] of [VC-IP], and proved that any basic feasible solution for [VC-LP] is *half-integral*, that is $x_i \in \{0, \frac{1}{2}, 1\}$ for all $i \in V$. Moreover, they showed [18] that those variables which assume binary values in an optimal solution for [VC-LP] retain the same value in an optimal solution for [VC-IP]. This is known as the *persistency property* of vertex cover, and a solution is said to *comply* with the persistency property if it retains the binary values of an optimal solution for [VC-LP]. Hochbaum [10] pointed out that any feasible solution that complies with the persistency property is a 2 approximate solution.

2 Results and Implications

The dominant role the Sidney decomposition plays in $1|prec|\sum w_jC_j$ seems to be very well reflected by the persistency property for the vertex cover problem. It was often suspected that there is a strong relationship between vertex cover and $1|prec|\sum w_jC_j$ [23].

In this paper we show that this speculation is justified. We hope that this result will be an important step towards a proof that both problems are equivalent in terms of approximability. Proving this would give a more or less satisfactory answer to the ninth problem of the famous ten open problems in scheduling theory [23]. In general terms we prove the following result.

Theorem 1. *$1|prec|\sum w_jC_j$ is a special case of the vertex cover problem.*

Given that many people have worked hard on improving the various 2-approximation algorithms of $1|prec|\sum w_jC_j$, it seems likely that $1|prec|\sum w_jC_j$ is as hard to approximate as vertex cover. Certainly people working on approximation algorithms for vertex cover should try their luck at $1|prec|\sum w_jC_j$ first. Concerning inapproximability, APX-hardness of $1|prec|\sum w_jC_j$ is still not established. Maybe the techniques used for vertex cover will prove helpful here.

Theorem 1 is proved by showing that an optimal solution for Potts' integer program ([P-IP]) is also optimal for the integer program of Chudak & Hochbaum ([CH-IP]). This solves an open problem posted in [4] whose answer was conjectured in [5]. Pott's IP is known to model $1|prec|\sum w_jC_j$ correctly, whereas [CH-IP] is a relaxation of [P-IP] obtained by removing a big chunk of the conditions from [P-IP]. Theorem 1 then follows from the equivalence of [CH-IP] and the integer program [CS-IP] of Correa & Schulz [5], which is a special case of [VC-IP]. We prove the same result also for the linear relaxations of the three IPs, which are subsequently denoted by [P-LP], [CH-LP], and [CS-LP].

Apart from the long term consequences of our result, there are a few direct ones. Since $1|prec|\sum w_jC_j$ is a special case of the vertex cover problem, any approximation algorithm for vertex cover translates into an approximation algorithm for $1|prec|\sum w_jC_j$ of the same approximation guarantee.

More importantly, our result considerably increases the class of instances that can be solved optimally in polynomial time to the class of two-dimensional partial orders. A partial order (N, P) has dimension two if P can be described as the intersection of two total orders of N (see [16] for a survey).

In [5] it is proved that the vertex cover graph associated with [CS-LP] is bipartite if and only if the precedence constraints are of dimension two. This means that every basic feasible solution to [CS-LP] is integral in this case. They also give a 3/2-approximation algorithm for the two-dimensional case, which improves on a previous result [11].

In this paper we show that any integral solution to [CH-LP] can be converted into a feasible solution for [P-IP] without deteriorating the objective function value. Together with a result of [5], this implies that instances with two-dimensional precedence constraints are solvable in polynomial time. We emphasize that series-parallel partial orders have dimension at most two, but the class of two-dimensional partial orders is substantially larger [2]. Thus, the polynomial-time solvability of the two-dimensional case considerably extends Lawler's result [12] from 1978 for series-parallel orders.

3 Preliminaries

To simplify notation, we implicitly assume hereafter that tuples and sets of jobs have no multiplicity. Therefore, $(a_1, a_2, \ldots, a_k) \in N^k$ and $\{b_1, b_2, \ldots, b_k\} \subseteq N$ denote a tuple and a set, respectively, with k distinct elements.

In the following, we introduce several linear programming formulations and relaxations of $1|prec|\sum w_jC_j$ using linear ordering variables δ_{ij}. The variable δ_{ij} has value 1 if job i precedes job j in the corresponding schedule, and 0 otherwise. The first formulation using linear ordering variables is due to Potts [21], and it can be stated as follows.

$$\text{[P-IP]} \qquad \min \quad \sum_{j\in N} p_j w_j + \sum_{(i,j)\in N^2} \delta_{ij} p_i w_j \tag{1}$$

$$\text{s.t.} \qquad \delta_{ij} + \delta_{ji} = 1 \qquad \{i,j\} \subseteq N \tag{2}$$

$$\delta_{ij} = 1 \qquad (i,j) \in P \tag{3}$$

$$\delta_{ij} + \delta_{jk} + \delta_{ki} \leq 2 \qquad (i,j,k) \in N^3 \tag{4}$$

$$\delta_{ij} \in \{0,1\} \qquad (i,j) \in N^2 \tag{5}$$

Constraint (2) ensures that either job i is scheduled before j or viceversa. If job i is constrained to precede j in the partial order P, then this is seized by Constraint (3). The set of Constraints (4) is used to capture the transitivity of the ordering relations (i.e., if i is scheduled before j and j before k, then i is scheduled before k). It is easy to see that [P-IP] is indeed a complete formulation of the problem [21].

Chudak & Hochbaum [4] suggested to study the following relaxation of [P-IP]:

$$\text{[CH-IP]} \qquad \min\ (1) \quad \text{s.t.}\ (2), (3), (5)$$

$$\delta_{jk} + \delta_{ki} \leq 1 \qquad (i,j) \in P, \{i,j,k\} \subseteq N \tag{6}$$

In [CH-IP], Constraints (4) are replaced by Constraints (6). These inequalities correspond in general to a proper subset of (4), since only those transitivity Constraints (4) for which two of the participating jobs are already related to each other by a precedence constraint are kept. If the integrality Constraints (5) are relaxed and replaced by

$$\delta_{ij} \geq 0 \qquad (i,j) \in N^2, \tag{7}$$

then we will refer to the linear relaxations of [P-IP] and [CH-IP] as [P-LP] and [CH-LP], respectively. Chudak & Hochbaum's formulations lead to two natural open questions, first raised by Chudak & Hochbaum [4] and whose answers were conjectured by Correa & Schulz [5]:

Conjecture 1. [5] An optimal solution to [P-IP] is optimal for [CH-IP] as well.

Conjecture 2. [5] An optimal solution to [P-LP] is optimal for [CH-LP] as well.

The correctness of these conjectures has several important consequences, the most prominent being that the addressed problem can be seen as a special case of the vertex cover problem. Indeed, Correa & Schulz proposed the following relaxation of [P-IP] that can be interpreted as a vertex cover problem [5]:

$$
\begin{aligned}
\text{[CS-IP]} \qquad \min \quad & \sum_{j \in N} p_j w_j + \sum_{(i,j) \in N^2} \delta_{ij} p_i w_j & \\
\text{s.t.} \quad & \delta_{ij} + \delta_{ji} \geq 1 & \{i,j\} \subseteq N \\
& \delta_{ik} + \delta_{kj} \geq 1 & (i,j) \in P, \{i,j,k\} \subseteq N \\
& \delta_{i\ell} + \delta_{kj} \geq 1 & (i,j),(k,\ell) \in P, \{i,j,k,\ell\} \subseteq N \\
& \delta_{ij} = 1, \delta_{ji} = 0 & (i,j) \in P \\
& \delta_{ij} \in \{0,1\} & (i,j) \in N^2
\end{aligned}
$$

As usual, let us denote by [CS-LP] the linear relaxation of [CS-IP]. The following result is implied in [5].

Theorem 2. *[5] The optimal solutions to [CH-LP] and [CS-LP] coincide. Moreover, any feasible solution to [CS-LP] can be transformed in $O(n^2)$ time into a feasible solution to [CH-LP] without increasing the objective value. Both statements are true for [CH-IP] and [CS-IP] as well.*

As [CS-IP] represents an instance of the vertex cover problem, it follows from the work of Nemhauser & Trotter [17, 18] that [CS-LP] is half-integral, and that an optimal solution can be obtained via a single min-cut computation. Hence, the same holds for [CH-LP].

Theorem 3. *[4, 5] Linear programs [CH-LP] and [CS-LP] are half-integral.*

In order to unify the IP and the LP versions of the conjectures, the following parameterized setting is considered throughout this paper. We introduce yet another version of [P] and [CH]. For a parameter $\Delta = 1/q$ with $q \in \mathbb{N}$, let [CH-Δ] and [P-Δ] be equal to [CH-LP] and [P-LP], respectively, but with the additional constraint that all δ_{ij} are multiples of Δ. For Conjecture 1, it is obvious that [CH-IP] is equivalent to [CH-1]. The same holds for [P-IP] and [P-1]. As far as Conjecture 2 is concerned, Theorem 3 implies that any optimal solution for [CH-$\frac{1}{2}$] is optimal for [CH-LP] as well. Also, any feasible solution for [P-$\frac{1}{2}$] is feasible for [P-LP].

4 Main Theorem and Proof Overview

Our main theorem can be stated as follows.

Theorem 4. *Any feasible solution for [CH-Δ] can be turned into a feasible solution for [P-Δ] in $O\left(n^3/\Delta^2\right)$ time without increasing the objective value.*

Theorem 4 represents the missing link between problem $1|prec|\sum w_j C_j$ and vertex cover. With Theorem 4 in place, the claim of Theorem 1 directly follows by using Theorem 2. Moreover, both Conjecture 1 and 2 are proved to be true as corollaries. The proof of Theorem 4 is given in the following.

Proof overview. Let vector $\delta = \left(\delta_{ij} : (i,j) \in N^2\right)$ denote a feasible solution to [CH-Δ] throughout the paper. For any $(i,j,k) \in N^3$, let $\langle ijk \rangle$ denote the set $\{(i,j),(j,k),(k,i)\}$. Let us call $\langle ijk \rangle$ an *oriented 3-cycle*. Moreover, let $\mathcal{C}$ be the set of all oriented 3-cycles of the set N.[1] The following values are used to measure the infeasibility of δ for [P-Δ].

$$\alpha_{\langle ijk \rangle} := \max\left(0, \delta_{ij} + \delta_{jk} + \delta_{ki} - 2\right) \qquad \text{for all } \langle ijk \rangle \in \mathcal{C} \tag{8}$$

Observe that δ is feasible for [P-Δ] if and only if $\alpha_{\langle ijk \rangle} = 0$ for all $\langle ijk \rangle \in \mathcal{C}$, otherwise the transitivity Constraint (4) is violated. Let α be the total sum of all $\alpha_{\langle ijk \rangle}$ values.

$$\alpha := \sum_{\langle ijk \rangle \in \mathcal{C}} \alpha_{\langle ijk \rangle} \tag{9}$$

Let us call α the *total infeasibility* of solution δ. For any job k and $1 \leq s \leq 1/\Delta$, we also define the following set of job pairs that together with k violate Constraint (4).

$$\mathcal{B}_s^{(k)} := \left\{(i,j) : \alpha_{\langle ijk \rangle} \geq s\Delta\right\} \qquad \text{for all } k \in N \text{ and } 1 \leq s \leq 1/\Delta \tag{10}$$

These sets will be used to alter a feasible solution δ towards [P-Δ] feasibility, as explained in the following. Note that δ is feasible for [P-Δ] if and only if all $\mathcal{B}_s^{(k)}$ are empty. Let $\mathfrak{B} := \left\{\mathcal{B}_s^{(k)} : k \in N \text{ and } 1 \leq s \leq 1/\Delta\right\}$. For any set $\mathcal{B} \in \mathfrak{B}$, consider the vector $\delta^{\mathcal{B}}$ defined as:

$$\delta_{ij}^{\mathcal{B}} := \begin{cases} \delta_{ij} - \Delta \text{ if } (i,j) \in \mathcal{B}; \\ \delta_{ij} + \Delta \text{ if } (j,i) \in \mathcal{B}; \\ \delta_{ij} \qquad \text{otherwise;} \end{cases} \qquad \text{for all } (i,j) \in N^2. \tag{11}$$

We will show later in Lemma 4 that it cannot happen that $(i,j) \in \mathcal{B}$ and $(j,i) \in \mathcal{B}$ at the same time, therefore $\delta^{\mathcal{B}}$ is well defined. For any $\langle ijk \rangle \in \mathcal{C}$, let $\alpha_{\langle ijk \rangle}^{\mathcal{B}}$ and $\alpha^{\mathcal{B}}$ be the values defined in (8) and in (9), respectively, for $\delta^{\mathcal{B}}$.

The following lemma plays a fundamental role in the proof of Theorem 4. It asserts that when δ is not feasible for [P-Δ], it is always possible to choose a set $\mathcal{B} \in \mathfrak{B}$ such that $\delta^{\mathcal{B}}$ has a lower total infeasibility and a not larger objective value.

Lemma 1. *Let δ be a feasible solution to [CH-Δ], which is not feasible for [P-Δ]. Then*

(a) Vector $\delta^{\mathcal{B}}$ is a feasible solution to [CH-Δ] for all $\mathcal{B} \in \mathfrak{B}$.
(b) There exists a nonempty set $\mathcal{B} \in \mathfrak{B}$ such that the objective value of $\delta^{\mathcal{B}}$ is not larger than the objective value of δ.
(c) For any $\mathcal{B} \in \mathfrak{B}$, we have $\alpha^{\mathcal{B}} \leq \alpha - |\mathcal{B}|\Delta$.

[1] Note that $\langle ijk \rangle = \langle jki \rangle = \langle kij \rangle$ and $\langle jik \rangle = \langle kji \rangle = \langle ikj \rangle$, but $\langle ijk \rangle \neq \langle jik \rangle$. In contrast to $\langle ijk \rangle \in \mathcal{C}$, any reordering of the jobs in $(i,j,k) \in N^3$ results in a different triple.

By using Lemma 1, the proof of Theorem 4 becomes straightforward.

Proof of Theorem 4. The parts (a) and (b) of Lemma 1 ensure that among all the solutions $\delta^{\mathcal{B}}$, with $\mathcal{B} \in \mathfrak{B}$ and $\mathcal{B} \neq \emptyset$, there is one whose objective value is not larger than δ.

What is more, part (c) even ensures that the total infeasibility value α decreases by $|\mathcal{B}|\Delta$ when moving from δ to $\delta^{\mathcal{B}}$. Since $|\mathcal{B}|\Delta \geq \Delta$ and $\alpha_{\langle ijk \rangle} \geq 0$, for any $\langle ijk \rangle \in \mathcal{C}$, repeating this transformation will eventually lead to a solution for which (9) evaluates to zero, which means that it is feasible for [P-Δ]. It is not too hard to prove that this task can be accomplished in $O\left(n^3/\Delta^2\right)$ time. ■

5 A Few Useful Properties

In the following, we provide two properties of the α-values that will prove useful in the proof of Lemma 1.

Lemma 2. *For any $\langle ijk \rangle, \langle j\ell k \rangle \in \mathcal{C}$, if $(i, \ell) \in P$ or $i = \ell$ then*

$$\min\left(\alpha_{\langle ijk \rangle}, \alpha_{\langle j\ell k \rangle}\right) = 0.$$

Proof. The proof is by contradiction. Assume $\alpha_{\langle ijk \rangle} > 0$ and $\alpha_{\langle j\ell k \rangle} > 0$. Then

$$\delta_{ij} + \delta_{jk} + \delta_{ki} > 2 \quad \text{and} \quad \delta_{j\ell} + \delta_{\ell k} + \delta_{kj} > 2.$$

By adding up the previous two inequalities, and using Constraint (2), we obtain

$$\delta_{ij} + \delta_{j\ell} + \delta_{\ell k} + \delta_{ki} > 3,$$

which is impossible since by Constraints (2), (6) and (7), we have $\delta_{ij} + \delta_{j\ell} \leq 2$ and $\delta_{\ell k} + \delta_{ki} \leq 1$.

Lemma 3. *For any $\langle ijk \rangle, \langle \ell jk \rangle \in \mathcal{C}$ with $(i, \ell) \in P$, if $\delta_{ij} = \delta_{\ell j}$ (or equivalently $\delta_{ji} = \delta_{j\ell}$ by Constraint (2)) then $\alpha_{\langle ijk \rangle} \leq \alpha_{\langle \ell jk \rangle}$ and $\alpha_{\langle jik \rangle} \geq \alpha_{\langle j\ell k \rangle}$.*

Proof. By Constraints (2) and (6) we have $\delta_{k\ell} \geq \delta_{ki}$ and $\delta_{ik} \geq \delta_{\ell k}$, that, by the assumptions (i.e., $\delta_{ij} = \delta_{\ell j}$ and $\delta_{ji} = \delta_{j\ell}$), imply

$$\delta_{ij} + \delta_{jk} + \delta_{ki} \leq \delta_{\ell j} + \delta_{jk} + \delta_{k\ell},$$
$$\delta_{ji} + \delta_{kj} + \delta_{ik} \geq \delta_{j\ell} + \delta_{kj} + \delta_{\ell k}.$$

The claim follows by (8).

6 Proof of Lemma 1

Lemma 4. *For any $\{i, j\} \subseteq N$ and $\mathcal{B} \in \mathfrak{B}$, at most one pair between (i, j) and (j, i) belongs to set $\mathcal{B}$.*

Proof. Consider any $\mathcal{B}_s^{(k)} \in \mathfrak{B}$. For both (i, j) and (j, i) to be in $\mathcal{B}_s^{(k)}$ we need $\alpha_{\langle ijk \rangle} > 0$ and $\alpha_{\langle jik \rangle} > 0$, which cannot happen by Lemma 2.

Proof of Lemma 1(a). To prove that claim, we fix an arbitrary set $\mathcal{B}_s^{(k)} \in \mathfrak{B}$. To ease notation, we will often write $\mathcal{B}$ instead of $\mathcal{B}_s^{(k)}$. The claim follows by showing that the solution $\delta^{\mathcal{B}}$ satisfies Constraints (2), (3), (6) and (7), and that all $\delta_{ij}^{\mathcal{B}}$ are multiples of Δ. The latter directly follows from the feasibility of δ and the definition (11) of $\delta^{\mathcal{B}}$. Constraints (3) hold since for $(i,j) \in P$, we have $\alpha_{\langle ijk \rangle} = 0$ and $\alpha_{\langle jik \rangle} = 0$ and therefore neither (i,j) nor (j,i) will be part of the set $\mathcal{B}_s^{(k)}$, which ensures $\delta_{ij}^{\mathcal{B}} = \delta_{ij} = 1$.

Concerning Constraints (2), it follows from Lemma 4 and the definition of $\delta^{\mathcal{B}}$ that the following identity holds, which in turn ensures that $\delta^{\mathcal{B}}$ satisfies Constraints (2).

$$\delta_{ij}^{\mathcal{B}} + \delta_{ji}^{\mathcal{B}} = \delta_{ij} + \delta_{ji}$$

Regarding Constraints (6) we distinguish the two complementary cases. In the first case, we assume $\delta_{\ell j} + \delta_{ji} = 1$. We then obviously have $\delta_{ij} = \delta_{\ell j}$ and $\delta_{ji} = \delta_{j\ell}$ by Constraints (2). By definition of $\delta^{\mathcal{B}}$, in order to violate the constraint, we need at least one of (i,j) and (j,ℓ) to be in $\mathcal{B}_s^{(k)}$. If $(i,j) \in \mathcal{B}_s^{(k)}$ (and therefore $(j,i) \notin \mathcal{B}_s^{(k)}$ by Lemma 4), then we have $\alpha_{\langle ijk \rangle} \leq \alpha_{\langle \ell jk \rangle}$ by Lemma 3, that implies $(\ell,j) \in \mathcal{B}_s^{(k)}$ (and $(j,\ell) \notin \mathcal{B}_s^{(k)}$). Similarly, if $(j,\ell) \in \mathcal{B}_s^{(k)}$ (and therefore $(\ell,j) \notin \mathcal{B}_s^{(k)}$), then by Lemma 3 we have $\alpha_{\langle jik \rangle} \geq \alpha_{\langle j\ell k \rangle}$, that implies $(j,i) \in \mathcal{B}_s^{(k)}$ (and $(i,j) \notin \mathcal{B}_s^{(k)}$). Hence in the case $\delta_{\ell j} + \delta_{ji} = 1$, Constraints (6) are satisfied since we have $\delta_{ij}^{\mathcal{B}} + \delta_{j\ell}^{\mathcal{B}} \leq \delta_{ij} + \delta_{j\ell}$.

In the second case we assume $\delta_{\ell j} + \delta_{ji} < 1$ and argue as follows. If $(i,j) \in \mathcal{B}_s^{(k)}$ then $\alpha_{\langle ijk \rangle} > 0$, and by Lemma 2 it is $\alpha_{\langle j\ell k \rangle} = 0$, which implies $(j,\ell) \notin \mathcal{B}_s^{(k)}$. By these arguments, it is easy to see that $\delta_{\ell j}^{\mathcal{B}} + \delta_{ji}^{\mathcal{B}} \leq \delta_{\ell j} + \delta_{ji} + \Delta \leq 1$. This completes to proof for Constraints (6).

Finally, consider the nonnegativity Constraint (7). Note that $(i,j) \in \mathcal{B}_s^{(k)}$ implies $\alpha_{\langle ijk \rangle} \geq \Delta$, by the feasibility of δ. This in turn means $\delta_{ij} \geq \Delta$. The latter ensures that any $\delta_{ij}^{\mathcal{B}}$ satisfies Constraint (7). ∎

Lemma 5. *(without proof)*

$$\sum_{(i,j,k) \in N^3} \alpha_{\langle ijk \rangle} \cdot p_k p_i w_j = \sum_{(i,j,k) \in N^3} \alpha_{\langle ijk \rangle} \cdot p_k p_j w_i$$

Proof of Lemma 1(b). We first observe that if we decrease the value of any δ_{ij} by Δ and increase δ_{ji} by Δ, then the difference between the new objective value and the previous one is equal to $\Delta \cdot (p_j w_i - p_i w_j)$. Now, for any $\mathcal{B} \in \mathfrak{B}$, let $V(\mathcal{B})$ be defined as follows.

$$V(\mathcal{B}) := \sum_{(i,j) \in \mathcal{B}} p_j w_i - \sum_{(i,j) \in \mathcal{B}} p_i w_j \tag{12}$$

If we have a look at Transformation (11), the difference between the objective values of $\delta^{\mathcal{B}}$ and δ, respectively, is equal to $\Delta \cdot V(\mathcal{B})$.

If solution δ is not feasible for [P-Δ], what we have to show is that there always exists a set $\mathcal{B} \neq \emptyset$ (with $\mathcal{B} \in \mathfrak{B}$) for which (12) is non-positive. We prove this by contradiction.

Let us first define the following indicator function.

$$\mathcal{T}(S) := \begin{cases} 1 \text{ if the statement } S \text{ is true;} \\ 0 \text{ otherwise.} \end{cases}$$

Recall that by the definition of $\mathcal{B}_s^{(k)}$ in (10), it holds $(i,j) \in \mathcal{B}_s^{(k)}$ if and only if $\alpha_{\langle ijk \rangle} \geq s\Delta$. This allows to write

$$\alpha_{\langle ijk \rangle} = \Delta \cdot \sum_{s=1}^{1/\Delta} \mathcal{T}\left((i,j) \in \mathcal{B}_s^{(k)}\right).$$

Now, assume that for all $\mathcal{B} \neq \emptyset$ the value $V(\mathcal{B})$ is positive, i.e., the following inequality holds.

$$\sum_{(i,j)\in\mathcal{B}} p_i w_j < \sum_{(i,j)\in\mathcal{B}} p_j w_i \qquad \text{for all nonempty } \mathcal{B} \in \mathfrak{B} \tag{13}$$

Using Assumption (13), we can conclude

$$\begin{aligned}
\sum_{(i,j,k)\in N^3} \alpha_{\langle ijk \rangle} \cdot p_k p_i w_j &= \sum_{(i,j,k)\in N^3} p_k \left(\Delta \cdot \sum_{s=1}^{1/\Delta} \mathcal{T}\left((i,j) \in \mathcal{B}_s^{(k)}\right) \right) p_i w_j \\
&= \sum_{k\in N} p_k \Delta \sum_{s=1}^{1/\Delta} \left(\sum_{(i,j)\in\mathcal{B}_s^{(k)}} p_i w_j \right) \\
&< \sum_{k\in N} p_k \Delta \sum_{s=1}^{1/\Delta} \left(\sum_{(i,j)\in\mathcal{B}_s^{(k)}} p_j w_i \right) \qquad (14) \\
&= \sum_{(i,j,k)\in N^3} p_k \left(\Delta \cdot \sum_{s=1}^{1/\Delta} \mathcal{T}\left((i,j) \in \mathcal{B}_s^{(k)}\right) \right) p_j w_i \\
&= \sum_{(i,j,k)\in N^3} \alpha_{\langle ijk \rangle} \cdot p_k p_j w_i,
\end{aligned}$$

which clearly contradicts Lemma 5. The interesting line here is the strict Inequality (14). If we assume $p_x > 0$ for all jobs $x \in N$, it just follows from Assumption (13) and the obvious fact that $V(\mathcal{B}) = 0$ when $\mathcal{B}$ is an empty set.

To prove it also for the case with zero processing times, we need to show that there exists a job $x \in N$ and $s \in \{1, \ldots, 1/\Delta\}$ with $p_x > 0$ and $\mathcal{B}_s^{(x)} \neq \emptyset$. With this aim, consider any nonempty set $\mathcal{B}_1^{(k)}$. Note that the assumption that δ is not feasible for [P-Δ] guarantees the existence of set $\mathcal{B}_1^{(k)} \neq \emptyset$. Because of

Assumption (13), there must be at least one $(i,j) \in \mathcal{B}_1^{(k)}$ with $p_i w_j < p_j w_i$, which implies $p_j > 0$. It also follows from $(i,j) \in \mathcal{B}_1^{(k)}$ that $(k,i) \in \mathcal{B}_1^{(j)}$. Hence, job j meets our requirements since $p_j > 0$ and $\mathcal{B}_1^{(j)} \neq \emptyset$. ∎

The proof of Lemma 1(c) is omitted due to lack of space.

7 Open Problems

It would be interesting to investigate further connections between the vertex cover problem and $1|prec|\sum w_j C_j$. It is known that vertex cover cannot be approximated in polynomial time within a factor of $10\sqrt{5} - 21 \approx 1.36067$, unless P=NP (Dinur & Safra [6]). It would be nice to have a similar result for $1|prec|\sum w_j C_j$ or, as suggested in [23], to prove that a polynomial time ρ-approximation algorithm for $1|prec|\sum w_j C_j$ implies the existence of a polynomial time ρ-approximation algorithm for the vertex cover problem.

Acknowledgments. The authors thank Andreas Schulz and an anonymous referee for useful comments. The first author is supported by Nuffield Foundation Grant NAL32608. The second author is supported by Swiss National Science Foundation project 200021-104017/1, "Power Aware Computing", and by the Swiss National Science Foundation project 200020-109854, "Approximation Algorithms for Machine scheduling Through Theory and Experiments II". The second author would like to dedicate this work to Emma on the occasion of her birth.

References

1. Brenda S. Baker. Approximation algorithms for NP-complete problems on planar graphs. *Journal of the ACM*, 41(1):153–180, 1994.
2. K. A. Baker, P. C. Fishburn, and F. S. Roberts. Partial orders of dimension 2. *Networks*, 2:11–28, 1971.
3. C. Chekuri and R. Motwani. Precedence constrained scheduling to minimize sum of weighted completion times on a single machine. *Discrete Applied Mathematics*, 98(1-2):29–38, 1999.
4. F. A. Chudak and D. S. Hochbaum. A half-integral linear programming relaxation for scheduling precedence-constrained jobs on a single machine. *Operations Research Letters*, 25:199–204, 1999.
5. José R. Correa and Andreas S. Schulz. Single machine scheduling with precedence constraints. *Mathematics of Operations Research*, 30:1005–1021, 2005.
6. I. Dinur and S. Safra. On the hardness of approximating minimum vertex cover. *Ann. of Math. (2)*, 162(1):439–485, 2005.
7. Michel X. Goemans and David P. Williamson. Two-dimensional Gantt charts and a scheduling algorithm of Lawler. *SIAM J. Discrete Math.*, 13(3):281–294, 2000.
8. R. Graham, E. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. In *Annals of Discrete Mathematics*, volume 5, pages 287–326. North–Holland, 1979.

9. L. A. Hall, A. S. Schulz, D. B. Shmoys, and J. Wein. Scheduling to minimize average completion time: off-line and on-line algorithms. *Mathematics of Operations Research*, 22:513–544, 1997.
10. D. S. Hochbaum. Efficient bounds for the stable set, vertex cover and set packing problems. *Discrete Applied Mathematics*, 6:243–254, 1983.
11. Stavros G. Kolliopoulos and George Steiner. Partially-ordered knapsack and applications to scheduling. In *Proceedings of the 10th Annual European Symposium on Algorithms (ESA)*, pages 612–624, 2002.
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. E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys. Sequencing and scheduling: Algorithms and complexity. In S. C. Graves, A. H. G. Rinnooy Kan, and P. Zipkin, editors, *Handbooks in Operations Research and Management Science*, volume 4, pages 445–552. North-Holland, 1993.
14. J. K. Lenstra and A. H. G. Rinnooy Kan. The complexity of scheduling under precedence constraints. *Operations Research*, 26:22–35, 1978.
15. F. Margot, M. Queyranne, and Y. Wang. Decompositions, network flows and a precedence constrained single machine scheduling problem. *Operations Research*, 51(6):981–992, 2003.
16. R. H. Möhring. Computationally tractable classes of ordered sets. In I. Rival, editor, *Algorithms and Order*, pages 105–193. Kluwer Academic, 1989.
17. G. L. Nemhauser and L. E. Trotter. Properties of vertex packing and independence system polyhedra. *Mathematical Programming*, 6:48–61, 1973.
18. G. L. Nemhauser and L. E. Trotter. Vertex packings: Structural properties and algorithms. *Mathematical Programming*, 8:232–248, 1975.
19. V. T. Paschos. A survey of approximately optimal solutions to some covering and packing problems. *ACM Computing Surveys*, 29(2):171–209, 1997.
20. N. N. Pisaruk. A fully combinatorial 2-approximation algorithm for precedence-constrained scheduling a single machine to minimize average weighted completion time. *Discrete Applied Mathematics*, 131(3):655–663, 2003.
21. C. N. Potts. An algorithm for the single machine sequencing problem with precedence constraints. *Mathematical Programming Study*, 13:78–87, 1980.
22. Andreas S. Schulz. Scheduling to minimize total weighted completion time: Performance guarantees of LP-based heuristics and lower bounds. In *Proceedings of the 5th Conference on Integer Programming and Combinatorial Optimization (IPCO)*, pages 301–315, 1996.
23. P. Schuurman and G. J. Woeginger. Polynomial time approximation algorithms for machine scheduling: ten open problems. *Journal of Scheduling*, 2(5):203–213, 1999.
24. J. B. Sidney. Decomposition algorithms for single-machine sequencing with precedence relations and deferral costs. *Operations Research*, 23:283–298, 1975.
25. W. E. Smith. Various optimizers for single-stage production. *Naval Research Logistics Quarterly*, 3:59–66, 1956.
26. G. Steiner. Single machine scheduling with precedence constraints of dimension 2. *Mathematics of Operations Research*, 9(2):248–259, 1984.
27. W. T. Trotter. *Combinatorics and Partially Ordered Sets: Dimension Theory*. The John Hopkins University Press, Baltimore and London, 1992.
28. G. J. Woeginger. On the approximability of average completion time scheduling under precedence constraints. *Discrete Applied Mathematics*, 131(1):237–252, 2003.

Cooperative TSP

Amitai Armon, Adi Avidor, and Oded Schwartz

School of Computer Science,
Tel-Aviv University, Tel-Aviv 69978, Israel
{armon, adi, odedsc}@tau.ac.il

Abstract. In this paper we introduce and study cooperative variants of the Traveling Salesperson Problem. In these problems a salesperson has to make deliveries to customers who are willing to help in the process. Customer cooperativeness may be manifested in several modes: they may assist by approaching the salesperson, by reselling the goods they purchased to other customers, or by doing both.

Several objectives are of interest: minimizing the total distance traveled by all the participants, minimizing the maximal distance traveled by a participant and minimizing the total time until all the deliveries are made.

All the combinations of cooperation-modes and objective functions are considered, both in weighted undirected graphs and in Euclidean space. We show that most of the problems have a constant approximation algorithm, many of the others admit a PTAS, and a few are solvable in polynomial time. On the intractability side we provide NP-hardness proofs and inapproximability factors, some of which are tight.

1 Introduction

The Traveling Salesperson Problem (TSP) is a classical problem in combinatorial optimization, which has been studied extensively in many forms. Cooperative TSP is a set of variants of TSP in which the customers are allowed to move in order to assist the selling process. They may move in order to expedite the deliveries, and may also move after meeting the salesperson in order to help the distribution of the goods. For example, consider a secret message that has to be distributed to several people, but is only allowed to be passed in person. Every person who receives the message may then assist by passing it forward. We may want to devise a scheme for delivering the secret to all the recipients as fast as possible. A further illustration is the problem of an ice cream van vendor. The vendor wishes to sell ice cream to all children in town. The children are eager to cooperate, by approaching the van in order to buy ice cream. However, in contrast to the previous example, they are not interested in selling ice cream to others.

Formally, an instance of Cooperative TSP (cTSP) is a set of *agents* and a *salesperson*, located in a finite metric space or a Euclidean space. A solution is a synchronized series of move instructions to all *participants* (i.e., the salesperson

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 40–51, 2006.

and the agents), such that all the agents eventually receive the delivery. We next elaborate on the various cooperation modes and the cost of solutions.

Cooperation Modes. We consider three modes of cooperation. In the PURCHASE-COOPERATION mode the salesperson has to meet all agents, and the agents are allowed to move towards the salesperson. In the SALES-COOPERATION mode, each agent receiving a delivery becomes capable of making deliveries similarly to the salesperson. However, an agent is not allowed to move before receiving a delivery. In the FULL-COOPERATION mode, an agent may cooperate in both the purchase and sales phases. That is, an agent may move before receiving the delivery and may make deliveries after receiving it.

Goal Functions. Three objectives are considered for Cooperative TSP: minimizing the total length traversed by all participants (MIN-SUM), minimizing the maximal length traversed by a participant (MIN-MAX), and minimizing the total time until the sales process ends (MIN-MAKESPAN).

We consider Cooperative TSP in a fixed-dimension Euclidean space and in weighted undirected graphs (note that w.l.o.g, we may assume that the graph is complete and weights satisfy the triangle inequality). We consider the ROUNDTRIP versions, in which all participants are required to return to their initial location, and the PATH versions in which there is no such requirement.

Related Studies. TSP remains NP-hard even in the special planar variant. However, the latter variant has a PTAS [Aro98, Mit99]. When metric space is assumed, the Christofides [Chr76] approximation algorithm yields a $\frac{3}{2}$-approximation ratio and an inapproximability factor of $\frac{203}{202}$ was shown [EK01].

The Freeze-Tag Problem was first suggested and studied by Arkin *et al.* in [ABF$^+$02]. This problem arises in the context of swarm robotics: how to wake a set of slumbering robots, by having an already awake robot move to their locations. Once a robot is awake it can assist in waking up other slumbering robots. The objective is to have all robots awake as early as possible. In our terminology this is the PATH version of MIN-MAKESPAN SALES cTSP. Arkin *et al.* [ABF$^+$02] provided an NP-hardness proof, a PTAS for the Euclidean variant, and a constant approximation for some graph families. A series of studies followed (e.g., [SABM02, ABG$^+$03, KLS04]) culminating with an $O(\sqrt{\log n})$-approximation for the general weighted graph case [KLS04].

TSP with Neighborhoods is a proximity-related variant of TSP. In this problem each customer is willing to meet the salesperson anywhere within some neighborhood. The problem was first studied by Arkin *et al.* [AH94], followed by quite a few papers (e.g., [MM95, DM01, dBGK$^+$05, SS05, Mit06]). This problem seems quite related to PURCHASE cTSP, as in both customers are willing to approach the salesperson. However, in TSP with Neighborhoods the customers' travel is not counted in the goal function, while in Cooperative TSP their moves do cost, and are part of the optimization task.

Other Cooperative Multi-Agents Routing Problems. As noted in [ABF$^+$02], the Freeze-Tag Problem (and thus the Cooperative TSP problems) bears features of broadcasting, routing, scheduling and network design. The *minimum broadcast time*, the *multicast problem* and the *minimum gossip time problem* are all closely

Table 1. Summary of approximation factors vs. inapproximability ratios in graphs. The parameter ε stands for an arbitrarily small positive constant, or for a positive function that tends to zero as the input size increases. (1) is by [KLS04], (2) is by [ABF$^+$02].

		Purchase Cooperation		Sales Cooperation		Full Cooperation	
	Goal	Approx.	Inapprox.	Approx.	Inapprox.	Approx.	Inapprox.
path	Min-Sum	$2+\ln 3$	NP-hard	2	NP-hard	$2+\ln 3$	APX-hard
	Min-Max	PTAS	no FPTAS	3	$2-\varepsilon$	4	$2-\varepsilon$
	Min-Makespan	Polynomial		$O(\sqrt{\log n})$ [1]	$\frac{5}{3}-\varepsilon$ [2]	2	$2-\varepsilon$
roundtrip	Min-Sum	$\frac{3}{2}$	$\frac{203}{202}-\varepsilon$	$\frac{3}{2}$	$\frac{203}{202}-\varepsilon$	$\frac{3}{2}$	$\frac{203}{202}-\varepsilon$
	Min-Max	PTAS	no FPTAS	3	$\frac{3}{2}-\varepsilon$	2	$2-\varepsilon$
	Min-Makespan	Polynomial		$O(\sqrt{\log n})$	$\frac{5}{4}-\varepsilon$	2	$2-\varepsilon$

related to Cooperative TSP (see [HHL88] for a survey and [Rav94, BNGNS98] for approximation results). Controlling swarms of robots in order to perform a certain task, has also been studied in various algorithmic aspects, including environment exploration, robot formation, searching and recruitment (see [ABF$^+$02] for a list of relevant papers). Other researches are trying to confront similar scenarios, but with no central control, where each agent has to make decisions with limited knowledge regarding the environment and the other agents (for example, the problem of routing autonomous agents in wireless sensor network; and ants behavior inspired algorithms; see [ABF$^+$02] for a list of relevant papers).

As cTSP is a generalization of both the Freeze-Tag and the TSP with Neighborhoods problems, the algorithms (and intractability results) achieved for cTSP apply to similar scenarios, e.g, cooperative robots tasks (see for example [AH94, ABF$^+$02] for other relevant scenarios).

The Min-Max cost function is suitable, for example, when there is a bound on the energy that each robot is allowed to spend. The Min-Sum cost function, on the other hand, is relevant when travel costs of all robots are covered by a single entity, who is therefore interested in minimizing the sum of these costs.

Our Contribution: We consider all combinations of cooperation modes, goal functions, path / roundtrip and graph / Euclidean. See Table 1 for the results on graphs and Table 2 for the results in Euclidean space. We obtain constant approximations and hardness results for most of the problems, PTAS for many of the others and polynomial-time exact solutions for a few. On the intractability side we give NP-hardness and inapproximability factors for all the graph problems and for some of the Euclidean problems.

Paper Organization. From here on, by cTSP we mean the path (rather than the roundtrip) version of the corresponding Cooperative TSP problem, unless otherwise stated.

In Section 2 we present some of the results for Euclidean cTSP. Section 3 contains some of the results for cTSP problems on graphs. The proofs for the other results are either achieved using similar methods to those presented here, or are straightforward, and are omitted from this version due to space limitation.

Table 2. Summary of approximation factors in Euclidean space, for any fixed dimension. The parameter ε stands for an arbitrarily small positive constant, or for a positive function that tends to zero as the input size increases. (1) is by [ABF+02].

	Goal	Purchase Cooperation	Sales Cooperation	Full Cooperation
path	Min-Sum	PTAS	$\frac{5}{3}+\varepsilon$	$2+\varepsilon$
	Min-Max	PTAS	3	4
	Min-Makespan	Polynomial	PTAS [1]	PTAS
round trip	Min-Sum	PTAS	PTAS	PTAS
	Min-Max	PTAS	3	2
	Min-Makespan	Polynomial	PTAS	PTAS

2 Euclidean cTSP

This section presents some of the results obtained for Euclidean cTSP.

2.1 A PTAS for Min-Sum Purchase Euclidean-cTSP

We next provide a PTAS for Min-Sum Purchase Euclidean-cTSP. The algorithm and analysis below use Arora's technique for the PTAS of Euclidean TSP [Aro98]. Our algorithm differs from Arora's algorithm in that it has to consider *all* the agents' paths and not just the salesperson's path. We show how this can be done while keeping the dynamic programming polynomial. We show:

Theorem 1. Min-Sum Purchase Euclidean-cTSP *admits a PTAS.*

Note that the problem is NP-hard even for the planar case. This follows, since an instance of the planar TSP can be reduced to an instance of Min-Sum Purchase Euclidean-cTSP by simply replacing each costumer with three agents. This makes an instance where the salesperson is the only participant who moves.

We next describe the PTAS for the planar case. The extension to any fixed dimension is straightforward. Our terminology resembles the one of Arora [Aro98] and is given here for completeness.

Let $\varepsilon > 0$ be an arbitrary small constant. Denote by n the number of participants and by OPT the cost of the optimal solution. Let $L = 2^{3+\lceil 2 \log n \rceil}$. Without loss of generality, that all the participants are located inside the *bounding box* $[0, L/2]^2$ and that $OPT > L/4$.

Super-pixels. We call each square $[j, j+2] \times [j', j'+2]$, where $j, j' \in \{0, 2, 4, \ldots, L-2\}$, a *pixel*. We name the point $(j+1, j'+1)$ the *center of the pixel* $[j, j+2] \times [j', j'+2]$. For every $i = 0, \ldots, \log L - 1$, we call each square $[j, j + L/2^i] \times [j', j' + L/2^i]$, where $j, j' \in \{0, L/2^i, 2 \cdot L/2^i \ldots, L - L/2^i\}$, a *super-pixel* of level i. It is not hard to see that, without loss of generality, we may consider only instances for which all the participants are located at pixel centers.

An (a,b)**-shifting.** Let $0 \leq a,b < L/2$ be two **even** integers. For a set $A \subseteq [0,L/2]^2$ we define the (a,b)-*shift* of A to be the set $\{(x+a,y+b) \mid (x,y) \in A\}$.

Portals. Let $m \in [\frac{8\sqrt{2}\log L}{\varepsilon}, \frac{16\sqrt{2}\log L}{\varepsilon})$ be a power of 2. For each super-pixel we mark each one of its four boundaries with m equidistant points that we refer to as *portals*. In particular, the portals include the four corners of the super-pixel. Note that, as m is a power of 2, each portal of a super-pixel of level i is also a portal of a smaller super-pixel of level $i+1$, for $i = 0,\ldots,\log L - 2$.

Portals-limited-solutions. We define a *portals-limited-solution* as a solution that satisfies the following four conditions:

1. Each participant may cross the boundary of a super-pixel only at its portals.
2. The salesperson does not cross her own route except on portals, where she may visit at most twice.
3. A meeting between an agent and the salesperson occurs only at a pixel center.
4. If two (or more) agents happen to reside at a pixel, then they all travel to (or stay at) the pixel's center and cease to move.

Therefore, in a portals-limited-solution, the tour of each participant is a collection of segments which connect portals to portals, and centers of pixels to portals. Additionally, a meeting between an agent and the salesperson occurs only at a pixel center, and tours of two agents do not cross.

Lemma 1. *Let a,b be two even integers chosen uniformly at random from the set $\{0,2,\ldots,L/2-2\}$. Then, the expected cost of a minimal cost portals-limited-solution of the (a,b)-shifted instance, is at most $(1+\varepsilon)\cdot OPT$.*

Lemma 2. *A minimal cost portals-limited-solution can be found in time polynomial in n.*

The proof of Lemma 1 mainly follows arguments from the PTAS of Euclidean TSP and will appear in the full version of the paper. The PTAS enumerates over all $O(L^2)$ values of (a,b) pairs. For each pair it applies Lemma 2 to find a minimal cost portals-limited-solution. Finally, it outputs the cheapest solution found, which according to Lemma 1, must have a cost of at most $(1+\varepsilon)\cdot OPT$. Clearly, the $O(n^4)$ factor in running time, caused by the enumeration over all (a,b) pairs, can be avoided if only an *expected* $(1+\varepsilon)\cdot OPT$ cost is desired.

Proof. **(of Lemma 2)** We use dynamic programming to build a polynomial-size table. For each super-pixel, the table contains $6^{4m} = n^{O(1/\varepsilon)}$ entries. For each entry we store *portions* of some portals-limited-solutions (the portions of solutions limited to that super-pixel) together with their contribution to the overall cost.

The construction of the table is conducted in a bottom-up manner, starting from the pixels. A minimal value portals-limited-solution for the whole instance is obtained at the bounding box super-pixel.

The entries of the table for each super-pixel are represented by a list of $4m$ elements, one element for each portal of the super-pixel. Each element takes one of the following six values:

1. The salesperson enters the super-pixel at this portal
2. The salesperson leaves the super-pixel at this portal
3. The salesperson enters and leaves the super-pixel at this portal
4. One agent enters the super-pixel at this portal
5. One agent leaves the super-pixel at this portal
6. None of the participants uses this portal

Note that the conditions defining a portals-limited-solution guarantee that these six cases cover all possible tour portions induced by all portals-limited-solutions (here we use the fact that two agents do not happen to reach the same portal, as they start at pixel centers, their tours do not cross and they end up at pixel centers). Also note that not all the $4m$-size lists represent a valid portion of some portals-limited-solution. We use the term *valid-list* for a list that represents a collection of tours that can be extended to some portals-limited-solution. Clearly, there are at most $6^{4m} = n^{O(1/\varepsilon)}$ (valid-)lists. Finally, note that the salesperson's paths can intersect only at his entrance or exit points. Hence, given a valid-list, *pairings* of the participants' entrance and exit points can be found as in the algorithm of Arora [Aro98].

We now describe the construction in a bottom-up manner. Consider a pixel. Each valid-list of the pixel falls into one of the following three categories:

1. There is no agent in the pixel and the salesperson may visit the pixel one or more times.
2. There is one agent in the pixel. If the salesperson visits the pixel they meet at the pixel's center.
3. Two or more agents pass through the pixel. The salesperson also visits the pixel. In one of the visits she arrives at the center of the pixel. In this case, each agent travels along a straight line from a portal of the pixel to the center of the pixel. Alternatively, an agent's route may be an empty route if the agent is already located at the center of the pixel.

In each case, the computation of the cost for each valid-list of the pixel can be done in polynomial time.

We now turn to the computation of the table's entries for the super-pixels of level i, assuming all valid-lists of super-pixels of level $i+1$ were computed. Let S be a level i super-pixel and consider a list of $1, \ldots, 6$ values for its portals. The list already fixes the entrances and exits on the boundary of S. The super-pixel S contains four level $i+1$ super-pixels, which have four boundaries internal to S, with a total of at most $4m$ more portals. Each of these portals may be used in one out of the six ways, giving rise again to $n^{O(1/\varepsilon)}$ possibilities. The cost for each possibility can be computed by using the values for the four $i+1$ level super-pixels previously obtained. Thus, we can find the minimal cost that corresponds to each list in $O(n^{O(1/\varepsilon)})$ time.

For the top-level super-pixel (the bounding-box) we may only consider the list for which neither the salesperson nor an agent visit a portal. The last table update of level 0 produces the cost of a minimal portals-limited-solution.

Makespan-Sales PTAS

1. For each subset S of participants of size up to $3m^4$, which includes the salesperson and contains a representative from each non-empty pixel:
   (a) Find an optimal solution for S by conducting an exhaustive search.
   (b) In each non-empty pixel apply a constant-approximation to all original participants of the pixel, where the salesperson is a representative of the pixel.
   (c) Extend the partial solution of S to a solution for the original instance: when all the participants in S return to their pixels - simultaneously perform the solution found in step 1(b).
2. Return the minimal cost solution found

Fig. 1. A PTAS for the ROUNDTRIP version of MIN-MAKESPAN SALES EUCLIDEAN-CTSP. The parameter m is assumed to be $\lceil 1/\varepsilon \rceil$.

2.2 MIN-MAKESPAN EUCLIDEAN-CTSP

We next present a simple PTAS for the ROUNDTRIP version of MIN-MAKESPAN SALES EUCLIDEAN-CTSP. A PTAS for the corresponding FULL-COOPERATION problem can be obtained by similar means. We note that the corresponding PURCHASE problem is polynomial-time solvable, as there is always an optimal solution in which all participants meet at a single point.

The PTAS for the two dimensional case appears in Figure 1. The generalization to any fixed dimension is straightforward.

Theorem 2. *The* ROUNDTRIP *version of* MIN-MAKESPAN SALES EUCLIDEAN-CTSP *admits a PTAS. The running time of the PTAS is $O(n + f(\varepsilon))$, where $\varepsilon > 0$ is an arbitrarily small constant, $f(\varepsilon)$ depends only on ε, and n is the number of participants.*

A constant approximation algorithm for the PATH version of this problem appears in [ABF+02]. The solution found by their algorithm is also $O(1)$ times the diameter (the maximal distance between any two points) of the input. One can adapt this approximation to the ROUNDTRIP version by returning each participant to its origin. The cost of the resulting solution is at most twice the original solution. Since an optimal solution to the PATH version costs less than an optimal solution for the corresponding ROUNDTRIP version, this heuristic is a constant approximation for the ROUNDTRIP version.

We assume, w.l.o.g. that the instance lies inside $[0,1]^2$ and has an optimal cost of at least $1/2$. Let $m = \lceil 1/\varepsilon \rceil$. We divide the unit square $[0,1]^2$ into m^2 *pixels*. I.e., a *pixel* is a square of the form $[\frac{j}{m}, \frac{j+1}{m}] \times [\frac{j'}{m}, \frac{j'+1}{m}]$, where $j, j' = 0, 1, \ldots, m-1$. The PTAS for the SALES version relies on the next lemma:

Lemma 3. *Let I be an instance of n participants with an optimal makespan of OPT. Then, there exists an instance $S \subseteq I$ with at most $3m^4$ participants, in which each non-empty pixel in I is also non-empty in S and the optimal makespan of S is at most $(1 + O(\varepsilon))OPT$.*

Proof. We may assume, w.l.o.g., that no two participants in I are located at the same point and that no three participants lie on a straight line. Otherwise, we can perturb each participant's location by at most ε/n and obtain an instance with an optimal cost of at most $(1 + O(\varepsilon))OPT$.

Let π be an optimal solution to I. We define the *sales-tree* of π to be a directed graph in which the nodes are the locations of the participants and there is a directed edge from u to v if a participant traveled from u to v in π. Since no two participants are located at the same point and no three participants lie on a straight line the in-degree of every node is one and the out-degree is at most two. We prune the sales-tree of π by iteratively removing leaves: we remove a leaf u if there exists another node in the sales-tree which resides in the same pixel as u. At the end of the process we are left with at most $m^2/2$ leaves, and at most $m^2/2$ nodes of degree 3 (in-degree plus out-degree). Note that the makespan of an optimal solution for the new instance, denoted π_0, is at most OPT. We now further decrease the number of participants by pruning some of the degree-2 vertices. We call a maximal set of participants along a path in which all the nodes are of degree 2 a *chain*. Clearly, each chain ends with a degree 3 node or a leaf. Hence, there are at most m^2 chains. For each chain, and a pixel it intersects with, we intend to keep at most two nodes (participants). All the other nodes are removed from the chain. For a given pixel and a chain, the two participants that we keep are the first and the last (of this chain, inside the pixel) who receive the goods. We call such nodes a *beginner* node and an *ender* node, respectively. Note that, we are left with at most $2 \cdot m^2$ participants per chain, giving rise to at most $2m^4$ nodes of degree 2.

The new instance constructed, denoted S has at most $3m^4$ participants. We next show that

Claim. There exists a solution π_S for S of cost at most $OPT + O(\varepsilon)$.

Proof. Recall that π_0 (the optimal solution after pruning the leaves) is of cost at most OPT. We construct the solution π_S from π_0 as follows: each participant of a beginner node travels along the corresponding original chain until it reaches the corresponding ender node, and then travels back to its starting location. All other participants travel along the same route they travel in π_0. Since all the non-beginner participants travel the way they do in π_0, they arrive to their original location by the time OPT. Beginner participants may be delayed by the time it takes to travel from the corresponding ender node back to their original location. This is at most the time it takes to cross a pixel which is at most $\sqrt{2}\varepsilon$. Thus, the cost of an optimal solution to S is at most $(1 + O(\varepsilon))OPT$.

The correctness of the PTAS algorithm for the ROUNDTRIP version of MIN-MAKESPAN SALES EUCLIDEAN-CTSP can now be deduced:

Proof. (of Theorem 2) Let π be an optimal solution for the instance I and let $S \subseteq I$ be an instance that satisfies the condition of Lemma 3. Clearly, the subset of participants S is included in the enumeration of our algorithm. The cost of an optimal solution to S, which is $(1 + \varepsilon)OPT$ is computed at stage 2(b) of

Hop-visit$(G(V,E),v)$:

1. Let $G' = (V', E')$ be a weighted complete graph, where $V' \subseteq V$ is the set of vertices which contain participants, and the edge-weights are the corresponding distances in G.
2. Compute a minimum-spanning-tree T of G', rooted at the salesperson's vertex v.
3. The salesperson visits an arbitrary child, and doesn't move any further.
4. When an agent receives a delivery:
   (a) If the agent has a *sibling* in T who hasn't received the delivery, then the agent visits such a sibling and one of that sibling's children.
   (b) Otherwise, the agent visits a child of the sibling which was visited first (a child of the "eldest" sibling of that agent), if such a child exists.

Fig. 2. A 3-approximation algorithm for Min-Max Sales cTSP

our algorithm. The additional cost produced at stage 2(c) is at most a constant times the diameter of the pixel, which is $O(\varepsilon)$. Note that this is an additive $O(\varepsilon)$ increase of the makespan, as after all the participants in S return to their pixels the delivery to the other participants is done in parallel. Hence, the total cost of the solution produced by our algorithm is at most $(1+O(\varepsilon))$ times the cost of π.

Finally, note that there are less than $O(n^{O(m^4)}) = O(n^{O(1/\varepsilon^4)})$ sets of participants to enumerate on. For each such subset S, a solution is a sequence of at most $2|S|-1$ moves. This follows as in each move either a participant receives the delivery or a participant returns to its original location. In any case, each move can be represented as a pair of two of the original input locations. Hence, for a given subset $|S|$, the number of solutions the algorithm enumerates on is at most

$$\binom{|S|}{2}^{2|S|-1} = O\left(\binom{m^4}{2}^{O(m^4)}\right) = \left(\frac{1}{\varepsilon}\right)^{O(1/\varepsilon^4)}.$$

Thus, the algorithm is a PTAS and runs in time $O\left(n + \left(\frac{1}{\varepsilon}\right)^{O(\frac{1}{\varepsilon^4})}\right)$.

3 cTSP in Graphs

In this section we present some of the algorithmic results for cTSP in graphs.

3.1 Min-Max Sales cTSP

We present a simple constant approximation algorithm for Min-Max Sales cTSP in Figure 2.

Theorem 3. Min-Max Sales cTSP *is* 3*-approximable.*

Proof. We prove that Algorithm **Hop-visit** is a 3-approximation algorithm for this problem. Clearly, all the agents are visited. Each participant traverses at

Coarse-Path$(G(V,E),v,\varepsilon)$:

1. For each ordered subset $V' \subseteq V$ of size $1+\lfloor 1/\varepsilon \rfloor$ or less, which starts with v.
   (a) For each $u \notin V'$ that contains an agent, find its distance to a closest vertex in V'. Denote the maximal distance found by *MaxDist(V')*.
   (b) Compute the sum of distances between pairs of consecutive vertices in V', and denote it by *Length(V')*.
   (c) Let *Cost(V')* be the maximum of *Length(V')* and *MaxDist(V')*.
2. Pick the ordered subset V' for which *Cost(V')* is minimal.
3. Return the following solution: The salesperson follows the shortest paths between the consecutive vertices of V'. Each of the agents meets the salesperson at a closest vertex to that agent in V'. The salesperson waits for all the agents who come to a certain vertex before moving to the next vertex.

Fig. 3. A PTAS for Min-Max Purchase cTSP

most three edges of the MST, which means that the cost of the solution is at most thrice the weight of the heaviest edge of the MST.

On the other hand, consider an optimal solution, and define $G'' = (V', E'')$, such that $(u_1, u_2) \in E''$ iff the participant from u_1 sold the goods to the participant from u_2, or vice versa. Let the weight of $(u_1, u_2) \in E''$ in G'' be the distance between u_1 and u_2 in G. The optimal cost is clearly at least the weight of the heaviest edge in E'', since selling to an agent requires traveling to this agent's vertex.

Note that G'' is a connected subgraph of G'. It is well-known that an MST is lexicographically minimal, i.e., its heaviest edge is not heavier than that of any other spanning-tree or spanning connected-subgraph. Therefore, the cost of the solution found by the above algorithm is at most thrice the cost of an optimal solution.

3.2 Min-Max Purchase cTSP

We next present a simple PTAS, **Coarse-Path**, described in Figure 3.

Theorem 4. *Algorithm* **Coarse-Path** *is a PTAS for* Min-Max Purchase cTSP*.*

Proof. Clearly, the Min-Max cost of the solution returned by the algorithm is the minimal $Cost(V')$ of the subsets it considers. We show that one of these subsets has $Cost(V')$ of at most $(1+\varepsilon)$ times the optimum.

Consider an optimal solution to the problem π, in which the cost is OPT. Choose a subset of the vertices of the path traveled by the salesperson in the following way. Start with vertex v, and then choose a vertex iff its distance from the previous vertex chosen is at least $\varepsilon \cdot OPT$. Clearly, at most $1/\varepsilon$ vertices are selected. Denote this subset by V'. Note that $Length(V') \leq OPT$.

For each vertex $u \notin V'$ that contains an agent, there is a vertex in V' at a distance of at most $(1+\varepsilon) \cdot OPT$. This holds, since for each vertex w visited

by the salesperson in π, V' contains a vertex at a distance of at most $\varepsilon \cdot OPT$ from w. Thus, $Cost(V') \leq MaxDist(V') \leq (1+\varepsilon)OPT$. Therefore, Algorithm Coarse-Path indeed finds a $(1+\varepsilon)$-approximate solution. The running-time of the algorithm is $O((2n)^{\lfloor 1/\varepsilon \rfloor + 3})$, since it enumerates over ordered subsets of vertices of size at most $\lfloor 1/\varepsilon \rfloor$, and the required computation for each ordered subset takes at most $O(n^3)$ time. Thus, **Coarse-Path** is a PTAS.

4 Discussion and Open Problems

We obtained quite tight approximation and intractability results for most of the cTSP problems. Some of the cTSP problems turn out to be easier (in sense of approximation) than the classical TSP, while others are strictly harder.

The status of Min-Makespan Sales cTSP is not settled, as there is an $O(\sqrt{\log n})$ approximation and a constant inapproximability factor. Improving the factors of this problem as well as tightening the factors for some others is yet to be achieved. It is also likely that the running time of some of the PTAS can be improved.

There are some disturbing asymmetries in the Euclidean results (see Table 2). For example, while the roundtrip versions of Min-Sum Sales and Full-Cooperation cTSP have a PTAS, the best approximations for the corresponding path-cTSP problems only guarantee some constant factors. We conjecture that these two path-cTSP versions indeed have a PTAS, but we suspect that this may not be very easy to prove. This follows since it can be shown that a PTAS for the first problem implies a (currently unknown) PTAS for the well-studied 3-bounded-degree-planar Minimum Spanning Tree (e.g., [PV84, KRY96, FKK+97, Cha03, AC04]).

Acknowledgements

We would like to thank Joseph Mitchell and Uri Zwick for many fruitful discussions.

References

[ABF+02] E. M. Arkin, M. A. Bender, S. P. Fekete, J. S. B. Mitchell, and M. Skutella. The Freeze-Tag Problem: How to Wake Up a Swarm of Robots. In *Proc. of SODA'02*, pages 568–577, 2002.

[ABG+03] E. M. Arkin, M. A. Bender, D. Ge, S. He, and J. S. B. Mitchell. Improved Approximation Algorithms or the Freeze-Tag Problem. In *SPAA'03*, pages 295–303, 2003.

[AC04] S. Arora and K. L. Chang. Approximation Schemes for Degree-Restricted MST and Red-Blue Separation Problems. *Algorithmica*, 40(3):189–210, 2004.

[AH94] E. Arkin and R. Hassin. Approximation Algorithms for the Geometric Covering Salesman Problem. *Discrete Applied Math.*, 55:197–218, 1994.

[Aro98] S. Arora. Polynomial-time Approximation Schemes for Euclidean TSP and other Geometric Problems. *Journal of the ACM*, 45(5):753–782, 1998.

[BNGNS98] A. Bar-Noy, S. Guha, J. Naor, and B. Schieber. Multicasting in Heterogeneous Networks. In *Proc. of STOC'98*, pages 448–453, 1998.

[Cha03] T. M. Chan. Euclidean Bounded-Degree Spanning Tree Ratios. In *Proc. 19th ACM SoCG*, pages 11–19, 2003.

[Chr76] N. Christofides. Worst-Case Analysis of a New Heuristic for the Traveling Salesman Problem. Technical report, Graduate School of Industrial Administration, Carnegy–Mellon University, 1976.

[dBGK+05] M. de Berg, J. Gudmundsson, M. J. Katz, C. Levcopoulos, M. H. Overmars, and A. F. van der Stappen. TSP with Neighborhoods of Varying Size. *Journal of Algorithms*, 57:22–36, 2005.

[DM01] A. Dumitrescu and J. S. B. Mitchell. Approximation Algorithms for TSP with Neighborhoods in the Plane. In *Proc. of SODA'01*, pages 38–46, 2001.

[EK01] L. Engebretsen and M. Karpinski. Approximation Hardness of TSP with Bounded Metrics. In *Proc. of ICALP'01*, pages 201–212, 2001.

[FKK+97] S. P. Fekete, S. Khuller, M. Klemmstein, B. Raghavachari, and N. Young. A Network Flow Technique for Finding Low-Weight Bounded-Degree Trees. *Journal of Algorithms*, 24:310–324, 1997.

[HHL88] S. M. Hedetniemi, S. T. Hedetniemi, and A.L. Liestman. A Survey of Gossiping and Broadcasting in Communication Networks. *Networks*, 18(4):319–359, 1988.

[KLS04] J. Knemann, A. Levin, and A. Sinha. Approximating the Degree-Bounded Minimum Diameter Spanning Tree Problem. *Algorithmica*, 41(2):117–129, 2004.

[KRY96] S. Khuller, B. Raghavachari, and N. Young. Low Degree Spanning Trees of Small Weight. *SIAM Journal of Computing*, 25(2):355–368, 1996.

[Mit99] J. S. B. Mitchell. Guillotine Subdivisions Approximate Polygonal Subdivisions: Part II – A simple polynomial-time approximation scheme for geometric TSP, k-MST, and related problems. *SIAM Journal of Computing*, 28(4):1298–1309, 1999.

[Mit06] J. S. B. Mitchell. A PTAS for TSP with Neighborhoods Among Fat Regions in the Plane. Private communication, 2006.

[MM95] C. Mata and J. S. B. Mitchell. Approximation Algorithms for Geometric Tour and Network Design Problems. In *SCG'95*, pages 360–369, 1995.

[PV84] C. H. Papadimitriou and U. V. Vazirani. On Two Geometric Problems Related to the Traveling Salesman Problem. *J. of Alg.*, 5:231–246, 1984.

[Rav94] R. Ravi. Rapid Rumor Ramification: Approximating the Minimum Broadcast Time. In *Proc. of FOCS'94*, pages 202–213, 1994.

[SABM02] M. O. Sztainberg, E. M. Arkin, M. A. Bender, and J. S. B. Mitchell. Analysis of Heuristics for the Freeze-Tag Problem. In *Proc. of SWAT'02*, pages 270–279, 2002.

[SS05] S. Safra and O. Schwartz. On the Complexity of Approximating TSP with Neighborhoods and Related Problems. *Computational Complexity*, 14:281–307, 2005.

Fréchet Distance for Curves, Revisited

Boris Aronov[1,⋆], Sariel Har-Peled[2], Christian Knauer[3], Yusu Wang[4], and Carola Wenk[5]

[1] Dept. of Comp. and Info. Sci., Polytechnic Univ., Brooklyn, NY 11201
aronov@cis.poly.edu
[2] Dept. of Comp. Sci., University of Illinois,1304 West Springfield Ave., Urbana, IL 61801
sariel@cs.uiuc.edu
[3] Freie Universität Berlin, Inst. of Comp. Sci., Takustr 9, 14195 Berlin, Germany
christian.knauer@inf.fu-berlin.de
[4] Dept. of Comp. Sci. and Engineering, The Ohio State Univ., Columbus, OH 43016
yusu@cse.ohio-state.edu
[5] Dept. of Comp. Sci., Univ. of Texas at San Antonio, One UTSA Circle, San Antonio, TX 78249
carola@cs.utsa.edu

Abstract. We revisit the problem of computing the Fréchet distance between polygonal curves, focusing on the *discrete* Fréchet distance, where only distance between vertices is considered. We develop efficient approximation algorithms for two natural classes of curves: κ-*bounded* curves and *backbone* curves, the latter of which are widely used to model molecular structures. We also propose a pseudo–output-sensitive algorithm for computing the discrete Fréchet distance exactly. The complexity of the algorithm is a function of the complexity of the free-space boundary, which is quadratic in the worst case, but tends to be lower in practice.

1 Introduction

The Fréchet distance is a natural measure of similarity between two curves [AG95]. An intuitive definition of the Fréchet distance is to imagine that a dog and its handler are walking on their respective curves. Both can control their speed but can only go forward. The Fréchet distance of these two curves is the minimal length of any leash necessary for the dog and the handler to move from the starting points of the two curves to their respective endpoints. The Fréchet distance and its variants have been widely used in many applications such as dynamic time-warping [KP99], speech recognition [KHM+98], signature verification [PP90], and matching of time series in databases [KKS05].

Alt and Godau [AG95] present an algorithm to compute the Fréchet distance between two polygonal curves of n and m vertices, respectively, in $O(nm \log(nm))$ time. Improving this roughly quadratic-time solution for general curves seems to be hard, and so far, no algorithm, exact or approximate, with running time

⋆ Research supported in part by NSF ITR Grant CCR-00-81964 and by a grant from US-Israel Binational Science Foundation.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 52–63, 2006.

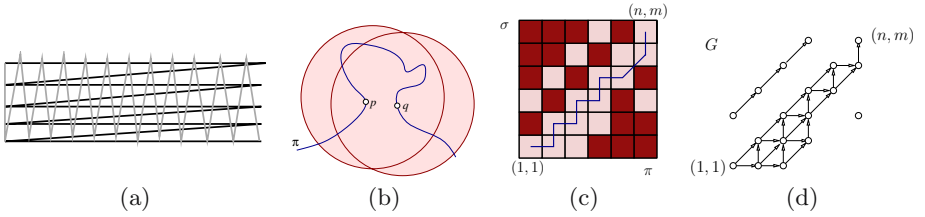


Fig. 1. (a) Light and dark curves are close under Hausdorff, but far under Fréchet distance. (b) π is κ-*bounded* if and only if for any $p, q \in \pi$, subchain $\pi(p, q)$ lies inside the shaded region: the radius of the two disks (centered at p and q, respectively) is $\kappa \mathrm{d}(p, q)/2$. (c) The free-space diagram $D(\pi, \sigma, \delta)$ and a viable path. (d) The directed graph G corresponds to the white cells in (c).

lower than $O(nm)$ has been found for this problem for general curves. A slightly simpler version of the Fréchet distance is the *discrete Fréchet distance*, which only considers vertices of polygonal curves. Its computation takes $\Theta(n^2)$ time and space using dynamic programing [EM94], and no subquadratic algorithm is known either. Both the continuous and discrete Fréchet distance are related to the edit distance problem, for which no substantially subquadratic algorithm is known.

On the other hand, another similarity measure, the *Hausdorff distance*, can be computed faster in the plane and approximated efficiently in higher dimensions. Unfortunately, the Hausdorff distance does not reflect curve similarity well (see Figure 1(a) for an example). Alt et al. [AKW04] showed that the Hausdorff distance and the Fréchet distance are the same for a pair of closed convex curves. They also showed that the two measures are closely related for κ-bounded curves (see Figure 1(b) for definition). In particular, they showed that the Fréchet distance between any two κ-bounded curves is bounded by $\kappa+1$ times the Hausdorff distance between them. This leads to a $(\kappa + 1)$-approximation algorithm for the Fréchet distance for any pair of κ-bounded curves that runs in near linear time in $\mathbb{R}^2$. Little is known about computing the Fréchet distance for other types of curves, including even x-monotone curves.

The problem of minimizing the Fréchet distance under various classes of transformations has also been studied [AKW01, Wen02, CM05], however the runtimes are very high and practical solutions remain elusive. Fréchet distance has also been extended to graphs (maps) [AERW03, BPSW05], to piecewise smooth curves [Rot05], to simple polygons [BBW06], and to surfaces [AB05]. Finally, Fréchet distance was used for high-dimensional approximate nearest neighbor search [Ind02], for efficient curve simplification [AHPMW05], and for curve morphing [EGHPM01].

Our results. Given the apparent difficulty of improving the worst-case time complexity of computing the (continuous or discrete) Fréchet distance between two unrestricted polygonal curves, we aim at developing algorithms for more realistic cases. First, in Section 3 we consider efficient approximation algorithms for the *discrete Fréchet distance*. Most current algorithms for computing the Fréchet

distance rely on a so-called *decision* procedure which determines whether the Fréchet distance between the two given curves is larger or smaller than a given value. We observe that an approximate solution to the decision problem can lead to an approximation of the discrete Fréchet distance, and curve simplification can help us approximate the decision problem efficiently. We apply this idea to two common families of curves: *κ-bounded curves* and *backbone curves*. In the former case, given an arbitrary polygonal curve π and a κ-bounded polygonal curve σ, of complexity n and m, respectively, we can $(1+\varepsilon)$-approximate their discrete Fréchet distance in $O((m + n\kappa^d\varepsilon^{-d})\log(nm))$ time in d dimensions. In the second case, both curves are so-called *backbone curves*, used widely to model molecular structures, such as protein backbones and DNA/RNA. We $(1+\varepsilon)$-approximate their discrete Fréchet distance in near linear time in two dimensions, and in $O(nm^{1/3}\log(nm)/\varepsilon^2)$ time in three dimensions.

In Section 4, we shift our focus back to the exact computation of the discrete Fréchet distance. Previously, the problem of deciding whether the Fréchet distance was smaller than some threshold was cast as finding some viable path in the so-called *free-space diagram*. We observe that such a path can be computed using only a subset $\mathcal{S}$ of cells in the free-space diagram. The size of $\mathcal{S}$ is nm in the worst case, but should be smaller in practical settings. Based on this observation, we present algorithms that compute the discrete Fréchet distance under the L_∞ norm in $O(|\mathcal{S}|\log(nm) + (n+m)\log^{2d}(nm))$ time in d dimensions. The case of the L_2 norm can be handled as well but with worse performance; running time and details are omitted from this version.

2 Preliminaries

Fréchet distance. A *(parameterized) curve* in $\mathbb{R}^d$ can be represented as a continuous function $f\colon [0,1] \to \mathbb{R}^d$. A *(monotone) reparametrization* α is a continuous non-decreasing function $\alpha\colon [0,1] \to [0,1]$ with $\alpha(0) = 0$ and $\alpha(1) = 1$. Given two curves $f, g\colon [0,1] \to \mathbb{R}^d$, their *Fréchet distance*, $\delta_{\mathrm{F}}(f,g)$, is defined as

$$\delta_{\mathrm{F}}(f,g) := \inf_{\alpha,\beta} \max_{t\in[0,1]} \mathrm{d}(f(\alpha(t)), g(\beta(t))).$$

where $\mathrm{d}(x,y)$ denotes the Euclidean distance between points x and y, and α and β range over all monotone reparametrizations.

Discrete Fréchet distance. A simpler variant of the Fréchet distance for two polygonal curves $\pi = \langle p_1, p_2, \ldots, p_n\rangle$ and $\sigma = \langle q_1, q_2, \ldots, q_m\rangle$ is the *discrete Fréchet distance*, denoted by $\delta_{\mathrm{D}}(\pi,\sigma)$. Imagine that both the dog and its handler can only stop at vertices of π and σ, and at any step, each of them can either stay at their current vertex or jump to the next one. The discrete Fréchet distance is defined as the minimal leash necessary at these discrete moments.

To formally define the discrete Fréchet distance, we first consider a discrete analog of (α,β), i.e, the correspondences between continuous reparametrizations. In particular, an *order-preserving complete correspondence* between π and σ is a

set $M \subseteq \{(p,q) \mid p \in \pi, q \in \sigma\}$ of pairs of vertices which is (a) *order-preserving*: if $(p_i, q_j) \in M$, then no $(p_s, q_t) \in M$ for $s < i$ and $t > j$, nor for $s > i$ and $t < j$; and (b) *complete*: for any $p \in \pi$ (respectively, $q \in \sigma$), there exists some pair involving p (respectively, q) in M. The *discrete Fréchet distance* between π and σ, $\delta_{\mathrm{D}}(\pi, \sigma)$, is then

$$\delta_{\mathrm{D}}(f,g) := \min_{M} \max_{(p,q)\in M} \mathrm{d}(p,q),$$

where M range over all order-preserving complete correspondences between π and σ.

It is well known that discrete and continuous versions of the Fréchet distance relate to each other as follows:

$$\delta_{\mathrm{F}}(\pi,\sigma) \leq \delta_{\mathrm{D}}(\pi,\sigma) \leq \delta_{\mathrm{F}}(\pi,\sigma) + \max\{\ell_1, \ell_2\},$$

where ℓ_1 and ℓ_2 are the lengths of the longest edges in π and σ, respectively. This suggests using δ_{D} to approximate δ_{F}. Unfortunately, it seems that computing $\delta_{\mathrm{D}}(\pi,\sigma)$ is asymptotically almost as hard as computing $\delta_{\mathrm{F}}(\pi,\sigma)$.

Decision problem. In the original paper [AG95], to compute $\delta_{\mathrm{F}}(\pi,\sigma)$, first, the *decision problem* "Given a parameter $\delta \geq 0$, is $\delta_{\mathrm{F}}(\pi,\sigma) \leq \delta$?" is solved by a dynamic programming algorithm that runs in $\Theta(nm)$ time and space. This algorithm is then used as a subroutine to search for $\delta_{\mathrm{F}}(\pi,\sigma)$ using the parametric search paradigm within $O(nm \log nm)$ time [AG95, AST94]. Our algorithms follow a similar framework combining decision problem and binary search (instead of parametric search). Thus we describe below how to solve the decision problem for the discrete case.

Given two polygonal chains π and σ and a distance threshold $\delta \geq 0$, we construct the following *free-space diagram* $D = D(\pi, \sigma, \delta)$: D is an $n \times m$ matrix (grid) and a cell $D[i,j]$ has value 1 if $\mathrm{d}(p_i, q_j) \leq \delta$, and value 0 otherwise. We refer to 1-cells as *white* and 0-cells as *black*. A *viable* path in D is a path connecting $(1,1)$ to (n,m), visiting only white cells of D, and moving in one step from (i,j) to either $(i, j+1)$, $(i+1, j)$, or $(i+1, j+1)$. It is easy to check that a complete order-preserving correspondence M induces a viable path in D and vice versa (see Figure 1 (c)). Hence the problem of deciding "$\delta_{\mathrm{D}}(\pi,\sigma) \leq \delta$?" is equivalent to deciding the existence of a viable path in D.

Given D, one can extract a viable path, if it exists, in $\Theta(nm)$ time using dynamic programming. Alternatively, one can traverse a directed graph G defined as follows: The nodes of G are the white cells of D. A white cell is connected to its top, right, and/or top-right neighbor cells by a directed edge, if they are white. See Figure 1(d). The size of G is $O(|W|)$, where $|W|$ is the number of white cells of D. Given G, testing "$\delta_{\mathrm{D}}(\pi,\sigma) \leq \delta$?" corresponds to a connectivity check in G (from $(1,1)$ to (n,m)) that can be performed in time $O(|W|)$.

Approximations. We say that τ is an $(1+\varepsilon)$-approximation of $\delta(\pi,\sigma)$ if

$$(1-\varepsilon)\tau \leq \delta(\pi,\sigma) \leq (1+\varepsilon)\tau.$$

An algorithm $(1+\varepsilon)$-*approximates* the decision problem "Is $\delta(\pi,\sigma) \leq \tau$?", if it returns YES whenever $\delta(\pi,\sigma) < (1-\varepsilon)\tau$ and NO whenever $\delta(\pi,\sigma) > (1+\varepsilon)\tau$. If τ is a $(1+\varepsilon)$-approximation of $\delta(\pi,\sigma)$, the algorithm is allowed to return either YES or NO. We also call such an algorithm a *fuzzy decision procedure*.

3 Approximation Algorithms Based on Simplification

In this section, we first introduce a general framework for approximating the discrete Fréchet distance by using a fuzzy decision procedure. Based on this framework we then develop efficient approximation algorithms, using curve simplification and packing arguments, for two families of common curves: κ-bounded curves and backbone curves.

3.1 Approximation Via a Fuzzy Decision Procedure

Given a set P of N points in $\mathbb{R}^d$, a *well-separated pairs decomposition (WSPD)* of P with separation constant s is a collection $\{(A_i, B_i)\}$ of pairs of subsets of P, with the property that (1) for every pair of points $x, y \in P$, there is an index i, so that $x \in A_i$ and $y \in B_i$ and (2) the minimum distance between A_i and B_i is at least s times the diameter of either set. The *size* of a WSPD is $\sum_i (|A_i| + |B_i|)$. For a constant s, a WSPD of size $O(N)$ can be computed in $O(N \log N)$ time [CK95]. For every pair (A_i, B_i) in the WSPD, we choose an arbitrary pair of points (p_i, q_i), with $p_i \in A_i$ and $q_i \in B_i$, as its *representatives*. We set $s = 10$. It is easy to check that the distance between any two points $x, y \in P$ is $(1+\frac{1}{5})$-approximated by the distance between the representatives of the corresponding WSPD pair.

Consider an optimization problem whose solution δ^* is a distance determined by a pair of points in P. Let X be the set of all distances induced by pairs of points of P. We now describe how to solve the optimization problem approximately by using an exact decision procedure. We start by constructing a WSPD as above and considering the set $Y := \{\mathrm{d}(p_i, q_i)\}$ of $O(N)$ distances between representative points of the decomposition pairs. By definition of WSPD, every distance in X is $(1+\frac{1}{5})$-approximated by some distance in Y. Hence some value in Y is a $(1+\frac{1}{5})$-approximation of δ^*, as δ^* is defined by a pair of points in P. Next, form a larger set Y' of distances by adding to Y', for each value $y \in Y$, the two values $\frac{4}{5}y$ and $\frac{6}{5}y$. We then perform a binary search on Y' using the decision procedure to identify the smallest interval $\mathcal{I} = [a, b]$ that contains δ^*. (The cost is dominated by $O(\log N)$ invocations of the decision procedure and the $O(N \log N)$ time to construct the WSPD.) Notice that $b \leq \frac{3}{2}a$, as by above discussion, δ^* is contained in the interval $[\frac{4}{5}y, \frac{6}{5}y]$ for some $y \in Y$ and we have included both $\frac{4}{5}y$ and $\frac{6}{5}y$ in Y'. We now perform another (numerical) binary search on this interval to identify the interval $[a', b'] \subseteq [a, b]$ containing δ^* with $b' \leq (1+\varepsilon)a'$, giving rise to a $(1+\varepsilon)$-approximation of δ^*. The second binary search invokes the decision procedure $O(\log \varepsilon^{-1})$ times.

Interestingly, the decision procedure does not have to be exact—the above binary search can be adapted to work with a fuzzy decision procedure with the same performance guarantees. We omit the details from current version.

Theorem 1. *Let P be a set of N points in $\mathbb{R}^d$, and let Z be any optimization problem, for which the optimal answer is a distance induced by a pair of points of P. Given a fuzzy decision procedure for Z, one can $(1+\varepsilon)$-approximate the optimal solution in*

$$O(N \log N + T_{\text{FDECISION}}(N, 1/10) \log N + T_{\text{FDECISION}}(N, \varepsilon/4) \log \varepsilon^{-1})$$

time, where $T_{\text{FDECISION}}(N, c)$ is the running time of the fuzzy decision procedure on N points when the required approximation factor is $1 + c$.

Returning to the computation of discrete Fréchet distance, observe that there must exist some $p^* \in \pi$ and $q^* \in \sigma$ such that $\mathrm{d}(p^*, q^*) = \delta_{\mathrm{D}}(\pi, \sigma)$. Hence, by applying the above theorem to the set of all vertices from π and σ, we only need a fuzzy decision procedure for $\delta_{\mathrm{D}}(\pi, \sigma)$ in order to approximate $\delta_{\mathrm{D}}(\pi, \sigma)$.

3.2 Approximation with Simplifications

The remaining question is how to implement a fuzzy decision procedure efficiently. We show that curve simplification together with a packing argument can be used to achieve guaranteed efficiency for the two classes of common curves that we investigate.

Greedy simplification. Given a polygonal chain $\pi = \langle p_1, \ldots, p_n \rangle$, we simplify π to obtain $\widetilde{\pi} = \langle \hat{p}_1, \ldots, \hat{p}_k \rangle$, where vertices of $\widetilde{\pi}$ form a subsequence of π, with $\hat{p}_1 = p_1$ and $\hat{p}_k = p_n$. Let $I_\pi(i) = j$ if $\hat{p}_i = p_j \in \pi$; the subscript π is omitted when it is clear from context. We say that $\widetilde{\pi}$ *μ-simplifies* π if (i) $I(i) < I(k)$ for $i < k$, and (ii) $\mathrm{d}(\hat{p}_i, p_k) \leq \mu$ for any k such that $I(i) \leq k < I(i+1)$. (This definition is slightly different from the standard one in the literature.) We construct $\widetilde{\pi}$, a μ-simplification of π, in a greedy manner: Start with $\hat{p}_1 = p_1$. At some stage, suppose we have already computed $\hat{p}_i = p_j$. In order to find $I(i+1)$, we check each vertex of π starting from p_j in order, and stop when we reach the first edge $p_k p_{k+1}$ of π such that $\mathrm{d}(p_j, p_k) \leq \mu$ and $\mathrm{d}(p_j, p_{k+1}) > \mu$. We set $\hat{p}_{i+1} = p_{k+1}$ and continue, until we reach p_n, at which point we add p_n as the final vertex of $\widetilde{\pi}$. The entire procedure takes linear time. By construction, the following observation is straightforward.

Observation 2. *Any edge $\hat{p}_i \hat{p}_{i+1}$ in $\widetilde{\pi}$ other than the last edge, $\mathrm{d}(\hat{p}_i, \hat{p}_{i+1}) > \mu$.*

The following fact follows easily by an explicit construction. We omit the details.

Lemma 1. *If $\widetilde{\pi}$ and $\widetilde{\sigma}$ be μ-simplifications of curves π and σ, respectively, then*

$$\delta_{\mathrm{D}}(\pi, \sigma) - 2\mu \leq \delta_{\mathrm{D}}(\widetilde{\pi}, \widetilde{\sigma}) \leq \delta_{\mathrm{D}}(\pi, \sigma) + \mu.$$

The above lemma implies that if the answer to $\delta_{\mathrm{D}}(\widetilde{\pi},\widetilde{\sigma}) \leq \delta$ is YES, then, $\delta_{\mathrm{D}}(\pi,\sigma) \leq \delta + 2\mu$. If it is NO, then $\delta_{\mathrm{D}}(\pi,\sigma) \geq \delta - \mu$. Thus the decision problem for $\delta_{\mathrm{D}}(\widetilde{\pi},\widetilde{\sigma})$ $(1 + 2\mu/\delta)$-approximates that for $\delta_{\mathrm{D}}(\pi,\sigma)$. We next show that $\delta_{\mathrm{D}}(\widetilde{\pi},\widetilde{\sigma})$ can be answered asymptotically much faster for two special classes of curves, giving rise to efficient fuzzy decision procedure for them.

3.3 Fréchet Distance for κ-Bounded Curves

As defined by Alt et al. [AKW04], π is *κ-bounded* if $\pi(x,y) \subseteq B(x, \frac{\kappa}{2}\mathrm{d}(x,y)) \cup B(y, \frac{\kappa}{2}\mathrm{d}(x,y))$, for all $x, y \in \pi$, where $\pi(x,y)$ is the arc of π between x and y and $B(x,r)$ is the radius-r Euclidean ball centered at x[1]. See Figure 1(b) for an illustration in two dimensions. Examples of κ-bounded curves include κ-straight curves [AKW04] which in turn include *curves with increasing chords* [Rot94] and *self-approaching curves* [AAI+01].

We now describe how to construct a fuzzy decision procedure for the problem "$\delta_{\mathrm{D}}(\pi,\sigma) \leq \delta$?" for two polygonal curves π, σ where σ is κ-bounded. We first μ-simplify π and σ into $\widetilde{\pi}$ and $\widetilde{\sigma}$ respectively, using $\mu := \varepsilon\delta/2$. By Lemma 1, the decision problem for $\delta_{\mathrm{D}}(\widetilde{\pi},\widetilde{\sigma})$ is an $(1+\varepsilon)$-approximation decision procedure for $\delta_{\mathrm{D}}(\pi,\sigma)$. Hence we now focus on checking whether $\delta_{\mathrm{D}}(\widetilde{\pi},\widetilde{\sigma}) \leq \delta$. Let n, m, r, s be the size of π, σ, $\widetilde{\pi}$, and $\widetilde{\sigma}$ respectively; $r \leq n$ and $s \leq m$.

Decision problem for $\delta_{\mathrm{D}}(\widetilde{\pi},\widetilde{\sigma})$. Let $\widetilde{D}$ be the free-space diagram for $\widetilde{\pi}$ and $\widetilde{\sigma}$ with respect to δ. Recall that $\delta_{\mathrm{D}}(\widetilde{\pi},\widetilde{\sigma}) \leq \delta$ if and only if there exists a viable path in $\widetilde{D}$. This can be tested in $O(|W|)$ time once W, the set of white cells of $\widetilde{D}$, is given. We first bound the size of W.

For every $\hat{p} \in \widetilde{\pi}$, let $N(\hat{p})$ be the set of vertices from $\widetilde{\sigma}$ contained in $B(\hat{p},\delta)$. Obviously, $|W| = \sum_{\hat{p}\in\widetilde{\pi}} |N(\hat{p})|$. Consider any two points $q_1, q_2 \in \widetilde{\sigma}$ that lie in $B(\hat{p},\delta)$ for some $\hat{p} \in \widetilde{\pi}$. If q_1q_2 is an edge of $\widetilde{\sigma}$, $\mathrm{d}(q_1,q_2) \geq \mu$ by Observation 2. Otherwise

$$\sigma(q_1,q_2) \subseteq B(q_1, \frac{\kappa}{2}\mathrm{d}(q_1,q_2)) \cup B(q_2, \frac{\kappa}{2}\mathrm{d}(q_1,q_2)),$$

as σ is κ-bounded. Furthermore, in this case, let $q_1q \subset \widetilde{\sigma}$ be the edge with $q \in \sigma(q_1,q_2)$; $\mathrm{d}(q_1,q) \geq \mu$ by Observation 2. It then follows that $(1+\kappa/2)\mathrm{d}(q_1,q_2) \geq \mu$ and therefore $\mathrm{d}(q_1,q_2) \geq 2\mu/(\kappa+2)$. Hence $N(\hat{p}) = O((\kappa\delta/\mu)^d)$ by a straightforward packing argument. This means that the number of white cells is $|W| = O(s(\kappa\delta/\mu)^d) = O(n(\kappa\delta/\mu)^d)$ given that σ is a κ-bounded curve.

We still need to compute $N(\hat{p})$ efficiently, that is, to enumerate the set of vertices of $\widetilde{\sigma}$ contained in $B(\hat{p},\delta)$ for every $\hat{p} \in \widetilde{\pi}$. This can be done by a spherical range query, which unfortunately, there is no known efficient algorithms. We hence replace exact spherical range queries by approximate ones by rounding vertices of $\widetilde{\sigma}$ to vertices of some grid of appropriate size. Overall, the set of white cells in $\widetilde{D}$ can be computed in $O(r + s + r(\kappa\delta/\mu)^d)$ time. Details are omitted from the conference version.

[1] We have slightly abused the notation by treating a curve section as a point set.

Putting everything together and substituting $\mu = \varepsilon\delta/2$, we have a $(1+\varepsilon)$-approximation decision procedure for $\delta_{\mathrm{D}}(\pi,\sigma)$ that runs in $O(n+m+n\kappa^d\varepsilon^{-d})$ time and space in $\mathbb{R}^d$. An application of Theorem 1 now yields.

Theorem 3. *A $(1+\varepsilon)$-approximation of $\delta_{\mathrm{D}}(\pi,\sigma)$ for a polygonal curve π and a κ-bounded curve σ, of size n and m respectively, can be computed in $O((m+n\kappa^d\varepsilon^{-d})\log(n/\varepsilon))$ time and $O(n+m+n\kappa^d/\varepsilon^d)$ space in d dimensions.*

3.4 Fréchet distance for Protein Backbones

In molecular biology, it is common to model a protein backbone by a polygonal chain, where each C_α atom becomes a vertex, and each edge represents a covalent bond between two consecutive amino acids. All the bonds have approximately the same bond length, and no two atoms (thus vertices) can get too close to each other due to van der Waals interactions. This is the motivation behind the study of *backbone* curves, which have the following properties:

P1. For any two non-consecutive vertices u and v of the curve, $\mathrm{d}(u,v) \geq 1$,
P2. Every edge of the curve has length between c_1 and c_2, where $c_2 > c_1 > 0$ are constants.

Although proteins lie in three-dimensional space, there are simplified models for protein backbones in both two and three dimensions [GIP99, KS94].

Given backbone curves π and σ in $\mathbb{R}^d$ and given a distance threshold $\delta \geq 0$, we want to test whether $\delta_{\mathrm{D}}(\pi,\sigma) \leq \delta$. We μ-simplify π and σ to obtain $\widetilde{\pi}$ and $\widetilde{\sigma}$ as in the previous section, for $\mu = \varepsilon\delta/2$, and construct the free-space diagram $\widetilde{D}$ for $\widetilde{\pi}$ and $\widetilde{\sigma}$ with respect to δ. $\widetilde{D}$ is an $r \times s$ grid, where by Observation 2 and property P2, $r = |\widetilde{\pi}| \leq c_2 n/\mu$ and $s = |\widetilde{\sigma}| \leq c_2 m/\mu$. Given $\widetilde{D}$, we can compute W, the set of white cells in $\widetilde{D}$ by the same approach as the one for κ-bounded curves in $O(r+s+|W|)$ time and space. Once $\widetilde{D}$ and W are given, the decision problem can be solved in time proportional to $|W|$. Below we present an upper bound for $|W|$.

Bounding $|W|$. A straightforward bound[2] for $|W|$ is $O(\min\{r\delta^d, s\delta^d\})$, as by a packing argument and property P1, there are at most $O(\delta^d)$ vertices lying in δ-neighborhood of any vertex of $\widetilde{\pi}$ and $\widetilde{\sigma}$. If $\delta < 1$, then the number of white cells is $O(n+m)$. Hence we now assume that $\delta \geq 1$.

We can improve this bound on $|W|$ by a more careful counting analysis. Assume without loss of generality that $r \leq s$. For any vertex $\hat{p} \in \widetilde{\pi}$ and its δ-neighborhood $B(\hat{p},\delta)$, let $E(\hat{p})$ be the set of edges of $\widetilde{\sigma}$ intersecting the ball $B(\hat{p},\delta)$. The number of vertices of $\widetilde{\sigma}$ in $B(\hat{p},\delta)$ is upper bounded by $O(|E(\hat{p})|)$. Furthermore, given any edge $e = (\hat{q}_i, \hat{q}_{i+1}) \in \widetilde{\sigma}$, let $\sigma(e) = \sigma(q_{I_q(i)}, q_{I_q(i+1)})$ (that is, the subchain $\sigma(e) \subseteq \sigma$ that simplified into edge e in chain $\widetilde{\sigma}$). $E(\hat{p})$ can be partitioned into two sets: (i) $E_1 = \{e \in E(\hat{p}) \mid \sigma(e) \subseteq B(\hat{p},\delta)\}$, and (ii) $E_2 = \{e \in E(\hat{p}) \mid$ at least one vertex of $\sigma(e)$ lies outside $B(\hat{p},\delta)\}$.

[2] In what follows, the big-O notation may hide factors depending on constants c_1 and c_2.

By property P2, the number of vertices in $\sigma(e)$ is at least μ/c_2 for any $e \in \widetilde{\sigma}$. Therefore $|E_1| = O(c_2\delta^d/\mu)$. On the other hand, for every edge $e \in E_2$, there is at least one vertex of $\sigma(e)$ that lies in the spherical shell of $B(\hat{p}, \delta + c_2) \setminus B(\hat{p}, \delta)$, as the length of edges in σ is at most c_2. Since the volume of this spherical shell is $O(c_2(c_2+\delta)^{d-1})$, the size of E_2 is bounded by $O((c_2(c_2+\delta)^{d-1}/(c_1^{d-1})))$. Therefore, $|E(\hat{p})| = |E_1| + |E_2| = O(\delta^{d-1} + \delta^d/\mu)$. Summing over all r vertices of $\widetilde{\pi}$, we obtain $|W| = O(\frac{n}{\mu}(\delta^{d-1} + \delta^d/\mu))$. As this number cannot exceed the size of $\widetilde{D}$ which is $O(rs) = O(nm/\mu^2)$, we have $|W| = \min\{nm/\mu^2, O(\frac{n}{\mu}(\delta^{d-1} + \delta^d/\mu)\}$. $|W|$ is maximized when the two balancing terms are equal: $\frac{nm}{\varepsilon^2\delta^2} = \frac{\delta^{d-2}}{\varepsilon^2}$, that is, when $\delta = m^{1/d}$. This implies that $|W| = O(nm^{1-2/d}/\varepsilon^2)$.

Finally, by applying Theorem 1 and putting everything together, we conclude with the following result. We remark that similar but more involved argument can also be used to approximate the continuous Fréchet distance for two backbone curves. We omit the details from current conference version.

Theorem 4. *Given two backbone curves π and σ of n and $m \geq n$ vertices respectively, we can compute a $(1+\varepsilon)$-approximation of $\delta_{\mathrm{D}}(\pi, \sigma)$ in time $O((n+m)\varepsilon^{-2}\log(nm))$ in the plane, and $O(nm^{1/3}\varepsilon^{-2}\log(nm))$ in three dimensions.*

4 Pseudo-Output-Sensitive Algorithm

In this section, we present a pseudo-output-sensitive algorithm for computing $\delta_{\mathrm{D}}(\pi, \sigma)$ for general curves π and σ of size n and m, respectively. Although in the worst-case the runtime may still be $\Theta(nm)$ for solving the decision problem, we believe that our observation should help produce efficient algorithms for the Fréchet distance in practice. In what follows, we describe results for the L_∞ norm (which yield a constant-factor approximation for the L_2 norm). The case of the L_2 norm is more involved and the running times are slower; details are omitted from the conference version.

Binary search. We show how to compute $\delta^* = \delta_{\mathrm{D}}(\pi, \sigma)$ using a variant of binary search. Assume we have algorithms to solve the decision problem "Is $\delta^* = \delta_{\mathrm{D}}(\pi, \sigma) \leq \delta$?" in time $A(n+m)$ and to answer the *distance selection* query "Given a set of N points P and a rank k, what is the kth smallest distance among all pairwise distances from P?" in $B(N)$ time. Combining the two algorithms by performing a binary search on the set of all inter-vertex distances, we can find δ^* in $O((A(n+m) + B(n+m))\log(nm))$ time. For the L_∞ norm, the distance-selection problem can be solved in $O(dN\log^{d-1} N)$ in $\mathbb{R}^d$ [Sal89]. For the decision problem, a straightforward bound for the runtime of A is $O(|W|)$ plus the time to compute W, the set of white cells in the free-space diagram $D = D(\pi, \sigma, \delta)$ for a threshold $\delta > 0$. Below we provide a tighter bound for A although its worst-case complexity is still $\Theta(nm)$.

Boundary cells. Given an $n \times m$ matrix D representing the free-space diagram with respect to some threshold δ, a *boundary cell* is a white cell whose immediate

neighbor above or below it is black. So if $D[i, j]$ is a boundary cell, then the edge $q_jq_{j+1} \subset \sigma$ (or q_jq_{j-1}) intersects the boundary of $B(p_i, \delta)$ exactly once (one endpoint must lie inside and one must be outside); we say that the edge q_jq_{j+1} *crosses* the boundary of $B(p_i, \delta)$. Let $\mathcal{S} = \mathcal{S}(\pi, \sigma, \delta)$ denote the set of boundary cells of $D(\pi, \sigma, \delta)$. Although in the worst case $|\mathcal{S}| = \Omega(|W|) = \Omega(nm)$, we expect it to be much smaller than $|W|$ in practice. For example, consider the case when vertices of σ form lines of a cubic lattice of size $n^{1/3} \times n^{1/3} \times n^{1/3}$ and δ is roughly $n^{1/3}/2$. For a vertex p at the center of this cube, the number of white cells in the corresponding column in D is $\Theta(n)$, while the number of boundary cells is $\Theta(n^{2/3})$. The remaining questions are (i) how to compute the set of boundary cells $\mathcal{S}(\pi, \sigma, \delta)$ and (ii) how to solve the decision problem once $\mathcal{S}$ is given.

Computing $\mathcal{S}$. Given $p \in \mathbb{R}^d$ and δ, let $\mathcal{S}(p, \delta)$ denote the set of edges from σ crossing the boundary of $B(p, \delta)$. Since one endpoint of each edge has to be inside $B(p, \delta)$ and the other endpoint outside, this is different from the standard segment/ball intersection problem. To compute $\mathcal{S}$, we need to perform n *edge/ball crossing queries*, one for each vertex from π. Under the L_∞ norm, $B(p, \delta)$ is a cube centered at p, and the basic operation is an edge/cube crossing query, where all cubes are congruent. We can preprocess the set of edges in σ by building a standard multi-level data structure for their endpoints (similar to the multi-level range tree for orthogonal range reporting problem). In particular, there are altogether $2d$ levels: the first d levels are used to locate edges with one endpoint inside the query cube, and the second d levels are used to find those with the second endpoint outside of the query cube. The entire data structure has size $O(m \log^{2d-1} m)$ and can be built in $O(m \log^{2d} m)$ time. Given any query cube (in fact, the query can be any orthogonal box), the set of edges crossing it can be reported in $O(\log^{2d} m + k)$ time, where k is the number of such edges. The query time can be improved to $O(\log^{2d-1} m + k)$ using the fractional-cascading technique [CG86, Lue78].

Once $\mathcal{S}$ is given, we can solve the decision problem using dynamic programming in $O(|\mathcal{S}|)$ time and space. The technical details are omitted due to lack of space. Putting everything together, we conclude with the following theorem:

Theorem 5. *Given any two polygonal curves π and σ in $\mathbb{R}^d$, with n and m vertices, respectively, one can compute $\delta_{\mathrm{D}}(\pi, \sigma)$ under the L_∞-norm in $O(\Phi \log(nm) + (n + m) \log^{2d}(nm))$ time and $O(\Phi + (n + m) \log^{2d-1}(nm))$ space, where Φ is the maximum number of boundary cells for any threshold δ.*

5 Conclusion and Discussion

In this paper, we considered the problem of computing the discrete Fréchet distance between two polygonal curves either approximately or exactly. Our main contribution is a simple approximation framework that leads to efficient $(1+\varepsilon)$-approximation algorithms for two families of common curves: κ-bounded curves and backbone curves. We also considered the exact algorithm for general curves, and proposed a pseudo–output-sensitive algorithm by observing that

only a subset of the white cells from the free-space diagram are necessary for the decision problem. It will be interesting to investigate whether there are families of curves that are guaranteed to have small Φ, which is the maximum number of boundary cells, over all possible values of the threshold δ. We are currently working on extending the pseudo-output-sensitive algorithms to the continuous weak Fréchet distance.

It might be hard to develop algorithms that are significantly sub-quadratic in the worst case, given that no such algorithm exists for the related and widely studied problem of computing the edit distance for strings. Hence our future directions will focus on practical variants of the Fréchet distance that can handle outliers, partial matching, and/or efficient multiple-curve alignment. Another important direction is to develop efficient (approximation) algorithms for minimizing the Fréchet distance under transformations such as rigid motions.

References

[AAI+01] O. Aichholzer, F. Aurenhammer, C. Icking, R. Klein, E. Langetepe, and G. Rote. Generalized self-approaching curves. *Discrete Applied Mathematics*, 109:3–24, 2001.

[AB05] H. Alt and M. Buchin. Semi-computability of the Fréchet distance between surfaces. In *Proc. 21th European Workshop on Computational Geometry*, 2005.

[AERW03] H. Alt, A. Efrat, G. Rote, and C. Wenk. Matching planar maps. *J. Algorithms*, 49:262–283, 2003.

[AG95] H. Alt and M. Godau. Computing the Fréchet distance between two polygonal curves. *Internat. J. Comput. Geom. Appl.*, 5:75–91, 1995.

[AHPMW05] P. K. Agarwal, S. Har-Peled, N. Mustafa, and Y. Wang. Near-linear time approximation algorithms for curve simplification in two and three dimensions. *Algorithmica*, 2005. To appear.

[AKW01] H. Alt, C. Knauer, and C. Wenk. Matching polygonal curves with respect to the fréchet distance. In *Proceedings 18th International Symposium on Theoretical Aspects of Computer Science*, pages 63–74, 2001.

[AKW04] H. Alt, C. Knauer, and C. Wenk. Comparison of distance measures for planar curves. *Algorithmica*, 38(1):45–58, 2004.

[AST94] P. K. Agarwal, M. Sharir, and S. Toledo. Applications of parametric searching in geometric optimization. *J. Algorithms*, 17:292–318, 1994.

[BBW06] K. Buchin, M. Buchin, and C. Wenk. Computing the Fréchet distance between simple polygons in polynomial time. In *Proc. 22st Annu. ACM Sympos. Comput. Geom.*, pages 80–87, 2006.

[BPSW05] S. Brakatsoulas, D. Pfoser, R. Salas, and C. Wenk. On map-matching vehicle tracking data. In *Proc. 31st VLDB Conference*, pages 853–864, 2005.

[CG86] B. Chazelle and L. J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1:133–162, 1986.

[CK95] Paul B. Callahan and S. Rao Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *J. ACM*, 42(1):67–90, 1995.

[CM05] M. Clausen and A. Mosig. Approximately matching polygonal curves with respect to the Fréchet distance. *Comput. Geom. Theory Appl.*, 30:113–127, 2005.

[EGHPM01] A. Efrat, L.J. Guibas, S. Har-Peled, and T.M. Murali. Morphing between polylines. In *Proc. 12th ACM-SIAM Sympos. Discrete Algorithms*, pages 680–689, 2001.

[EM94] T. Eiter and H. Mannila. Computing discrete Fréchet distance. Technical Report CD-TR 94/64, Christian Doppler Laboratory for Expert Systems, TU Vienna, Austria, 1994.

[GIP99] D. Goldman, S. Istrail, and C. H. Papadimitriou. Algorithmic aspects of protein structure similarity. In *Proc. 40th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 512–522, 1999.

[Ind02] P. Indyk. Approximate nearest neighbor algorithms for Fréchet distance via product metrics. In *Proc. 18th Annu. ACM Sympos. Comput. Geom.*, pages 102 – 106, 2002.

[KHM+98] S. Kwong, Q. H. He, K. F. Man, K. S. Tang, and C. W. Chau. Parallel genetic-based hybrid pattern matching algorithm for isolated word recognition. *Int. J. Pattern Recognition & Artificial Intelligence*, 12(5):573–594, August 1998.

[KKS05] M.S. Kim, S.W. Kim, and M. Shin. Optimization of subsequence matching under time warping in time-series databases. In *Proc. ACM symp. Applied comput.*, pages 581–586, 2005.

[KP99] E. J. Keogh and M. J. Pazzani. Scaling up dynamic time warping to massive dataset. In *Proc. of the Third Euro. Conf. Princip. Data Mining and Know. Disc.*, pages 1–11, London, UK, 1999.

[KS94] A. Kolinski and J. Skolnick. Monte carlo simulations of protein folding: Lattice model and interaction scheme. In *Proteins*, volume 18, pages 338–352, 1994.

[Lue78] G. S. Lueker. A data structure for orthogonal range queries. In *Proc. 19th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 28–34, 1978.

[PP90] M. Parizeau and R. Plamondon. A comparative analysis of regional correlation, dynamic time warping, and skeletal tree matching for signature verification. *IEEE Trans. Pattern Anal. Mach. Intell.*, 12(7):710–717, 1990.

[Rot94] G. Rote. Curves with increasing chords. *Math. Proc. Camb. Phil. Soc.*, 115:1–12, 1994.

[Rot05] G. Rote. Computing the Fréchet distance between piecewise smooth curves. Technical Report ECG-TR-241108-01, May 2005.

[Sal89] J. S. Salowe. L_∞ interdistance selection by parametric search. *Information Processing Letters*, 30:9–14, 1989.

[Wen02] C. Wenk. *Shape Matching in Higher Dimensions*. PhD thesis, Dept. of Comput. Sci., Freie Universität Berlin, 2002.

Resource Allocation in Bounded Degree Trees

Reuven Bar-Yehuda[1], Michael Beder[1], Yuval Cohen[1], and Dror Rawitz[2]

[1] Department of Computer Science, Technion, Haifa 32000, Israel
{reuven@cs, sbederml@cs, scyuval@t2}.technion.ac.il
[2] Caesarea Rothschild Institute, University of Haifa, Haifa 31905, Israel
rawitz@cri.haifa.ac.il

Abstract. We study the *bandwidth allocation problem* (BAP) in bounded degree trees. In this problem we are given a tree and a set of connection requests. Each request consists of a path in the tree, a bandwidth requirement, and a weight. Our goal is to find a maximum weight subset S of requests such that, for every edge e, the total bandwidth of requests in S whose path contains e is at most 1. We also consider the *storage allocation problem* (SAP), in which it is also required that every request in the solution is given the same contiguous portion of the resource in every edge in its path. We present a deterministic approximation algorithm for BAP in bounded degree trees with ratio $(2\sqrt{e}-1)/(\sqrt{e}-1)+\varepsilon < 3.542$. Our algorithm is based on a novel application of the *local ratio technique* in which the available bandwidth is divided into narrow strips and requests with very small bandwidths are allocated in these strips. We also present a randomized $(2+\varepsilon)$-approximation algorithm for BAP in bounded degree trees. The best previously known ratio for BAP in general trees is 5. We present a reduction from SAP to BAP that works for instances where the tree is a line and the bandwidths are very small. It follows that there exists a $(2+\varepsilon)$-approximation algorithm for SAP in the line. The best previously known ratio for this problem is 7.

1 Introduction

The problems. We study the *bandwidth allocation problem* (BAP) in trees. In this problem we are given a tree $T = (V, E)$, where $m = |E|$, and a set J of n connection requests from clients. Each request j is associated with a *path* in the tree that is denoted by P_j (P_j is a set of edges), and a weight $w(j)$ that may be gained by accommodating it. It also has a bandwidth requirement, or *demand*, $d_j \in [0, 1]$. Given a parameter $\delta \in (0, 1)$, a request j is called *δ-narrow* (or simply *narrow*) if $d_j \leq \delta$. Otherwise, it is called *δ-wide* (*wide*). An instance in which all requests are narrow is called a *narrow instance*, and an instance in which all requests are wide is called a *wide instance*. A feasible solution, or a *schedule*, is a subset $S \subseteq J$ of connection requests such that, for every edge e, the total demand of requests whose path contains e is at most 1. That is, S is feasible if $\sum_{j \in S\,:\, e \in P_j} d_j \leq 1$, for every edge e. Our goal is to find a schedule with maximum total weight. When the given tree T is a line (i.e., a tree with two leaves) it is convenient to use temporal terms. In this case, the path of a request

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 64–75, 2006.

becomes a time interval, and our goal is to find a maximum weight subset $S \subseteq J$ such that at any given time the total bandwidth is bounded by 1.

We also consider the *storage allocation problem* (SAP) which is a variation of BAP with two additional constraints: (i) the specific portion of the resource allocated to a request cannot change between edges (or over time), and (ii) the allocation must be contiguous. Hence, given a SAP instance, a solution can be described by a set of requests $S \subseteq J$ and an assignment of every request $j \in S$ to a specific portion of the resource. Formally, a solution consists of the set of requests S and a height function $h : S \rightarrow [0, 1]$ such that the following constraints are satisfied: (i) $h(j)+d_j \leq 1$ for every $j \in S$, and (ii) for every two requests $j, k \in S$ such that $j \neq k$ and $P_j \cap P_k \neq \emptyset$ either $h(j) + d_j \leq h(k)$ or $h(k) + d_k \leq h(j)$. That is, we require that the portion of the resource assigned to j is within the range $[0, 1]$, and that no two requests occupy the same portion of the resource on the same edge (or at the same time). Observe that a feasible SAP schedule is a feasible BAP schedule, while the converse may not be true. Hence, the BAP optimum is at least as large as the SAP optimum of a given problem instance.

In the line topology a request j can be represented by an axis-parallel rectangle, whose length is $|P_j|$ (or the duration of the request, in temporal terms), and whose height is d_j. The rectangles are allowed to move vertically but not horizontally. We wish to select a maximum weight subset of rectangles that can be placed within a strip of height 1 such that no two rectangles overlap. A natural application of SAP in the line arises in a multi-threaded environment, where threads require contiguous memory allocations for fixed time intervals.

Related Work. Both BAP and SAP are NP-hard even in the line since they contain *knapsack* as the special case in which the paths of all requests share an edge. BAP in the line with unit demands is the problem of finding an independent set in a weighted interval graph which is solvable in polynomial time (see, e.g., [1]). BAP in the tree with unit demands is the problem of finding a maximum weight independent set of paths in a tree. This problem is also solvable in polynomial time [2]. Notice that BAP and SAP are equivalent in the case of unit demands.

The special case of BAP in the line, where all requests have the same length (or duration), was studied by Arkin and Silverberg [3]. Bar-Noy et al. [4] considered an online version of BAP in which the weight of a request is proportional to the area of the rectangle it induces. Phillips et al. [5] developed a 6-approximation algorithm for BAP in the line. Leonardi at al. [6] observed that SAP in the line can be used to model the problem of scheduling requests for remote medical consulting on a shared satellite channel. They obtained a 12-approximation algorithm for SAP in the line. Bar-Noy et al. [7] used the local ratio technique to improve the ratios for BAP and SAP in the line to 3 and 7, respectively.

Chen et al. [8] studied the special cases of BAP and SAP, where all demands are multiples of $1/K$ for some integer K. They developed dynamic programming algorithms for both problems that compute optimal solutions for wide instances. Their algorithm for BAP easily extends to general wide instances of BAP. However, in the case of SAP, the running time of the algorithm depends on K that may be exponential in the input size. They presented an approximation

algorithm for BAP with proportional weights. They also presented an approximation algorithm with ratio $\frac{e}{e-1} + \varepsilon$, for any $\varepsilon > 0$, for a special case of SAP in which $d_j = i/K$ for some $i \in \{1, \ldots, q\}$ where q is a constant.

Calinescu et al. [9] developed a randomized approximation algorithm for BAP in the line with expected performance ratio of $2 + \varepsilon$, for every $\varepsilon > 0$. They obtained their results by dividing the given instance into a wide instance and a narrow instance. They use dynamic programming to compute an optimal solution for the wide instance, and a randomized LP-based algorithm to obtain a $(1 + \varepsilon)$-approximate solution for the narrow instance. They also present a 3-approximation algorithm for BAP that is different from the one from [7].

The more general version of BAP in which the edge capacities are not uniform is called the *unsplittable flow problem* (UFP). Chakrabarti et al. [10] presented the first $O(1)$-approximation algorithm for UFP in the line by extending the approach of [9] to the non-uniform capacity case. However, their 13-approximation algorithm works under the *no-bottleneck assumption* which states that the maximum demand is not larger than the minimum edge capacity. Chekuri et al. [11] used an LP-based deterministic algorithm instead of a randomized algorithm to obtain a $(2 + \varepsilon)$-approximation algorithm for UFP in the line and a 48-approximation algorithm for UFP in trees both under the no-bottleneck assumption.

Lewin-Eytan et al. [12] studied the *admission control problem* in the tree topology. The problem instance in this case is similar to a BAP-instance. However, each request is also associated with a time interval. A feasible schedule is a set of connection requests such that at any given time, the total bandwidth requirement on every edge in the tree is at most 1. The goal is to find a feasible schedule with maximum total weight. Clearly, the admission control problem in trees is a generalization of BAP. Lewin-Eytan et al. [12] presented a divide and conquer $(5 \log n)$-approximation algorithm for admission control in trees. It divides the set of requests using the temporal dimension, and conquers a set of requests whose time intervals overlap using a local ratio 5-approximation algorithm. This is in fact a 5-approximation algorithm for BAP in the general tree topology.

Our results. We consider BAP in the tree topology where the maximum degree of a vertex in the given tree is a constant. For wide instances of BAP we provide a polynomial time dynamic programming algorithm that extends the algorithms from [8, 9]. We present an approximation algorithm for narrow instances of BAP in general trees (respectively, in lines) with ratio $\frac{\sqrt{e}}{\sqrt{e}-1} + \varepsilon$ (respectively, $\frac{e}{e-1} + \varepsilon$), for every $\varepsilon > 0$. This algorithm is based on a novel application of the *local ratio technique* in which the available bandwidth is divided into narrow strips, and requests with very small demands are allocated in these strips. By combining the two algorithms we get an approximation algorithm for BAP in bounded degree trees with ratio $\frac{2\sqrt{e}-1}{\sqrt{e}-1} + \varepsilon < 3.542$ (respectively, $\frac{2e-1}{e-1} + \varepsilon < 2.582$). We also present a randomized $(1 + \varepsilon)$-approximation algorithm, for every $\varepsilon > 0$, for narrow instances of BAP in bounded degree trees that extends the $(1 + \varepsilon)$-approximation algorithm for BAP in the line from [9]. This implies a randomized $(2 + \varepsilon)$-approximation algorithm for BAP in bounded degree trees.

For wide instances of SAP in bounded degree trees we provide a polynomial time dynamic programming algorithm that extends the algorithm from [8]. We present a reduction from SAP to BAP that works on very narrow instances in the line topology. The reduction is based on an algorithm for the *dynamic storage allocation problem* by Buchsbaum et al. [13]. This reduction implies a $(2+\varepsilon)$-approximation algorithm for SAP in the line.

2 Preliminaries

Definitions and Notation. We denote the optimum of a given problem instance by OPT. Given a schedule S, we denote by $w(S)$ the total weight of S, i.e., $w(S) = \sum_{j \in S} w(j)$.

Throughout the paper we assume that the given tree T is rooted, and we denote the root by r. We also assume that the maximum degree of a vertex in T is a constant. We denote the maximum degree by Δ, i.e., $\Delta = \max_u \deg(u)$.

The *peak* of a request j is the vertex in jth path that is closest to the root r. We denote the peak of j by $\text{peak}(j)$. We denote by $E(j)$ the set of edges in P_j that are incident on $\text{peak}(j)$. $E(j)$ contains either two edges or one edge. We define a partial order on the requests as follows. For requests j and ℓ we write $j \prec \ell$ if $\text{peak}(j)$ is an ancestor of $\text{peak}(\ell)$. We denote by $A(\ell)$ the set of requests j such that $\text{peak}(j)$ is an ancestor of $\text{peak}(\ell)$, i.e., $A(\ell) = \{j : j \prec \ell\}$.

Henceforth, we assume that the requests are topologically ordered according to the partial order. That is, we assume that if $j < k$, then $k \not\prec j$. In other words, $j < k$ if $\text{peak}(k)$ is not found on the path for $\text{peak}(j)$ to the root r.

Observation 1. *Let ℓ be a request, and let $S \subseteq J$ be a feasible solution such that $\ell \not\prec j$ for every $j \in S$. Then, $S \cup \{\ell\}$ is a feasible solution if the load on e is at most $1 - d_\ell$ for every $e \in E(\ell)$.*

Narrow and Wide Instances. Given a parameter $\delta \in (0,1)$, we can divide a given instance into a narrow instance and a wide instance. We denote the corresponding sets of requests by R_N and R_W, respectively.

Lemma 1. *Let S_N and S_W be an r_1-approximate solution with respect to R_N and a r_2-approximate solution with respect to R_W, respectively. Then, the solution of greater weight is an $(r_1 + r_2)$-approximation for the original instance.*

Proof. Let S^* be an optimal solution for the original instance. Either $w(S^* \cap R_N) \geq \frac{r_1}{r_1+r_2} w(S^*)$ or $w(S^* \cap R_W) \geq \frac{r_2}{r_1+r_2} w(S^*)$. Hence, either $w(S_N) \geq \frac{1}{r_1}\frac{r_1}{r_1+r_2} w(S^*) = \frac{1}{r_1+r_2} w(S^*)$ or $w(S_W) \geq \frac{1}{r_2}\frac{r_2}{r_1+r_2} w(S^*) = \frac{1}{r_1+r_2} w(S^*)$. □

The Local Ratio Technique. The local ratio technique [14, 15, 16, 7] is based on the Local Ratio Theorem, which applies to optimization problems of the following type. The input is a non-negative weight vector $w \in \mathbb{R}^n$ and a set of feasibility constraints $\mathcal{F}$. The problem is to find a solution vector $x \in \mathbb{R}^n$ that maximizes (or minimizes) the inner product $w \cdot x$ subject to the constraints $\mathcal{F}$.

Theorem 1 (Local Ratio [7]). *Let $\mathcal{F}$ be a set of constraints and let w, w_1, and w_2 be weight vectors such that $w = w_1 + w_2$. Then, if x is r-approximate both with respect to $(\mathcal{F}, w_1)$ and with respect to $(\mathcal{F}, w_2)$, for some r, then x is also an r-approximate solution with respect to $(\mathcal{F}, w)$.*

3 Bandwidth Allocation

In this section we consider the bandwidth allocation problem in bounded degree trees. For the special case of wide instances we present a polynomial time dynamic programming algorithm that computes optimal solutions. For the special case of narrow instances we present a deterministic approximation algorithm whose ratio is $1/(1 - 1/\sqrt{e} - \varepsilon) < 2.542$. We note that the algorithm for narrow instances works even in the case of general trees. In the line topology the approximation ratio of this algorithm is $1/(1 - 1/e - \varepsilon) < 1.582$. By Lemma 1 it follows that there is a 3.542-approximation algorithm for BAP in bounded degree trees, and a 2.582-approximation algorithm for BAP in the line topology. For narrow instances we also present a randomized LP-based $(1 + \varepsilon)$-approximation algorithm that extends the $(1 + \varepsilon)$-approximation algorithm by Calinescu et al. [9] for narrow instances of BAP in the line. Using the dynamic programming algorithm from Section 3 it follows from Lemma 1 that there is a randomized $(2 + \varepsilon)$-approximation algorithm for BAP on bounded degree trees.

3.1 Dynamic Programming Algorithm for Wide Instances

We present a polynomial time dynamic programming algorithm for BAP on wide instances and in bounded degree trees. This algorithm extends the algorithms for BAP in the line topology by Chen et al. [8] and by Calinescu et al. [9].

In order to solve the problem on wide instances we consider a variation of BAP in which we are given a constant L that limits the number of requests per edge. We present a dynamic programming algorithm whose running time is $O(m \cdot n^{\Delta \cdot L})$. Clearly, if S is a feasible solution for a wide instance, then there are at most $1/\delta$ requests in S that go through e for any edge e. Hence, the running time of this algorithm is $O(m \cdot n^{\Delta/\delta})$.

We use the following notation. Consider a vertex u_0 in the tree. Let T_0 be the subtree whose root is u_0, and let e_0 the edge that is going from u_0 to its parent. u_0's children are denote by $u_1, \ldots, u_k$. Also, let e_i be the edge connecting u_i and u_0 for $i \in \{1, \ldots, k\}$. See example in Figure 1. Using this notation, we refer to a set of requests S_i as *proper with respect to a vertex* u_i if (1) $e_i \in P_j$ for every $j \in S_i$, (2) $\sum_{j \in S_i : e_i \in P_j} d_j \leq 1$, and (3) $|S_i| \leq L$. Given a proper set S_0 with respect to u_0, the sets $S_1, \ldots, S_k$ are said to be *compatible* with S_0 if (1) S_i is proper with respect to u_i for every i, and (2) for every j and $i, i' \in \{0, \ldots, k\}$, if $e_i, e_{i'} \in P_j$, then either $j \in S_i, S_{i'}$ or $j \notin S_i, S_{i'}$.

The dynamic programming table is of size $O(m \cdot n^L)$, and it is defined as follows. For a vertex u_0 and a set of requests S_0 that is proper with respect to u_0, the state $\Pi(u_0, S_0)$ is the maximum weight of a set $S' \subseteq J(T_0)$, where $J(T_0)$

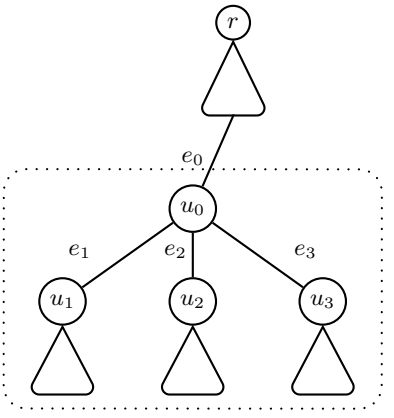


Fig. 1. u_0 and its children; T_0 is marked by the dotted line

contains requests j such that P_j is fully contained in T_0, such that $S_0 \cup S'$ is feasible. We initialize the table by setting $\Pi(u_0, S_0) = 0$ for every leaf u_0 and a proper set S_0. We compute the rest of the entries by using:

$$\Pi(u_0, S_0) = \max_{S_1,\ldots,S_k \text{ are compatible with } S_0} \left\{ w(\cup_{i=1}^{k} S_k \setminus S_0) + \sum_{i=1}^{k} \Pi(u_i, S_i) \right\}$$

when u_0 is an internal node. The weight of an optimal solution is $\Pi(r, \emptyset)$.

We show that the running time of the dynamic programming algorithm is $O(m \cdot n^{L \cdot \Delta})$. To compute each entry $\Pi(u_0, S_0)$ we need to go through all the possibilities of sets $S_1, \ldots, S_k$ that are compatible with S_0. There are no more than $\sum_{i=1}^{L} \binom{n}{i} = O(n^L)$ possibilities of choosing a proper set S_i that is compatible with $S_0, \ldots, S_{i-1}$. Hence, the number of possibilities is $O(n^{L \cdot (\Delta - 1)})$. Hence, the total running time is $O(m \cdot n^L n^{L \cdot (\Delta - 1)}) = O(m \cdot n^{L \cdot \Delta})$.

Note that the computation of $\Pi(u_0, S_0)$ can be modified so as to compute a corresponding solution. This can be done by keeping track on which option was taken in the recursive computation. Afterwards an optimal solution can be reconstructed in a top down manner.

3.2 Local Ratio Algorithm for Narrow Instances

In this section we consider the special case of narrow instances. For this case we present a deterministic approximation algorithm whose ratio is $1/(1-1/\sqrt{e}-\varepsilon) < 2.542$. In the line topology the ratio is $1/(1-1/e-\varepsilon) < 1.582$.

Scheduling Requests in Layers. Throughout this section we assume that all the requests in the given instance are δ-narrow for some small constant $\delta > 0$. Let α be a constant such that $\delta < \alpha \leq 1$. We assume that α is significantly larger than δ. In this section we show how to construct an approximate solution that

uses at most α of the capacity of every edge in the given tree. Henceforth we refer to a tree whose edges has capacity α as an α-*layer*, or simply a *layer*. (Note that, in general, α may be larger than one.) Using these terms, in this section we present an algorithm that computes a solution that resides in an α-layer. We note that the approximation ratio of the algorithm is with respect to the original problem in which the capacity of the edges is 1.

Algorithm **Layer** is a local ratio algorithm that computes a $(1 + 2/(\alpha - \delta))$-approximate solution S such that the total demand of requests in S on any edge is at most α. Algorithm **Layer** is recursive and works as follows. If there are no requests, then it returns $\emptyset$. Otherwise, it chooses a request ℓ such that $\ell \not\prec j$ for every $j \neq \ell$. This can be done by choosing a request whose peak is furthest away from the root. It constructs a new weight function w_1, and solves the problem recursively on $w_2 = w - w_1$ and the set of jobs with positive weight that is denoted by J^+. Note that $\ell \notin J^+$. Then, it adds ℓ to the solution that was computed recursively only if feasibility is maintained.

Algorithm 1. Layer(J, w)

1: **if** $J = \emptyset$ **then** return $\emptyset$
2: Let $\ell \in J$ be a request such that $\ell \not\prec j$ for every $j \in J \setminus \{\ell\}$
3: Define $w_1(j) = w(\ell) \cdot \begin{cases} 1 & j = \ell, \\ \frac{d_j}{\alpha - \delta} & j \neq \ell, P_j \cap P_\ell \neq \emptyset, \\ 0 & \text{otherwise,} \end{cases}$ and $w_2 = w - w_1$
4: Let J^+ be the set of positive weighted requests
5: $S' \leftarrow$ **Layer**(J^+, w_2)
6: **if** $\sum_{j \in S' : e \in P_j} d_j \leq \alpha - d_\ell$ for every $e \in E(\ell)$ **then** $S \leftarrow S' \cup \{\ell\}$
7: **else** $S \leftarrow S'$
8: return S

Observe that due to Lines 6–7 and Obs. 1 the total demand of requests from the computed solution on any edge is at most α. Also, the running time of the algorithm is clearly polynomial, since the number of recursive calls is at most n. In fact, using similar arguments to those used in [7] it can be implemented to run in $O(n \log n)$ time.

Lemma 2. *Algorithm* ***Layer*** *computes a* $(1 + \frac{2}{\alpha - \delta})$*-approximate solution* S *that resides within an* α*-layer.*

Proof. The proof is by induction on the number of recursive calls. In the base case ($J = \emptyset$) the computed solution is optimal. For the inductive step, we assume that $w(S')$ is $(1 + \frac{2}{\alpha - \delta})$-approximate with respect to J^+ and w_2. If this is the case, then S is also $(1 + \frac{2}{\alpha - \delta})$-approximate with respect to J and w_2, since $w_2(\ell) = 0$. We show that S is also $(1 + \frac{2}{\alpha - \delta})$-approximate with respect to J and w_1. Due to Lines 6-7 either $\ell \in S$ or $S \cup \{\ell\}$ is infeasible. If $\ell \in S$, then $w_1(S) \geq w(\ell)$. Otherwise, $\sum_{j \in S' : e \in P_j} d_j > \alpha - d_\ell$ for some edge $e \in E(\ell)$,

and therefore $w_1(S) \geq w(\ell) \cdot \frac{\alpha - d_\ell}{\alpha - \delta} \geq w(\ell)$. On the other hand, we show that $w_1(T) \leq w(\ell) \cdot (1 + 2/(\alpha - \delta))$, for every solution T. Let $j \in T$ be a request whose path intersects P_ℓ. Since $\ell \not\prec j$ for every $j \in J$ either peak(j) is an ancestor of peak(ℓ) or $peak(j) = peak(\ell)$. Hence, P_j contains at least one edge from $E(\ell)$. It follows that

$$w_1(T) \leq w(\ell) \cdot \max\{1 + 2(1 - d_\ell)/(\alpha - \delta), 2/(\alpha - \delta)\} \leq w(\ell) \cdot (1 + 2/(\alpha - \delta))$$

which means that S is $(1 + \frac{2}{\alpha - \delta})$-approximate with respect to J and w_1. This completes the proof since by the Local Ratio Theorem we get that S is $(1 + \frac{2}{\alpha - \delta})$-approximate with respect to J and w as well. □

When the given tree is a line we may choose one of the leafs to be the root, and in this case $|E(\ell)| = 1$. Hence, in Lemma 2, it can be shown that S is $(1 + \frac{1}{\alpha - \delta})$-approximate with respect to J and w_1. It follows that Algorithm **Layer** computes $(1 + 1/(\alpha - \delta))$-approximate solutions that reside within an α-layer.

Iterative Approximation in Layers. Algorithm **Multi-Layer** iteratively use Algorithm **Layer** with $\alpha = 1/k$ to schedule requests in $1/k$-layers for some large constant k, such that δ is significantly smaller than $1/k$.

Algorithm 2. Multi-Layer(J, w)

1: $J_1 \leftarrow J$
2: **for** $i = 1$ to k **do**
3: $\quad S_i \leftarrow$ **Layer**(J_i, w)
4: $\quad J_{i+1} \leftarrow J_i \setminus S_i$
5: **end for**
6: Return $\cup_{i=1}^{k} S_i$

Algorithm **Multi-Layer** computes feasible solutions, since Algorithm **Layer** computes feasible solutions each residing in a $1/k$-layer. The running time is polynomial, since Algorithm **Layer** is invoked a constant number of times.

We use the following notation. Let S^* be an optimal solution to the original instance. Let OPT_i denote the optimal value with respect to J_i. We denote $F_i = \cup_{l=1}^{i} S_l$. Hence, F_k is the computed solution. We also define $r \triangleq 1 + \frac{2}{1/k - \delta}$.

Lemma 3. $w(F_k) \geq (1 - (1 - 1/r)^k) \cdot \text{OPT}$.

Proof. We show that $w(F_i) \geq (1 - (1 - 1/r)^i) \cdot \text{OPT}$ for every i. We prove the claim by induction on i. The base case ($i = 0$) is trivial. For the inductive step, we assume that $w(F_{i-1}) \geq (1 - (1 - 1/r)^{i-1}) \cdot \text{OPT}$. The set $S^* \setminus F_{i-1}$ is feasible with respect to $J_i = J \setminus F_{i-1}$, and therefore, $\text{OPT}_i \geq w(S^* \setminus F_{i-1})$. By Lemma 2 it follows that S_i is r-approximate with respect to J_i. Hence,

$$w(S_i) \geq \frac{\text{OPT}_i}{r} \geq \frac{w(S^* \setminus F_{i-1})}{r} \geq \frac{w(S^*) - w(F_{i-1})}{r} = \frac{\text{OPT} - w(F_{i-1})}{r} .$$

It follows that $w(F_i) = w(F_{i-1}) + w(S_i) \geq (1 - 1/r) \cdot w(F_{i-1}) + \text{OPT}/r$. Putting it together with the induction hypothesis we get that

$$w(F_i) \geq (1 - 1/r) \cdot (1 - (1 - 1/r)^{i-1}) \cdot \text{OPT} + \text{OPT}/r = \text{OPT} \cdot (1 - (1 - 1/r)^i)$$

and the lemma follows. □

If δ is significantly smaller than $1/k$ it follows that $\lim_{k\to\infty,\delta\to 0}(1-1/r(\delta,k))^k) = 1/\sqrt{e}$. Hence, we may choose a sufficiently large constant k and then a sufficiently small constant δ such that $(1-1/r)^k \leq 1/\sqrt{e}+\varepsilon$. Hence, for every constant $\varepsilon > 0$ there exist $\delta > 0$ and k such that Algorithm **Multi-Layer** computes solutions that are $(1/(1-1/\sqrt{e}-\varepsilon))$-approximate. In the line we get a ratio of $1/(1-1/e-\varepsilon)$ by setting $r \triangleq 1 + \frac{1}{1/k-\delta}$.

3.3 Randomized Algorithm for Narrow Instances

In this section we present a randomized LP-based $(1 + \varepsilon)$-approximation algorithm that extends the $(1 + \varepsilon)$-approximation algorithm by Calinescu et al. [9] for BAP in the line topology.

BAP can be formalized as follows:

$$\begin{array}{lll} \max & \sum_{j\in R} w(j)x_j & \\ \text{s.t.} & \sum_{j\,:\,e\in P_j} d_j x_j \leq 1 & \forall e \in E \\ & x_j \in \{0,1\} & \forall j \in J \end{array} \qquad \text{(IP-BAP)}$$

The LP-relaxation of IP-BAP is obtained by replacing the integrality constraints by: $0 \leq x_j \leq 1$ for every $j \in J$, and is denoted by LP-BAP.

Next, we present a randomized approximation algorithm for narrow instances of BAP. Specifically, we show that for every $\varepsilon < 1/6$ there exists $\delta > 0$ small enough such that the algorithm computes $1/(1-6\varepsilon)$-approximate solutions. The approximation algorithm is described as follows. First, we solve LP-BAP. Denote by x^* the computed optimal solution, and let $\text{OPT}^* = \sum_{j\in J} w(j)x^*_j$. We choose independently at random the variables $Y_j \in \{0, 1\}$, for $j \in J$, where $\Pr[Y_j = 1] = (1 - \varepsilon)x^*_j$. Next we define the random variables $Z_j \in \{0, 1\}$, $j \in J$. The Z_js are considered in a top down manner. That is, Z_j is defined only after Z_ℓ was defined for every $\ell \prec j$. The Z_js are defined as follows:

$$Z_j = \begin{cases} 1 & \text{if } Y_j = 1 \text{ and } \sum_{i\,:\,Z_i=1\wedge e\in P_i} d_i \leq 1 - d_j \text{ for every } e \in E(j), \\ 0 & \text{otherwise.} \end{cases}$$

Notice that the Z_js are dependent, and they can be computed in the order $Z_1, \ldots, Z_n$ since if $i < j$ then $j \notin A(i)$ (or, $j \not\prec i$).

Let $Z = \{j : Z_j = 1\}$. Z is a feasible solution, and $\mathrm{E}[w(Z)] = \sum_{j\in J} w(j) \cdot \Pr[Z_j = 1]$. In the full version of the paper we show that $\Pr[Z_j = 1] \geq (1-3\varepsilon)x^*_i$. It follows that $\mathrm{E}[w(Z)] \geq \sum_{j\in J} w(j) \cdot (1 - 3\varepsilon)x^*_j = (1 - 3\varepsilon)\text{OPT}^*$. Furthermore, Markov's inequality implies that $\Pr[w(Z) \geq (1 - 6\varepsilon) \cdot \text{OPT}^*] \geq 1/2$. Hence, we can use repetition in order to amplify the probability of obtaining at least a weight of $(1 - 6\varepsilon) \cdot \text{OPT}^*$.

4 Storage Allocation

In this section we consider SAP in the line. For the special case of wide instances we present polynomial time dynamic programming algorithm that computes optimal solutions. Note that this algorithm works even in the case of bounded degree tree. For narrow instances of SAP in the line we present a general reduction from SAP to BAP. That is, given a narrow SAP instance in the line we show how to find an approximate solution using an algorithm for BAP. Thus, using the $(1+\varepsilon)$-approximation algorithm for narrow instances of BAP in the line from [11] we may obtain a $(1+\varepsilon)$-approximation algorithm for narrow instances of SAP. By Lemma 1, there exists a $(2+\varepsilon)$-approximation algorithm for SAP, for any $\varepsilon > 0$.

4.1 Storage Allocation on Wide Instances

We show how to extend the dynamic programming algorithm from Sect. 3.1 to solve wide instances of SAP in bounded degree trees. The running time of the modified algorithm is $O(m \cdot n^{2L \cdot \Delta})$, where L is an upper bound on the number of requests per edge. If S is a feasible solution for a wide instance of SAP, then there are at most $1/\delta$ requests in S that go through e for any edge e. Hence, the running time of this algorithm in the line topology is $O(m \cdot n^{4/\delta})$, where m is the number of edges in the line.

Our algorithm is based on the following simple observation.

Observation 2. *There exists an optimal solution such that, for every request j, either $h(j) = 0$ or there exists a request $j' \neq j$ such that $P_j \cap P_{j'} \neq \emptyset$ and $h(j) = h(j') + d_{j'}$.*

Proof. Given an optimal solution S^*, simply apply "gravity" on S^*. □

Let S^* be an optimal solution. By Obs. 2 we may assume that the height $h(j)$ of every request j is the sum of demands of some (possibly empty) subset of requests. Since S^* contains at most L requests per edge, the number of possible heights is bounded by $\sum_{i=0}^{L} \binom{n}{i} = O(n^L)$. Let H be the set of possible heights. In the full version of the paper we extend the definition of a proper set (see Sect. 3.1) to a *proper pair* (S_i, h_i), where h_i is a height function. Since H is of polynomial size, the number of possible proper pairs with respect to some vertex is polynomial as well. The rest of the details are given in the full version.

4.2 Reduction for Narrow Instances

Our reduction relies on a closely related problem to SAP called the *dynamic storage allocation problem* (DSA). Similarly to SAP in the line, in DSA we are given a set of rectangles that can only move vertically. The goal is to minimize the total height required to pack all rectangles such that no two rectangles overlap. Formally, a DSA solution is an assignment $h : S \to \mathbb{R}^+$ such that for every $j \neq k$ and $P_j \cap P_k \neq \emptyset$ either $h(j) + d_j \leq h(k)$ or $h(k) + d_k \leq h(j)$. Our goal is to minimize $\max_{j \in J} \{h(j) + d_j\}$.

We use the following result for DSA. Buchsbaum et al. [13] presented a polynomial time algorithm that computes a solution whose cost is at most $(1+O((\frac{D}{\text{LOAD}})^{1/7}))\cdot\text{LOAD}$, where $D = \max_j \{d_j\}$, and $\text{LOAD} = \max_e\{\sum_{j:e\in P_j} d_j\}$. Observe that when $D = o(\text{LOAD})$ the cost of the solution is $(1 + o(1)) \cdot \text{LOAD}$.

Lemma 4. *For every constant $\beta > 0$ there exists $\delta > 0$ such that if S is a feasible* BAP *solution to some δ-narrow instance, then S can be transformed into a* SAP *solution that fits into a $(1 + \beta)$-layer in polynomial time.*

Proof. Let S' be the solution computed by algorithm of Buchsbaum et al. [13]. It follows that $\text{LOAD}(S') \leq \text{LOAD}(S)+C\cdot D^{1/7}\cdot\text{LOAD}(S)^{6/7}$ for some constant C. Since $\text{LOAD}(S) \leq 1$, there exists δ small enough such that $\text{LOAD}(S') \leq 1+\beta$. □

Our reduction uses the notion of an α-layer. Recall that an α-layer is a tree (or a line) in which the capacity of the edges is α. In the case of BAP, this means that if a solution fits into an α-layer then the total demand on every edge is at most α. In the case of SAP, such a solution S must satisfy the following constraints: (i) $h(j) + d_j \leq \alpha$ for every $j \in S$, and (ii) for every two requests $j, k \in S$ such that $j \neq k$ and $P_j \cap P_k \neq \emptyset$ either $h(j) + d_j \leq h(k)$ or $h(k) + d_k \leq h(j)$.

Let Algorithm **BAP** be an approximation algorithm for narrow BAP instances in the line such that for every $\varepsilon' > 0$ there exists $\delta > 0$ such that the algorithm computes $r/(1-\varepsilon')$-approximate solutions. Algorithm **SAP** is our approximation algorithm for narrow SAP instances in the line that uses Algorithm **BAP**. We show that for every $\varepsilon > 0$ there exists $\delta > 0$ such that it computes $r/(1 - \varepsilon)$-approximate solutions for δ-narrow instances. We assume that δ is small enough such that (i) Algorithm **BAP** computes $r/(1-\varepsilon/4)$-approximate solutions on δ-narrow instances, and (ii) the conditions of Lemma 4 are satisfied with $\beta = \varepsilon/4$. We also assume that $\delta < \varepsilon/4$. Algorithm **SAP** starts by calling Algorithm **BAP** in order to obtain a BAP solution. Using Lemma 4, it transforms this solution into a SAP solution that fits into a $(1+\beta)$-layer, where $\beta = \varepsilon/4$. Then, it removes a small part of it in order to obtain a feasible solution for SAP.

Algorithm 3. SAP(J, w)

1: $S \leftarrow$ **BAP**(J, w)
2: Compute an assignment h for S in a $(1 + \beta)$-layer using Lemma 4
3: Divide the $(1 + \beta)$-layer into β-layers
   Let S_i be the requests that intersect the ith layer
4: $k \leftarrow \text{argmin}_i w(S_i)$
5: $S' \leftarrow S \setminus S_k$
6: For $j \in S'$, let $h'(j) = \begin{cases} h(j) & h(j) < (k-1)\beta, \\ h(j) - \beta & h(j) \geq k\beta, \end{cases}$
7: Return (S', h')

The computed solution is feasible since the removal of S_k leaves one β-layer empty, and this allows us to condense the assignment h such that the remaining

requests fit in a layer of height 1. Furthermore, since $\delta < \varepsilon/4$ each request j can be contained in at most two β-layers. Hence, $\sum_i w(S_i) \leq 2w(S)$. It follows that $w(S_k) \leq 2w(S)/\lceil 1/\beta \rceil = 2w(S)/\lceil 4/\varepsilon \rceil < 3\varepsilon \cdot w(S)/4$ and therefore $w(S') > (1 - 3\varepsilon/4) \cdot w(S)$. Since $w(S) \geq \text{OPT} \cdot (1 - \varepsilon/4)/r$, it follows that $w(S') > (1 - 3\varepsilon/4)(1 - \varepsilon/4)/r \cdot \text{OPT} \geq (1 - \varepsilon)/r \cdot \text{OPT}$ as required. (Recall that the BAP optimum is at least as large as the SAP optimum.) Finally, the running time is polynomial, because given ε all parameters except n are constants.

References

1. Golumbic, M.C.: Algorithmic Graph Theory and Perfect Graphs. Academic Press (1980)
2. Tarjan, R.E.: Decomposition by clique separators. Discrete Mathematics **55** (1985) 221–232
3. Arkin, E.M., Silverberg, E.B.: Scheduling jobs with fixed start and end times. Discrete Applied Mathematics **18** (1987) 1–8
4. Bar-Noy, A., Canetti, R., Kutten, S., Mansour, Y., Schieber, B.: Bandwidth allocation with preemption. SIAM J. Comp. **28** (1999) 1806–1828
5. Phillips, C., Uma, R.N., Wein, J.: Off-line admission control for general scheduling problems. Journal of Scheduling **3** (2000) 365–381
6. Leonardi, S., Marchetti-Spaccamela, A., Vitaletti, A.: Approximation algorithms for bandwidth and storage allocation problems under real time constraints. In: 20th Conference on Foundations of Software Technology and Theoretical Computer Science. Volume 1974 of LNCS. (2000) 409–420
7. Bar-Noy, A., Bar-Yehuda, R., Freund, A., Naor, J., Shieber, B.: A unified approach to approximating resource allocation and schedualing. J. ACM **48** (2001) 1069–1090
8. Chen, B., Hassin, R., Tzur, M.: Allocation of bandwidth and storage. IIE Transactions **34** (2002) 501–507
9. Calinescu, G., Chakrabarti, A., Karloff, H.J., Rabani, Y.: Improved approximation algorithms for resource allocation. In: 9th International Integer Programming and Combinatorial Optimization Conference. Volume 2337 of LNCS. (2002) 401–414
10. Chakrabarti, A., Chekuri, C., Gupta, A., Kumar, A.: Approximation algorithms for the unsplittable flow problem. In: 5th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems. Volume 2462 of LNCS. (2002) 51–66
11. Chekuri, C., Mydlarz, M., Shepherd, B.: Multicommodity demand flow in a tree. In: 30th Annual International Colloquium on Automata, Languages and Programming. Volume 2719 of LNCS. (2003) 410–425
12. Lewin-Eytan, L., Naor, J., Orda, A.: Admission control in networks with advance reservations. Algorithmica **40** (2004) 293–403
13. Buchsbaum, A.L., Karloff, H., Kenyon, C., Reingold, N., Thorup, M.: OPT versus LOAD in dynamic storage allocation. SIAM J. Comp. **33** (2004) 632–646
14. Bar-Yehuda, R., Even, S.: A local-ratio theorem for approximating the weighted vertex cover problem. Annals of Discrete Mathematics **25** (1985) 27–46
15. Bafna, V., Berman, P., Fujito, T.: A 2-approximation algorithm for the undirected feedback vertex set problem. SIAM J. Disc. Math. **12** (1999) 289–297
16. Bar-Yehuda, R.: One for the price of two: A unified approach for approximating covering problems. Algorithmica **27** (2000) 131–144

Dynamic Algorithms for Graph Spanners

Surender Baswana

Max-Planck Institute for Computer Science,
Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany
sbaswana@mpi-sb.mpg.de

Abstract. Let $G = (V, E)$ be an undirected weighted graph on $|V| = n$ vertices and $|E| = m$ edges. For the graph G, A spanner with stretch $t \in \mathbb{N}$ is a subgraph (V, E_S), $E_S \subseteq E$, such that the distance between any pair of vertices in this subgraph is at most t times the distance between them in the graph G. We present simple and efficient dynamic algorithms for maintaining spanners with essentially optimal (expected) size versus stretch trade-off for any given unweighted graph. The main result is a decremental algorithm that takes expected $O(\text{polylog}\, n)$ time per edge deletion for maintaining a spanner with arbitrary stretch. This algorithm easily leads to a fully dynamic algorithm with sublinear (in n) time per edge insertion or deletion. Quite interestingly, this paper also reports that for stretch at most 6, it is possible to maintain a spanner fully dynamically with expected constant time per update. All these algorithms use simple randomization techniques on the top of an existing static algorithm [6] for computing spanners, and achieve drastic improvement over the previous best deterministic dynamic algorithms for spanners.

1 Introduction

A spanner is a (sparse) subgraph of a given graph that preserves approximate distance between each pair of vertices. More precisely, a t-spanner of a graph $G = (V, E)$, for any $t \geq 1$ is a subgraph $(V, E_S), E_S \subseteq E$ such that, for any pair of vertices, their distance in the subgraph is at most t times their distance in the original graph. The parameter t is called the *stretch factor* associated with the t-spanner. The concept of spanners was defined formally by Peleg and Schäffer [15] though the associated notion was used implicitly by Awerbuch [3] in the context of network synchronizers. Since then, spanner has found numerous applications in the area of distributed systems, communication networks and all pairs approximate shortest paths [3, 7, 16, 17].

Each application of spanners requires, for a specified $t \in \mathbb{N}$, a t-spanner of smallest possible size (the number of edges). Based on the famous girth conjecture by Erdős [11], Bollobás [8], and Bondy and Simonovits [9], it follows that for any $k \in \mathbb{N}$, there are graphs on n vertices whose $(2k-1)$-spanner or a $2k$-spanner will require $\Omega(n^{1+1/k})$ edges. (The conjecture has been proved for $k = 1, 2, 3$ and 5). Note that the conjectured lower bound is the same for stretch $2k$ and $(2k-1)$, and by definition, a $(2k-1)$-spanner is also a $2k$-spanner, Therefore, from perspective of an algorithmist, the aim would be to design a static (or dynamic)

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 76–87, 2006.

algorithm to compute (or maintain) a $(2k-1)$-spanner of $(n^{1+1/k})$ size for a given graph. For unweighted graphs, Halperin and Zwick [13] designed a deterministic $O(m)$ time algorithm to compute a $(2k-1)$-spanner of $O(n^{1+1/k})$ size. However, for weighted graphs, it took a series of improvements [1, 4, 10, 20, 6, 5] till an expected $O(m)$ time algorithm for computing a $(2k-1)$-spanner could be designed. This linear time randomized algorithm [6, 5] computes a $(2k-1)$-spanner of size $O(kn^{1+1/k})$ for a given weighted graph. Recently Roditty *et al.* [18] derandomized this algorithm.

In this paper, we consider the problem of efficiently maintaining a $(2k-1)$-spanner in a dynamic environment : Given a graph $G = (V, E)$, we receive an online sequence of updates which could be insertions or deletions of edges, the aim is to maintain a data structure which stores a $(2k-1)$-spanner for the graph at each moment and is very efficient to handle these updates. It is also desirable that the algorithm ensures $O(n^{1+1/k})$ size of the $(2k-1)$-spanner after each update.

Previous work

Ausiello *et al.* [2] are the first to design dynamic algorithms for spanners. They present dynamic algorithms for maintaining spanners with stretch at most 6 only. They first design an $O(n)$ time decremental algorithm, and then employ the idea of handling insertions in a lazy fashion to design a fully dynamic algorithm with $O(n)$ time per update. The spanners maintained are of optimal size. However, the worst case space requirement of the associated data structure is $\theta(n^2)$. They extend their algorithm to weighted graphs with at most d different weights by maintaining separate spanner for the set of edges with the same weight. This leads to an increase in the size of the spanner and the update time by a factor of d.

New results

1. *Decremental algorithm*
   We present a partial dynamic algorithm for maintaining a $(2k-1)$-spanner under deletion of edges for any $k \in \mathbb{N}$. Our algorithm ensures an expected $O(kn^{1+1/k})$ size for the $(2k-1)$-spanner and the expected update time required is $O(\text{polylog}\, n)$ per edge deletion. We employ the static algorithm [6], and overcome a few subtle problems in dynamizing it by introducing a new clustering of vertices. Our algorithm also leads to an efficient decremental algorithm for all-pairs approximate shortest paths (see Corollary 1).
2. *Fully dynamic algorithms*
   We make our decremental algorithm fully dynamic by handling the edge insertions in a lazy fashion and rebuilding the entire data structure after a period of $kn^{1+1/k}$ edge insertions. This leads to a fully dynamic algorithm for a $(2k-1)$-spanner with amortized $\tilde{O}(\frac{m}{n^{1+1/k}})$ time per edge insertion/deletion. The expected size of the $(2k-1)$-spanner maintained by the algorithm is $O(kn^{1+1/k})$.

   We also show that a fully dynamic algorithm for stretch at most 6, can be maintained with expected constant update time and expected optimal

size. This algorithm follows by adding additional randomization to the static algorithm of [6], followed by dynamizing it. However, there are some potential difficulties in extending the algorithm for arbitrary stretch. Nevertheless, it is worth exploring whether the result can be extended for arbitrary stretch.

Our algorithms can be extended to weighted graphs with d different weights in the same way as done by Ausiello *et al.* [2]. For these graphs, the bounds on the spanner size and the update time of our algorithm will increase by a factor of d. All our algorithms require $\theta(m)$ space, which is much better than the $\theta(n^2)$ space requirement of [2].

2 Preliminaries

Throughout the paper, we deal with graphs which are undirected and unweighted. We assume that the vertices are numbered from 1 to n. We shall maintain the set of edges of the graph using a dynamic hash table (see [14]). Using this hash table, it requires $O(1)$ worst-case time for lookup and $O(1)$ expected time for any insertion and deletion. The space occupied by the hash table at any moment will be of the order of the number of edges present in the graph at that moment. Each edge of the graph will have a field to denote whether or not it is a spanner edge at that moment of time. We also assume without loss of generality that $m = \Omega(kn^{1+1/k})$, since otherwise for maintaining a $(2k-1)$-spanner we just keep all the edges in the $(2k-1)$-spanner and just update the hash table for edge insertion or deletion. The distance between any two vertices in not merely a function of the edges in their local neighborhood. However, the task of maintaining a spanner - a sparse set of edges that approximates all pairs distances - can be achieved by ensuring the following somewhat local proposition for each non-spanner edge (x, y).

$\mathcal{P}_t(x, y)$: the vertices x and y are connected in the subgraph (V, E_S) by a path consisting of at most t edges

In order to maintain a t-spanner in dynamic scenario, it suffices to maintain $\mathcal{P}_t$ for each non-spanner edge. This will be achieved by a careful partitioning of vertices, called clustering [6].

Definition 1. *A* **cluster** *is a subset of vertices. A* **clustering** $\mathcal{C}$*, is a union of disjoint clusters. Each cluster will have a unique vertex which will be called its* **center**. *A clustering can be represented by an array $C[]$ such that $C[v]$ for any $v \in V$ is the center of cluster to which v belongs, and $C[v] = 0$ if v is unclustered (does not belong to any cluster).*

The following notations will be used throughout the paper in the context of a given graph $G = (V, E)$, and $S, Y \subseteq V$.

- S_u : the set of vertices from S neighboring to u.
- $\delta(u, v)$: distance between u and v in the graph G.
- $\delta(u, Y)$: $\min\{\delta(u, v) \mid v \in Y\}$.

3 A Decremental $O(\text{polylog}\, n)$ Time Algorithm

3.1 New Clustering

Definition 2. *Given a permutation σ of some set $S \subseteq V$, and $i \in \mathbb{N}$, clustering $\mathcal{C}(\sigma, i)$ can be defined as follows.*

A vertex $u \in V$ with distance $\delta(u, S) \leq i$ is assigned to the cluster centered at the vertex in S nearest to u. In case of a tie, i.e., if there are multiple vertices at distance $\delta(u, S)$ from u, it is the nearest vertex that appears first in the permutation σ.

Note that the clustering $\mathcal{C}(\sigma, i)$ partitions only those vertices of the graph that are within distance i from S.

Efficient construction and maintenance: Given a permutation σ of some set $S \subseteq V$, and $i \in \mathbb{N}$, clustering $\mathcal{C}(\sigma, i)$ can be constructed in $O(m)$ time by algorithm described in Figure 1. A simple proof by induction on the distance from S shows that C stores the clustering $\mathcal{C}(\sigma, i)$. Moreover, the forest $\mathcal{F}$ spans each cluster by a tree rooted at its center such that for each vertex $v \in \mathcal{C}(\sigma, i)$, there is a path in $\mathcal{F}$ of length $\delta(v, S)$ connecting v to $C[v]$.

Let Q be a queue initialized to contain elements of S in the order as defined by σ.
Initially visited(v)=*false* $\forall v \in V$,
$C[s] = s\ \forall s \in S$, and $\mathcal{F} \leftarrow \emptyset$.
While not_empty(Q) **do**
{ $x \leftarrow$ **Dequeue**(Q);
For all $(x, y) \in E$ **do**
If visited(y)=*false*
{ visited(y) $\leftarrow$ *true*;
$C[y] \leftarrow C[x]$;
$\mathcal{F} \leftarrow \mathcal{F} \cup \{(x, y)\}$;
$\ell(y) \leftarrow \ell(x) + 1$;
If $\ell(y) < i$ **Enqueue**(y)}}

Fig. 1. Computing $\mathcal{C}(\sigma, i)$

The clustering $\mathcal{C}(\sigma, i)$ and the forest $\mathcal{F}$ can be associated with a breadth first search (BFS) tree in an augmented graph in the following way. If G' is a graph formed by adding a dummy vertex g and the edges $\{(g, s) | s \in S\}$ in G, then $\mathcal{F} \cup \{(g, s) | s \in S\}$ is a BFS tree rooted at g in G'. This BFS tree (not necessarily a unique one) satisfies the following condition. *Every vertex $v \in \mathcal{C}(\sigma, i)$ lies in the subtree rooted at $C[v]$.* Maintaining the clustering amounts to maintaining a BFS tree which also satisfies this condition at all times. An arbitrary BFS tree of depth i can be maintained in total $O(mi)$ cost over any sequence of edge deletions [12]. The algorithm involves finding new depth of each vertex $v \in V$ whose depth has increased, and then hooking it to any arbitrary neighbor from the level just above it. To maintain the clustering $\mathcal{C}(\sigma, i)$, we need to maintain a BFS tree wherein we hook a vertex v to its appropriate neighbor to satisfy the condition stated above. For this we need to maintain a search data structure for every vertex storing the centers of the clusters to which its neighbors belong. This will lead to $O(mi \log n)$ total update time over any sequence of edge deletions.

Lemma 1. *Given a graph $G = (V, E)$, an integer i, a permutation σ of a set $S \subseteq V$, we can maintain the clustering $\mathcal{C}(\sigma, i)$ and its spanning forest $\mathcal{F}$ with amortized $O(i \log n)$ time per edge deletion.*

The decremental algorithm employs a k-level hierarchy of clusterings: $\{\mathcal{C}(\sigma_i, i) | i \leq k\}$ whose defining sets S_i's and the permutations σ_i's are computed as follows.

1. Let $S_0 \leftarrow V$, $S_k = \emptyset$. For $0 < i < k$, let S_i contain each element of set S_{i-1} independently with probability $n^{-1/k}$.
2. For $0 \leq i < k$ let σ_i be a uniformly random permutation of set S_i.

3.2 Decremental Algorithm

Our decremental algorithm for $(2k-1)$-spanner will maintain the following structures and functions which can be initialized in $\tilde{O}(m)$ time easily.

1. **Clustering $\mathcal{C}(\sigma_i, i), i < k$:** Let C_i stores the clustering at level i, and $\mathcal{F}_i$ be its spanning forest. We maintain arrays C_i's and the set $\mathcal{F} = \cup_i \mathcal{F}_i$.
2. **Highest levels of vertices:** Let $H[v]$ be the highest level l such that v is present in $\mathcal{C}(\sigma_l, l))$. We maintain $H[v], \forall v \in V$.
3. **Assignment of edges to appropriate levels and endpoints:** Each edge $(u, v) \in E$ is kept at level $i = \max(H[u], H[v])$, and belongs to u if $H[u] \leq H[v]$ and to v otherwise. Let $E_i(u)$ denote the edges thus assigned to u at level i.
4. **Vertex-cluster Connectivity:** Let $E_i(u, o)$ be the set of edges from $E_i(u)$ which are incident from cluster centered at $o \in S_i$. Keep a set $\mathcal{E}$ which contains, for every $v \in V,\ i < k,\ o \in S_i$, one edge from $E_i(v, o)$.

Lemma 2. *The set $E_S = \mathcal{E} \cup \mathcal{F}$ is a $(2k-1)$-spanner of expected size $O(n^{1+1/k})$ for the graph at any time.*

Proof. We need to ensure that $\mathcal{P}_{2k-1}$ holds for every $(u, v) \notin E_S$. Let $(u, v) \in E_i(u, C_i[v])$ for some $i < k$. Vertex-cluster connectivity ensures that there must be an edge (u, w) satisfying $C_i[w] = C_i[v]$ which is present in $\mathcal{E}$. It follows from the clustering that v and w are connected in $\mathcal{F}_i$ by a path of length at most $2i$, so there is a path in E_S of length at most $2i + 1 < 2k - 1$ joining u and v. Hence $\mathcal{P}_{(2k-1)}(u, v)$ holds. For bounding the size, we bound the expected number of spanner edges contributed to $\mathcal{E}$ by any vertex v at any level $i < k$. Consider any clustering $\mathcal{C}(\sigma_i, i)$. With respect to any arbitrary set $E' \subseteq E$, let $c_1, ..., c_\ell$ be the clusters in this clustering which are neighboring to v. The vertex v will contribute at most one edge per neighboring cluster to $\mathcal{E}$, and a necessary (though not sufficient) condition for this contribution is that v is not present in any clustering of level $i + 1$ or higher. Given any clustering $\mathcal{C}(\sigma_i, i)$, the vertex v would appear in the clustering at next level only if the center of at least one of $c_1, ..., c_\ell$ is sampled. Therefore, it follows by elementary probability that the expected number of edges contributed by v at level i is at most $\ell \cdot (1 - n^{-1/k})^\ell + 1$, which is at most $n^{1/k}$ for any value of ℓ. It can be observed that this upper bound is derived for any set $E' \subseteq E$.

Data structures: In addition to the hash table storing all the edges of the graph and the usual adjacency lists for each vertex, we keep the following data structures. For each $v \in V$ and $i, 0 \leq i < k$, let $o_1, ..., o_\ell$ be the centers of the cluster which are adjacent to v through edges $E_i(v)$ allocated to v. Keep a search tree for the set $\{o_1, ..., o_\ell\}$ and the node associated with $o_j, 1 \leq j \leq \ell$ in this tree would store a doubly linked list storing the edges $E_i(v, o_j) \subseteq E_i(v)$ incident on v from cluster centered at o_j. Furthermore, each edge in the set $E_i(v, o)$ will keep a pointer to and from the entry in the hash table storing all the edges of the graph. These data structures will help in efficient maintenance of the structures and function mentioned above.

Deletion of an edge may cause two kinds of changes in $\mathcal{C}_i, i < k$: some vertices change their clusters within $\mathcal{C}_i$ and/or some vertices cease to belong to the clustering $\mathcal{C}_i$ forever. The former change alters vertex-cluster connectivity as follows. Let a vertex v move from cluster c to join another cluster c'. Let $w \in V$ be a neighbor of v. As a result of this movement, it might be that c is no longer adjacent to w, and the cluster c', which earlier might be non adjacent to w, has become adjacent to w. Hence vertex-cluster connectivity needs to be updated. We describe below a subroutine for handling this case. When a vertex v ceases to belong to a clustering we will reassign all the edges present at level i which have u as one endpoint to their new levels and endpoints, and update the vertex-cluster connectivity accordingly.

Change-cluster(v, o, o')
(*when* v *moves from cluster centered at* o *to cluster centered at* o' *in* $\mathcal{C}(\sigma_i, i)$)
For each neighbor w of v in the graph with $(v, w) \in E_i(w, o)$ do

1. Delete (v, w) from $E_i(w, o)$. If (v, w) was in $\mathcal{E}$, choose some other edge from $E_i(w, o)$ in $\mathcal{E}$ (unless $E_i(w, o) = \emptyset$ now).
2. Insert edge (v, w) to $E_i(w, o')$, and choose it in $\mathcal{E}$ if $E_i(w, o')$ was empty earlier.

Decremental algorithm for $(2k-1)$-spanner

Deletion of an edge (u, v) is processed as follows. Let (u, v) be present at level i, and belong to $E_i(u, o)$ where $o = \mathcal{C}_i[v]$. If $(u, v) \in \mathcal{E}$ delete it from $\mathcal{E}$, and select some other edge from $E_i(u, o)$ in $\mathcal{E}$ (unless $E_i(u, o) = \emptyset$ now). The edge (u, v) could be in $\mathcal{F}$ (simultaneously as well). In this case, deletion might cause change of the clusterings at various levels, and we handle it as follows.
For $i = 1$ to $k - 1$ do

1. Update the clustering $\mathcal{C}(\sigma_i, i)$. Let $\Delta \subseteq V$ be the set of vertices that changed their clusters within $\mathcal{C}_i$, and $U \subseteq V$ be the set of vertices that ceased to be member of $\mathcal{C}_i$.
2. For each vertex $x \in \Delta$ do
   Let x moved from cluster centered at o to cluster centered at o' in $\mathcal{C}_i$.
   Change-cluster(x, o, o')
3. For each vertex $x \in U$, reassign all the edges present at level i which have x as one endpoint to their new levels and endpoints, and update the vertex-cluster connectivity accordingly.

Lemma 3. *At any level i, a vertex changes its cluster expected $O(i \log n)$ times.*

Proof. Consider a vertex $v \in V$. On leaving a cluster in $\mathcal{C}(\sigma_i, i)$ whenever v joins the same cluster again, its distance from S_i must have increased. We shall now estimate the number of times a vertex changes its cluster within $\mathcal{C}(\sigma_i, i)$ while keeping $\delta(v, S_i) = d$, for some fixed $d \leq i$. Consider the first time when $\delta(v, S_i)$ becomes d. Fix any order in which the edges are being deleted. Corresponding to this order, let $o_1, ...o_\ell$ be the sequence of all the vertices of S_i at distance d from v at present, and arranged in the chronological order of their cessation from being at distance d from v. The number of times v changes its cluster while keeping $\delta(v, S_i) = d$ is the same as the number of clusters in this sequence which v joins during the period for which $\delta(v, S_i) = d$. The vertex v will join cluster centered at o_j if and only if o_j appears first among $\{o_j, ..., o_\ell\}$ in the permutation σ_i. Since σ_i is a uniformly random permutation of S_i, the probability of this event is $1/(\ell - j + 1)$. Hence the expected number of cluster changes for the vertex v while remaining at a fixed distance d from S_i is $\sum_{j=1}^{\ell} \frac{1}{\ell - j + 1} = O(\log n)$. Since the vertex v may change $\delta(v, S_i)$ at most i times before losing membership from the clustering $\mathcal{C}_i$, the lemma follows.

Analyzing the running time: When an edge is deleted from $\mathcal{E}$, it requires $O(\log n)$ cost to look for a replacement edge using the data structure amounting to a total of $O(m \log n)$ cost over any sequence of edge deletions. It follows from Lemma 1 that the total cost of maintaining clustering at any level over any sequence of edge deletions is $O(km \log n)$. The remaining cost incurred is for processing of the edges due to the changes in the clusterings and will be charged to the respective edges. An edge will be processed at most $2k$ times due to cessation of one of its endpoints from being member of clusterings. It follows from Lemma 3 that an edge will be processed expected $O(ki \log n)$ times due to change of clusters of its endpoints within any clustering since there are total k levels of clusterings. Using the data structures each processing of an edge costs $O(\log n)$ time. So the total expected cost charged to an edge throughout the algorithm is of the order of $\sum_{i<k} ik \log^2 n \leq k^2 \log^2 n = O(\text{polylog}\, n)$ since $k \leq \log n$. So we can conclude the following theorem.

Theorem 1. *Given a graph on n vertices undergoing edge deletions and $k \in \mathbb{N}$, we can maintain its $(2k-1)$-spanner of expected $O(kn^{1+1/k})$ size with expected $O(\text{polylog}\, n)$ time per edge deletion.*

Choosing $k = \log n$, we get the following corollary.

Corollary 1. *Given a graph on n vertices undergoing edge deletions, we can maintain all-pairs $O(\log n)$-approximate shortest paths with $O(\text{polylog}\, n)$ update time, $\tilde{O}(n)$ query time, and $O(m)$ space requirement.*

Roditty and Zwick [19] gave a decremental algorithm that maintains all-pairs $O(\log n)$-approximate shortest paths with $O(1)$-query time, $\tilde{O}(n)$ update time, and $O(m)$ space. For the scenario, where we want to minimize the update time at the expense of increased query time, our algorithm offers a better choice.

4 Fully Dynamic Algorithms

For a given graph (V, E), let $\mathcal{D}(V, E)$ be the data structure associated with our decremental algorithm for maintaining a $(2k-1)$-spanner. Our fully dynamic algorithm maintains $\mathcal{D}(V, E)$ and handles edge insertions in a lazy fashion by inserting the edge directly into the spanner and rebuilding the data structure $\mathcal{D}$ for the new graph periodically once there are $kn^{1+1/k}$ insertions. In this manner, cost of each insertion is $O(1)$. Using Theorem 1 the total update cost (including the initialization cost) for maintaining $\mathcal{D}$ is $O(m \operatorname{polylog} n)$ for each interval of rebuilding. So the fully dynamic algorithm achieves an amortized $\tilde{O}(m/n^{1+1/k})$ cost per edge insertion/deletion. We can thus conclude with the following theorem.

Theorem 2. *Given a graph on n vertices and $k \in \mathbb{N}$, we can maintain its $(2k-1)$-spanner of expected $O(kn^{1+1/k})$ size in fully dynamic environment with expected $\tilde{O}(\frac{m}{n^{1+1/k}})$ update time per edge insertion or deletion.*

5 A Fully Dynamic Algorithm for Small Stretch Spanners

We present a fully dynamic algorithm for 3-spanner. Our fully dynamic algorithm for 5-spanner is along similar lines, and we present a sketch of it. First we state a simple lemma whose proof follows from elementary probability.

Lemma 4. *Given a set A of ℓ elements, let S be a sample formed by selecting each element of set A independently with some probability. The following assertion holds :*

$$\forall a \in A, \qquad \mathbf{Pr}[a \in S \mid |S| = i] = \frac{i}{\ell}$$

The algorithm begins with the following preprocessing. A sample $S \subseteq V$ is formed by selecting each vertex independently with some probability p. During the whole algorithm, the set S serves as the set of the centers of clusters, and the clustering is essentially grouping each vertex $v \in V$ satisfying $S_v \neq \emptyset$ to some vertex from S_v. We now relabel the vertices of the graph so that the vertices of set S get labels which are a permutation of $1..S$. (This relabeling takes $O(m)$ time and is required as a minor technicality for sake of clarity of exposition of the algorithm).
The algorithm will maintain the following three invariants at each moment.

- $\mathcal{I}_1$: Each vertex $v \in V \backslash S$, with $S_v \neq \emptyset$, belongs to the the cluster centered at any of the vertex from S_v with equal probability.
- $\mathcal{I}_2$: Each vertex $v \in V \backslash S$ contributes all its edge to the spanner.
- $\mathcal{I}_3$: Each clustered vertex v has the edge $(v, C[v])$ and one edge to each of its neighboring clusters in the spanner.

The invariants $\mathcal{I}_2$ and $\mathcal{I}_3$ will ensure that the spanner has stretch 3 and expected size $O(n^{3/2})$, whereas the invariant $\mathcal{I}_1$ will play a key role in achieving expected constant time for handling any edge insertion/deletion.

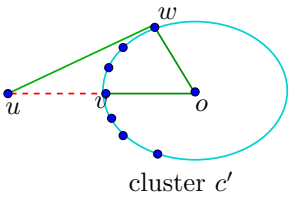


Fig. 2. For a non spanner edge, a stretch of 3

Note that all the edges incident on an unclustered vertex are included in the spanner. So let us consider an edge (u, v) which is not a spanner edge. In this case both u and v must be clustered. It follows from invariant $\mathcal{I}_2$ that a clustered vertex is connected to the center of its cluster by a spanner edge. If both u and v belong to the same cluster, there is a path between them consisting of two edges, both from the spanner. So consider the case when u and v belong to different clusters, say c and c' respectively (see Figure 2). The existence of the edge (u, v) in E shows that the cluster c' is neighboring to u, so $\mathcal{I}_3$ ensures that there must be some edge $(u, w), w \in c'$ in the spanner. This implies a path of three spanner edges between u and v (see Figure 2). Hence the spanner is surely a 3-spanner.

Now let us analyze the expected size of this 3-spanner. An unclustered vertex will contribute all its edges, whereas a clustered vertex will contribute one edge per incident cluster. Hence the expected number of edges contributed by a vertex will be at most $\deg(v) \cdot (1-p)^{\deg(v)} + np$, which is at most $1/p + np$. Hence the expected size of the 3-spanner is $O(n/p + n^2 p)$, which for $p = 1/\sqrt{n}$, is $O(n^{3/2})$. Hence we can conclude that

Lemma 5. *Maintaining invariants $\mathcal{I}_2$ and $\mathcal{I}_3$ for a dynamic graph ensures that the spanner is a 3-spanner of expected size $O(n^{3/2})$ at each stage.*

5.1 Data Structure

In order to efficiently maintain the invariants, we shall use the following data structures, which will require $O(m + n^{3/2})$ space, which is $O(m)$ since we have assumed that $m = \Omega(n^{3/2})$ (see Preliminaries).

- Let C be the array representing the clustering.
- Each vertex $v \in V$ keeps an array N_v of size $|S|$ such that $N_v[i]$ is (or points to) the head of a doubly linked list storing all those edges incident on v from a cluster centered at i. A node storing the edge (v, w) in this doubly linked list will also keep a pointer to (and from) the entry for the same edge in the hash table storing all the edges.
- Each vertex v maintains the set S_v of all the sampled vertices that are adjacent to it using a doubly linked list. A node storing w in this list will have a pointer to (and from) the node storing edge (v, w) in the list (pointed by) $N_v[C[w]]$. This will facilitate insertion/deletion of a vertex w from set S_v in $O(1)$ time whenever the corresponding edge (v, w) is inserted/deleted from the graph.

We shall employ the subroutine *Change-cluster* that we designed for our decremental algorithm. In addition, we shall use the following two subroutines.

Join-clustering(v, i) : (*an unclustered vertex* v *joins a cluster centered at* i)
$C[v] \leftarrow i$.
Process each clustered neighbor w of v as follows.
$j \leftarrow C[w]$.
Insert edge (v, w) to the lists $N_w[i]$ and $N_v[j]$.
Make (v, w) a non-spanner edge unless it is the only spanner edge present in either of $N_w[i]$ and $N_v[j]$.

Leave-clustering(v, i) : (*a vertex* v *clustered at* i *becomes unclustered*)
Process each clustered neighbor w of v as follows.
$j \leftarrow C[w]$.
Delete (v, w) from the lists $N_w[i]$ and $N_v[j]$, and make (v, w) a spanner edge.

Fully dynamic algorithm for 3-spanner

- **Deletion of an edge** (u, v) :
  If the edge (u, v) does not belong to the current spanner, it suffices to delete the edge from the data structures of u as well as v. So let us consider the situation when (u, v) is a spanner edge. If either u or v is an unclustered vertex, it also suffices to just delete the edge. Otherwise let u and v belong to clusters centered at i and j respectively. We process the vertex u as follows (the vertex v is processed in a similar manner).
  If $C[u] = v$
  { $S_u \leftarrow S_u \backslash \{v\}$;
  If $S_u = \emptyset$ { $C[v] \leftarrow 0$; *Leave-clustering*(u, v)}
  Else
  { let s be uniformly selected vertex from S_u;
  Make (u, s) a spanner edge;
  $C[u] \leftarrow s$; *Change-cluster*(u, i, s)}}
  Else delete the edge (u, v) from $N_u[j]$, choose another edge from $N_u[j]$ (unless $N_u[j] = \emptyset$ now), and make that a spanner edge.
- **Insertion of an edge** (u, v) :
  We process the vertex u as follows (the vertex v is processed similarly).
  If u is unclustered
  { Make (u, v) a spanner edge;
  If $v \in S$ { $C[u] \leftarrow v$; *Join-clustering*(u, v)}}
  Else
  { **If** v is clustered
  { $i \leftarrow C[u]$; $j \leftarrow C[v]$; Insert the edge (u, v) to $N_u[j]$;
  If ($i \neq j$ and $|N_u[j]| = 1$) make (u, v) a spanner edge;
  If $v \in S$
  { $S_u \leftarrow S_u \cup \{v\}$;
  With probability $1/|S_u|$ do
  { $C[u] \leftarrow v$; *Change-cluster*(u, i, j)}}}}.

It is easy to verify that the fully dynamic algorithm described above maintains the invariants $\mathcal{I}_1, \mathcal{I}_2$ and $\mathcal{I}_3$. The invariant $\mathcal{I}_1$ combined with Lemma 4 would imply the following crucial lemma whose proof follows by elementary probability.

Lemma 6. *For the graph G undergoing deletion and insertion of edges in any arbitrary order, our algorithm ensures the following equality at all times.*

$$\forall (u,v) \in E \quad \mathbf{Pr}[C[u] = v \mid u \text{ is clustered}] \;=\; \frac{1}{\deg(u)}$$

Analyzing the complexity of the algorithm
Consider deletion of an edge (u, v). It follows from the description of the algorithm that the processing of vertex u will take $O(1)$ time for all the cases except for the case when $C[u] = v$, in which case the update time is $O(\deg(u))$. Now applying Lemma 6 prior to deletion of edge (u, v), it follows that the probability of the latter case is $1/\deg(u)$. Hence the expected processing time for vertex u is $O(1)$ when an edge (u, v) is deleted. Similarly analyzing the vertex v, we conclude that an edge deletion can be processed in expected $O(1)$ time.

Now consider insertion of edge (u, v). It follows from the description of the algorithm that the update time is $O(1)$ for all the cases except when u gets assigned to cluster centered at v, in which case the update time is $O(\deg(u))$. Applying Lemma 6 just after the insertion (u, v), it follows that the probability of the latter case is $1/\deg(u)$. Hence the expected update time for maintaining a 3-spanner on inserting an edge (u, v) is constant. Combined with Lemma 5,

Theorem 3. *Given a graph on n vertices, we can maintain its 3-spanner of expected size $O(n^{3/2})$ with expected $O(1)$ time per edge insertion/deletion.*

5.2 Fully Dynamic Algorithm for Stretch 5 (or 6)

The algorithm is the same as the algorithm for 3-spanner except with the following modifications.

1. The sampling probability is $p = 1/n^{1/3}$.
2. **The data structure:** The data structure is identical to that of 3-spanner except that each cluster (instead of each vertex) c keeps an array N_c such that $N_c[c']$ is a doubly linked list storing the edges incident on c from c'.
3. **The invariants:** The fully dynamic algorithm will maintain the three invariants : The first two are just identical to $\mathcal{I}_1$ and $\mathcal{I}_2$ for the 3-spanner, while the invariant $\mathcal{I}_3$ is defined as : Each clustered vertex v has the edge $(v, C[v])$ in the spanner and each cluster has one spanner edge to each of its neighboring clusters.

With the slight difference in the data structure and the invariant $\mathcal{I}_3$ as described above, our fully dynamic algorithm for 5-spanner is identical to our fully dynamic algorithm for 3-spanner which we have described and analyzed in complete details earlier. Hence, we can state the following theorem.

Theorem 4. *Given a graph on n vertices, we can maintain its 5-spanner of expected size $O(n^{4/3})$ with expected $O(1)$ time per edge insertion/deletion.*

Acknowledgement. The author is grateful to Sandeep Sen and anonymous referees for their useful comments.

References

1. I. Althöfer, G. Das, D. P. Dobkin, D. Joseph, and J. Soares. On sparse spanners of weighted graphs. *Discrete and Computational Geometry*, 9:81–100, 1993.
2. G. Ausiello, P. G. Franciosa, and G. F. Italiano. Small stretch spanners on dynamic graphs. In *Proceedings of 13th Annual European Symposium on Algorithms*, volume 3669 of *LNCS*, pages 532–543. Springer, 2005.
3. B. Awerbuch. Complexity of network synchronization. *Journal of Ass. Compt. Mach.*, pages 804–823, 1985.
4. B. Awerbuch, B. Berger, L. Cowen, and D. Peleg. Near-linear time construction of sparse neighborhod covers. *SIAM Journal on Computing*, 28:263–277, 1998.
5. S. Baswana and S. Sen. A simple linear time randomized algorithm for computing sparse spanners in weighted graphs. *Random Structures and Algorithms (to appear)*.
6. S. Baswana and S. Sen. A simple linear time algorithm for computing a $(2k-1)$-spanner of $O(n^{1+1/k})$ size in weighted graphs. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 384–396, 2003.
7. S. Baswana and S. Sen. Approximate distance oracles for unweighted graphs in $\tilde{O}(n^2)$ time. In *Proceedings of the 15th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 271–280, 2004.
8. B. Bollobás. *Extremal Graph Theory.* Academic Press, 1978.
9. J. A. Bondy and M. Simonovits. Cycles of even length in graphs. *Journal of Combinatorial Theory, Series B*, 16:97–105, 1974.
10. E. Cohen. Fast algorithms for constructing t-spanners and paths with stretch t. *SIAM Journal on Computing*, 28:210–236, 1998.
11. P. Erdős. Extremal problems in graph theory. In *Theory of Graphs and its Applications (Proc. Sympos. Smolenice,1963)*, pages 29–36, Publ. House Czechoslovak Acad. Sci., Prague, 1964.
12. S. Even and Y. Shiloach. An on-line edge-deletion problem. *Journal of association for computing machinery*, 28:1–4, 1981.
13. S. Halperin and U. Zwick. Linear time deterministic algorithm for computing spanners for unweighted graphs. *unpublished manuscript*, 1996.
14. R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51:122–144, 2004.
15. D. Peleg and A. Schäffer. Graph spanners. *Journal of Graph Theory*, 13:99–116, 1989.
16. D. Peleg and J. D. Ullman. An optimal synchronizer for the hypercube. *SIAM Journal on Computing*, 18:740–747, 1989.
17. D. Peleg and E. Upfal. A trade-off between space and efficiency for routing tables. *Journal of Assoc. Comp. Mach.*, 36(3):510–530, 1989.
18. L. Roditty, M. Thorup, and U. Zwick. Deterministic construction of approximate distance oracles and spanners. In *Proceedings of 32nd International Colloquim on Automata, Languagaes and Programming (ICALP)*, volume 3580 of *LNCS*, pages 261–272. Springer, 2005.
19. L. Roditty and U. Zwick. On dynamic shortest paths problems. In *Proceedings of 12th Annual European Symposium on Algorithms (ESA)*, volume 3221 of *LNCS*, pages 580–591. Springer, 2004.
20. M. Thorup and U. Zwick. Approximate distance oracles. *Journal of Association of Computing Machinery*, 52:1–24, 2005.

Latency Constrained Aggregation in Sensor Networks*

Luca Becchetti[1], Peter Korteweg[2], Alberto Marchetti-Spaccamela[1], Martin Skutella[3], Leen Stougie[2,4], and Andrea Vitaletti[1]

[1] University of Rome "La Sapienza"
[2] TU Eindhoven
[3] University of Dortmund
[4] CWI Amsterdam

Abstract. A sensor network consists of sensing devices which may exchange data through wireless communication; sensor networks are highly energy constrained since they are usually battery operated. Data aggregation is a possible way to save energy consumption: nodes may delay data in order to aggregate them into a single packet before forwarding them towards some central node (sink). However, many applications impose constraints on data freshness; this translates into latency constraints for data arriving at the sink.

We study the problem of data aggregation to minimize maximum energy consumption under latency constraints on sensed data delivery and we assume unique transmission paths that form a tree rooted at the sink. We prove that the off-line problem is strongly NP-hard and we design a 2-approximation algorithm. The latter uses a novel rounding technique.

Almost all real life sensor networks are managed on-line by simple distributed algorithms in the nodes. In this context we consider both the case in which sensor nodes are synchronized or not. We consider distributed on-line algorithms and use competitive analysis to assess their performance.

1 Introduction

A sensor network consists of sensor nodes and one or more central nodes or *sinks*. Sensor nodes are able to monitor events, to process the sensed information and to communicate the sensed data. Sinks are powerful base stations which gather data sensed in the network; sinks either process this data or act as gateways to other networks. Sensors send data to the sink through multi-hop communication.

A particular feature of sensor nodes is that they are battery powered, making sensor networks highly energy constrained. Replacing batteries on hundreds of

* Supported by EU Integrated Project AEOLUS (FET-15964), EU project ADONET (MRTN-CT-2003-504438), EU COST-action 293, MIUR-FIRB project VICOM, Dutch project BRICKS, DFG Focus Program 1126, "Algorithmic Aspects of Large and Complex Networks", grant SK 58/5-3, MIUR-FIRB Israel-Italy project RBIN047MH9.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 88–99, 2006.

nodes, often deployed in inaccessible environments, is infeasible or too costly and, therefore, the key challenge in a sensor network is the reduction of energy consumption. Energy consumption can be divided into three domains: sensing, communication and data processing [1]. Communication is most expensive because a sensor node spends most of its energy in data transmission and reception [7]. This motivates the study of techniques to reduce overall data communication, possibly exploiting processing capabilities available at each node. Data aggregation is one such technique. It consists of aggregating redundant or correlated data in order to reduce the overall size of sent data, thus decreasing the network traffic and energy consumption.

Most literature on sensor networks assumes *total aggregation*, i.e. data packets are assumed to have the same size and aggregation of two or more incoming packets at a node results in a single outgoing packet. Observe that even if this might be considered a simplistic assumption, it allows us to provide an upper bound on the expected benefits of data aggregation in terms of power consumption. We refer here to a selection of papers, focused on the algorithmic side of the problem [3, 6, 10, 9, 11, 12]. However, these papers mainly focus on empirical and technical aspects of the problem.

We concentrate on data aggregation in sensor networks under constraints on the *latency* of sensed events; this means that data should be communicated to the sinks within a specified time after being sensed. Preliminary results are given in [8, 15]. In both cases formal proofs of the performance are not provided.

Time synchronization, in the sense of the existence of a common clock for the nodes, may or may not be a requirement of the sensor network. Therefore, we will consider both the *synchronous model* and the *asynchronous model*.

Contributions of the paper. A sensor network is naturally represented by a graph whose nodes are the sensors and the arcs the wireless communication links. Data aggregation, latency constraints and energy savings, give rise to a large variety of graph optimization problems depending on the following issues.

- Transmission energy and time can be seen as functions of the size of the packet and the transmission arc. Typically, these are concave functions exhibiting economies of scale in the size of the packets sent.
- The latency may depend on the (types of) sensor data or on the sensor nodes.
- Sensor networks can be modelled as synchronous or asynchronous systems.
- Data is delivered to one or more sinks.
- The overlay routing paths connecting nodes to the sinks can be fixed a priori, (e.g. a tree or a chain) or may also be chosen as part of the optimization process.
- There might be several objective functions; the most natural ones are to minimize the maximum energy consumption over all nodes or to maximize the amount of sensed data arriving at the sinks with a given energy constraint.

By considering the above issues, we formulate the sensor problem in a combinatorial optimization setting, which allows us to derive, what we believe to be, the first results on worst case analysis for on-line algorithms on wireless sensor networks, as opposed to mainly empirical current results.

We concentrate here on a basic subclass of latency constrained data aggregation problems. We assume that transit times and transit costs, in terms of energy consumption, are functions of the arcs only, modeling the situation of total aggregation, while the objective is to minimize the maximum transit cost per node over all nodes. There is only one sink and the transmission paths from the nodes to the sink are unique, forming an intree with the sink as the root. The tree is a typical routing topology in sensor networks; see [4, 6, 10, 13, 14].

In Section 2 we formalize the problem; for a thorough understanding of the problem we have studied both the off-line and the on-line version of the problem, although the latter version is the relevant one in practice.

In Section 3 we show that the off-line problem is NP-hard and we give a 2-approximate algorithm. We remark that our approximate solution is based on a new rounding technique of the LP-relaxation of an Integer Linear Programming formulation of the problem, which might be useful for other applications.

In Section 4 we describe the distributed on-line problem, both in the synchronous and the asynchronous settings. Our main results are:

(a) *Distributed synchronous.* We present a $\Theta(\log U)$-competitive algorithm, where U is the ratio between the maximum and the minimum time that a packet can wait in its route toward the sink. We also show an $\Omega(\log U)$ lower bound, whence the proposed algorithm is best possible up to a multiplicative constant. (b) *Distributed asynchronous.* We give an $O(\delta \log U)$-competitive algorithm, where δ is the depth of the tree, which belongs to a class of algorithms for which we can prove a lower bound of $\Omega(\delta^{1-\epsilon})$ for any $\epsilon > 0$ on the competitive ratio.

Omitted proofs and more related results can be found in a full version [2] of this abstract.

Related results. In spirit [4] come closest to our paper. In [4] the authors consider optimization of TCP acknowledgement (ACK) in a multicasting tree. The problem their work addresses is a data aggregation problem. However, energy consumption is not an issue in this problem and latency is considered as a cost instead of a constraint, resulting in an objective of minimizing the sum of the total number of transmissions and the total latency of the messages.

In [5] the authors studied the optimal aggregation policy in a single-hop scenario (i.e. the graph is a star). Namely an *aggregator* performs a request and starts waiting for answers from a set of sources. The time for each source to return its data to the aggregator is independent and identically distributed according to a known distribution F. The main differences with our paper are that they assume that F is known, and they focus on a single-hop scenario.

2 The Sensor Problem Formalized

We study sensor networks $D = (V, A)$, which are *intrees* rooted at a *sink node* $s \in V$. Nodes represent sensors and arcs represent the possibility of transmission between two sensors. Given an arc $a \in A$ we denote its head and tail nodes by head(a) and tail(a), respectively.

Over time, n messages, $N := \{1, \ldots, n\}$, arrive at nodes and have to be sent to the sink. Message j arrives at its *release node* v_j at its *release date* r_j and must arrive at the sink via the unique $v_j - s$-path at or before its *due date* d_j. Thus, each message is completely defined by the triple (v_j, r_j, d_j). Unless otherwise stated we assume that messages are indexed by increasing due date, i.e., $d_1 \leq d_2 \leq \cdots \leq d_n$. We refer to $L_j := d_j - r_j$ as the *latency* of message j.

A *packet* is a set of messages which are sent simultaneously along an arc. More precisely, each initial message is sent as one packet. Recursively, two packets j and ℓ can be aggregated at a node v. The resulting packet has due date $d = \min\{d_j, d_\ell\}$. This definition naturally extends to the case of more packets aggregated together.

Transition of a message along an arc takes time and energy (cost). In this paper we assume that the transit time $\tau : A \to \mathbb{R}_{\geqslant 0}$ and transit cost $c : A \to \mathbb{R}_{\geqslant 0}$ are independent of packet size. We often refer to the *transit cost* of a node as the transit cost of its unique outgoing arc. This models the situation in which all messages have more or less the same size and where *total aggregation* is possible, as discussed in the introduction. For $v \in V$, let τ_v and c_v be, respectively, the total transit time and total transit cost on the path from v to s. For message j and node u on the path from v_j to s, we define *transit interval* $I_j(u)$ as the time interval during which message j can transit at node u: $I_j(u) := [r_j + \tau_{v_j} - \tau_u, d_j - \tau_u]$. In particular, $I_j(s) = [r'_j, d_j]$, where $r'_j := r_j + \tau_{v_j}$ is the *earliest possible arrival time* of j at s. We abbreviate I_j for $I_j(s)$ and call it the *arrival interval* of message j. We also write $|I|$ for the length of interval I; note that $|I_j(u)| = |I_j|$ for all j and for all u on the path from v_j to u.

Finally, we define $\delta := \max_v \tau_v$ as the depth of the network in terms of the transit time.

The objective of the sensor problem is to send all messages to the sink in such a way as to minimize the maximum transit cost per node, while satisfying the latency restrictions. Given that transit costs are independent of the size of packets sent, but linear in the number of packets sent, it is clearly advantageous to aggregate messages into packets at tail nodes of arcs.

3 The Off-Line Problem

We start by proving some properties of optimal off-line solutions.

Lemma 1. *There exists a minimum cost solution such that:*

(i) whenever two messages are present together at the same node, they stay together until they reach the sink;
(ii) a message never waits at an intermediate node, i.e., a node different from its release node and the sink;
(iii) the time when a packet of messages arrives at the sink is the earliest due date of any message in that packet.

Proof. (i): Repeatedly apply the argument that whenever two messages are together at the same node but split up afterwards, keeping the one arriving later at the sink with the other message does not increase cost.

(ii): Use (i) and repeatedly apply the following argument. Whenever a packet of messages arrives at an intermediate node and waits there, changing the solution by shifting this waiting time to the tail node of the incoming arc does not increase cost.

(iii): Follows similarly as (ii) by interpreting the time between the arrival of a packet at the sink and earliest due date as waiting time. □

Theorem 1. *The off-line sensor problem is strongly NP-hard.*

The proof of Theorem 1 involves a non-trivial reduction from the Satisfiability Problem and is deferred to the full version of the paper.

We give an ILP-formulation of the problem, based on Lemma 1, and show that rounding the optimal solution of the LP-relaxation yields a 2-approximation algorithm. For every message-arc pair $\{i, a\}$, we introduce a binary decision variable x_{ia}, which is set to 1 if and only if arc a is used by some message j which arrives at s at time d_i. We use the notation $j_{\min}$ for the smallest index i such that $d_i \geq r'_j$ and a_j for the first arc on the (unique) $v_j - s$- path.

$$\begin{aligned} &\min z \\ &\text{s.t. } z \geq c(a) \textstyle\sum_{i=1}^{n} x_{ia} \quad \forall a \in A, \\ &\quad \textstyle\sum_{i=j_{\min}}^{j} x_{ia_j} \geqslant 1 \quad \forall\, 1 \leq j \leq n, \\ &\quad x_{ia} \geqslant x_{ia'} \quad \forall\, 1 \leq i \leq n \;\forall\, a, a' \in A \text{ with } \mathrm{head}(a') = \mathrm{tail}(a), \\ &\quad x_{ia} \in \{0,1\} \quad \forall\, 1 \leq i \leq n \;\forall\, a \in A. \end{aligned} \tag{1}$$

The first set of constraints ensures that z is at least the transit cost of any node. The second set of constraints forces each message to leave its release node in time to reach the sink before its due date. By the third set of constraints a message does not wait at intermediate nodes.

In the following lemma we develop a tool for rounding the corresponding LP-relaxation, which is obtained by replacing the integrality constraints with non-negativity constraints $x_{ia} \geqslant 0$.

Lemma 2. *Let* $\alpha_1, \ldots, \alpha_n \in \mathbb{R}_{\geqslant 0}$ *and* $\beta_1, \ldots, \beta_n \in \{0,1\}$ *with*

$$\textstyle\sum_{i=j}^{k} \alpha_i \geqslant 1 \quad \Longrightarrow \quad \sum_{i=j}^{k} \beta_i \geqslant 1 \quad \forall 1 \leq k \leq n \;\forall 1 \leq j \leq k. \tag{2}$$

By decreasing some of the β_i*'s from* 1 *to* 0*, one can enforce the inequality*

$$\textstyle\sum_{i=1}^{n} \beta_i \leqslant 2 \sum_{i=1}^{n} \alpha_i \tag{3}$$

while maintaining property (2)*. Moreover, this can be done in linear time.*

Proof. Consider the β_i's in order of increasing index. If $\beta_i = 1$, then round it down to 0, unless this yields a violation of (2). It is not difficult to see that this greedy algorithm can be implemented to run in linear time. It remains to be proven that inequality (3) holds for the resulting numbers $\beta_1, \ldots, \beta_n$.

For $h \in \{1,\dots,n\}$, let $\bar{h} := \min\{i > h \mid \beta_i = 1\}$; if $\beta_i = 0$ for all $i > h$ or $h = n$, then $\bar{h} := n+1$. Similarly, let $\underline{h} := \max\{i < h \mid \beta_i = 1\}$; if $\beta_i = 0$ for all $i < h$ or $h = 1$, then $\underline{h} := 0$. We prove the following generalization of (3):

$$\sum_{i=1}^{h} \beta_i \leqslant 2 \sum_{i=1}^{\bar{h}-1} \alpha_i \quad \forall\, 1 \leq h \leq n. \tag{4}$$

By contradiction, consider the smallest index h violating (4). Since h is chosen minimally, it must hold that $\beta_h = 1$; rounding β_h down to 0 would yield a violation of (2). In particular this would yield

$$\sum_{i=\underline{h}+1}^{\bar{h}-1} \alpha_i \geqslant 1 \tag{5}$$

while $\sum_{i=\underline{h}+1}^{\bar{h}-1} \beta_i = 0$. Notice that $\underline{h} \geqslant 1$, since, by choice of h,

$$\sum_{i=1}^{h} \beta_i > 2 \sum_{i=1}^{\bar{h}-1} \alpha_i \overset{(5)}{\geqslant} 2 \ .$$

Thus, $\beta_{\underline{h}} = \beta_h = 1$. We get a contradiction to the choice of h:

$$\sum_{i=1}^{h} \beta_i = \sum_{i=1}^{\underline{h}-1} \beta_i + 2 \overset{(4)}{\leqslant} 2 \sum_{i=1}^{\underline{h}-1} \alpha_i + 2 \overset{(5)}{\leqslant} 2 \sum_{i=1}^{\underline{h}-1} \alpha_i + 2 \sum_{i=\underline{h}+1}^{\bar{h}-1} \alpha_i \leqslant 2 \sum_{i=1}^{\bar{h}-1} \alpha_i \ .$$

The first inequality follows from (4) since $\overline{(\underline{h}-1)} = \underline{h}$. □

Theorem 2. *There is a polynomial time 2-approximation algorithm for the sensor problem on intree $D = (V, A)$.*

Proof. We round optimal (fractional) solution (x, z) of the LP relaxation of (1) to an integral solution $(\bar{x}, \bar{z})$. Consider the arcs in order of non-decreasing distance from s. For arc a with $\text{head}(a) = s$, set $\hat{x}_{ia} = 1 \ \forall\ i = 1,\dots,n$. Modify these values to $\bar{x}_{1a},\dots,\bar{x}_{na}$ by applying Lemma 2 to $x_{1a},\dots,x_{na}$ and $\hat{x}_{1a},\dots,\hat{x}_{na}$.

For an arc a' with larger distance to s, take the arc a with $\text{head}(a') = \text{tail}(a)$ and set $\hat{x}_{ia'} := \bar{x}_{ia}\ \forall i = 1,\dots,n$. We also modify these values into $\bar{x}_{1a'},\dots,\bar{x}_{na'}$ by applying Lemma 2 to the values $x_{1a'},\dots,x_{na'}$ and $\hat{x}_{1a'},\dots,\hat{x}_{na'}$. Premise (2) of Lemma 2 is satisfied for $x_{1a'},\dots,x_{na'}$ and $\hat{x}_{1a'},\dots,\hat{x}_{na'}$ since (2) holds for $x_{1a},\dots,x_{na}$ and $\bar{x}_{1a},\dots,\bar{x}_{na}$ and since $x_{ia'} \leq x_{ia}$.

By construction, the final solution $(\bar{x}, \bar{z})$ is feasible if we choose $\bar{z} = 2z$. □

4 The Distributed On-Line Problem

We consider a class of *distributed on-line* models, in which nodes communicate independently of each other, while messages are released over time. Each node

is equipped with an algorithm, which determines at what times the node sends its packets to the next node on the path to the sink. The input of each node's algorithm at any time t is restricted to the packets that have been released at or forwarded from that node in the period $[0, t]$.

We assume that all nodes are equipped with a clock to measure the latency of messages. We distinguish two distributed on-line models: In the *synchronous* model all nodes are equipped with a *common clock*, i.e. the times indicated at all clocks are identical. A common clock may facilitate synchronization of actions in various nodes. In the *asynchronous* model there is no such common clock; still, the duration of the time unit is assumed to be the same for all nodes.

We also assume in both models that each node v knows its total transit time τ_v to the sink. Moreover, for the asynchronous model we assume that all transit times $\tau(a)$ are equal, and without loss of generality we set $\tau(a) = 1\ \forall a \in A$.

4.1 The Synchronous Model

For the synchronous model we propose an algorithm based on the following simple result, the obvious proof of which we omit.

Lemma 3. *Given any interval* $[a, b]$, $a, b \in \mathbb{N}$, *let* $i^* = \max\{i \in \mathbb{N} \mid \exists k \in \mathbb{N} : k2^i \in [a, b]\}$. *Then* k^* *for which* $k^*2^{i^*} \in [a, b]$ *is odd and unique. Also,* $i^* \geq \lfloor \log_2(b - a) \rfloor$. *We use notation* $t(a, b) = k^*2^{i^*}$. □

> **Algorithm:CommonClock** (CC) Message j is sent from v_j at time $t(r'_j, d_j) - \tau_{v_j}$ to arrive at s at time $t(r'_j, d_j)$ unless some other message (packet) passes v_j in the interval $[r_j, t(r'_j, d_j) - \tau_{v_j}]$, in which case j is aggregated and the packet is forwarded directly.

First we derive a bound on the competitive ratio of CC for instances in which the arrival intervals I_j differ by at most a factor 2 in length.

Lemma 4. *If there exists an* $i \in \mathbb{N}$ *such that* $2^{i-1} < |I_j| \leq 2^i$ *for all messages* j, *then* CC *has a competitive ratio of at most* 3.

Proof. Assume that in an optimal solution packets arrive at s at times $t_1 < \cdots < t_\ell$. Let N_h^* be the packet arriving at t_h at s. Since $t_h \in I_j\ \forall j \in N_h^*$ and $|I_j| \leq 2^i\ \forall j$, we have $I_j \subset [t_h - 2^i, t_h + 2^i] =: I\ \forall j \in N_h^*$, and $|I| = 2 \cdot 2^i$. If $t_h = k2^i$ then in the CC-solution all messages in N_h^* may arrive at s at times t_h, $t_h - 2^i$ or $t_h + 2^i$. If $t_h \neq k2^i$ then I contains two different multiples of 2^i, say $k2^i$ and $(k+1)2^i$, such that $k2^i < t_h < (k+1)2^i$. In this case, since $|I_j| > 2^{i-1}\ \forall j$, we have $\forall j \in N_h^*$ that $I_j \cap \{k2^i, k2^i + 2^{i-1}, (k+1)2^i\} \neq \emptyset$. Lemma 3 implies that in a CC-solution every message $j \in N_h^*$ arrives at s at one of $\{k2^i, k2^i + 2^{i-1}, (k+1)2^i\}$. Hence, $\forall h = 1, \ldots, \ell$, all messages in N_h^* arrive at s at at most 3 distinct time instants in the CC-solution. CC does not delay messages at intermediate nodes. This implies that the arcs used by messages in N_h^* are traversed by these messages at most 3 times in the CC-solution, proving the lemma. □

Theorem 3. CC *is* $\Theta(\max\{\log U, 1\})$*-competitive with* $U = \frac{\max_j |I_j|}{\max\{1, \min_j |I_j|\}}$.

Proof. For each $i \in \mathbb{N}$ with $\log(\max\{1, \min_j |I_j|\}) \leq i \leq \lceil \log(\max_j |I_j|) \rceil$, CC sends the messages in $N_i := \{j \in N \mid 2^{i-1} < |I_j| \leq 2^i\}$, at a cost of no more than 3 times the optimum, by Lemma 4. This proves $O(\max\{\log U, 1\})$-competitiveness if $\min_j |I_j| \geq 1$. In case $\min_j |I_j| = 0$ we observe that restricted to the class of messages $N_0 = \{j \in N \mid |I_j| = 0\}$ CC's cost equals the optimal cost, because there is no choice for these messages.

To prove $\Omega(\log U)$ consider a chain of 2^{n+1} nodes $u_1, \ldots, u_{2^{n+1}} = s$ for some $n \in \mathbb{N}$. Take $\tau(a) = 1$ and $c(a) = 1$ $\forall a$. For $j = 1, \ldots, n$, $v_j = u_{2^j}$, $r_j = 0$, and $d_j = 2^{n+1} - 1$. Hence $r'_j = 2^{n+1} - 2^j = k2^j$ for some odd $k \in \mathbb{N}$ and $|I_j| = 2^j - 1$. Therefore, CC makes each message j arrive at s at time r'_j, no two messages are aggregated, and the cost is $\sum_{j=1}^{n}(2^{n+1} - 2^j) = (n-1)2^{n+1} + 2$. In an optimal solution all messages are aggregated into a single packet arriving at s at time $2^{n+1} - 1$ at a cost of $2^{n+1} - 2$. Notice that $U = 2^n - 1$ in this case. □

The following theorem shows that CC is best possible (up to a multiplicative constant).

Theorem 4. *Any deterministic synchronous algorithm is* $\Omega(\log U)$*-competitive.*

Proof. Consider an intree of depth $\delta = 2^{n+1}$ with n the number of messages, and where each node, except the leaves, has indegree n. We assume $\tau(a) = 1$ for all $a \in A$. For any on-line algorithm we will construct an adversarial sequence of n messages all with latency $L = \delta$, such that there exists a node at which the adversary can aggregate all messages in a single packet, but at which none of them is aggregated by the on-line algorithm. Using a similar argument as in the proof of Lemma 1 (i) the fact that all messages can be aggregated in a single packet implies that there exists a solution such that every node sends at most one packet, hence the cost of the adversarial solution is 1, whereas the cost of the on-line algorithm is n.

Fix any on-line algorithm. Given an instance of the problem, let $W_j(u)$ be the time interval message j spent at node u by application of the algorithm, i.e. the waiting time interval of message j on u. We denote its length by $|W_j(u)|$. Note that $\sum_u |W_j(u)| \leq |I_j|$ for each message j. We notice that the waiting time of a message in a node can be influenced by the other messages that are present at that node or have passed that node before. Since the algorithms are distributed the waiting time of a message in a node is not influenced by any message that will pass the node in the future.

The adversary chooses the source node v_j with total transit time $\tau_{v_j} := \delta - 2^j$ from s, for $j = 1, \ldots, n$, so that $|I_j| = 2^j$. Thus, $U = 2^{n-1} = \delta/4$. The choice of the exact position of v_j and the release time r_j is made sequentially and, to facilitate the exposition, described in a backward way starting with message n. The proof follows rather directly from the following claim.

Claim. For any set of messages $\{k, \ldots, n\}$ the adversary can maintain the properties:

(i) all messages in $\{k, \ldots, n\}$ pass a path p_k with 2^k nodes;
(ii) $I_k(u) = \bigcap_{j \geq k} I_j(u)\ \forall u \in p_k$;
(iii) if $k < n$, then $W_{k+1}(u) \bigcap I_k(u) = \emptyset\ \forall u \in p_k$;
(iv) if $k < n$, then $W_i(u) \bigcap W_j(u) = \emptyset\ \forall u \in p_k$, $i = k, \ldots, n$, $j > i$.

We notice that for any message j and any node u on the path from v_j to s, $W_j(u)$ may have length 0 but is never empty; it contains at least the departure time of message j from node u.

Note that properties (i) and (ii) for $k = 1$ imply that all messages can indeed be aggregated into one packet, hence as argued above, the adversarial solution has a cost of 1. Properties (iv) and (i) for $k = 1$ imply that the on-line algorithm sends all messages separately over a common path with 2 nodes, yielding a cost of n. This proves the theorem.

We prove the claim by induction. The basis of the induction, $k = n$, is trivially verified. Suppose the claim holds for message set $\{k, \ldots, n\}$ and p_k is the path between nodes $\overline{v}$ and $\hat{v}$. We partition p_k into two sub-paths $\overline{p}$ and $\hat{p}$ consisting of 2^{k-1} nodes each, such that $\overline{v} \in \overline{p}$ and $\hat{v} \in \hat{p}$. We denote the last node of $\overline{p}$ by $\overline{u}$ and the first node of $\hat{p}$ by $\hat{u}$. We distinguish two cases with respect to the waiting times the algorithm has selected for message k in the nodes on p_k.

Case a: $\sum_{u \in \overline{p}} |W_k(u)| \geq (1/2)|I_k|$. The adversary chooses v_{k-1} with total transit time $\tau_{v_{k-1}} = \delta - 2^{k-1}$ such that its path to s traverses $\hat{p}$ but not $\overline{p}$. More precisely, we ensure that the first node message $k-1$ has in common with any other message is $\hat{u}$. This is always possible, since the node degree is n. This choice immediately makes that setting $p_{k-1} = \hat{p}$ satisfies property (i). The release time of $k-1$ is chosen so that $I_{k-1}(\hat{u})$ and $I_k(\hat{u})$ start at the same time, implying that $I_{k-1}(u)$ and $I_k(u)$ start at the same time for every $u \in \hat{p}$. Since $|I_{k-1}(u)| = |I_k(u)|/2$ we have $I_{k-1}(u) \subset I_k(u)$ for all $u \in \hat{p}$, whence property (ii) follows by induction.

Note that, as we consider distributed algorithms, message $k-1$ does not influence the waiting time of $j, j > k-1$, on $\overline{p}$ as $\hat{u}$ is the first node which both j and $k-1$ traverse. In particular, $W_k(u), \forall u \in \overline{p}$ is not influenced by $k-1$.

Now, the equal starting times of $I_{k-1}(\hat{u})$ and $I_k(\hat{u})$ together with $\sum_{u \in \overline{p}} |W_k(u)| \geq (1/2)|I_k|$ and $|I_{k-1}(\hat{u})| = |I_k(\hat{u})|/2$ imply that k will not reach $\hat{u}$ before interval $I_{k-1}(\hat{u})$ ends. This, together with the consideration above, implies property (iii).

To prove (iv), note that by induction it is sufficient to prove that $W_{k-1}(u) \cap W_j(u) = \emptyset\, \forall j > k-1\ \forall u \in \hat{p}$. Since, as just proved, $W_k(u) \cap I_{k-1}(u) = \emptyset\ \forall u \in \hat{p}$ we have $W_{k-1}(u) \cap W_k(u) = \emptyset\ \forall u \in \hat{p}$. We have by induction that, for $j > k$, $W_j(u) \cap I_{j-1}(u) = \emptyset\ \forall u \in \hat{p}$ and we just proved that $I_{k-1}(u) \subset I_{j-1}(u) \subset I_j(u)\ \forall u \in \hat{p}$, which together imply $W_{k-1}(u) \cap W_j(u) = \emptyset\ \forall j > k\ \forall u \in \hat{p}$.

Case b: $\sum_{u \in \overline{p}} |W_k(u)| < (1/2)|I_k|$. As in the previous case, the adversary chooses v_{k-1} with total transit time $\tau_{v_{k-1}} = \delta - 2^{k-1}$ such that its path to s traverses $\overline{p}$ (therefore also $\hat{p}$) but does not intersect any of the paths used by

messages $\{k, \ldots, n\}$ before it reaches $\overline{p}$ in $\overline{v}$. Again, this is always possible since the indegree of each node is n. Hence, choosing $p_{k-1} = \overline{p}$ satisfies property (i). The release time of $k-1$ is chosen so that $I_{k-1}(\overline{v})$ and $I_k(\overline{v})$ end at the same time, implying that $I_{k-1}(u)$ and $I_k(u)$ end at the same time for every $u \in \overline{p}$. Since $|I_{k-1}(u)| = |I_k(u)|/2$ we have $I_{k-1}(u) \subset I_k(u)$ for all $u \in \overline{p}$, whence property (ii) follows by induction.

The equal ending times of $I_{k-1}(\overline{u})$ and $I_k(\overline{u})$ together with $\sum_{u \in \overline{p}} |W_k(u)| < 1/2|I_k|$ and $|I_{k-1}(\overline{u})| = |I_k(\overline{u})|/2$ imply that k has left $\overline{u}$ before $I_{k-1}(\overline{u})$ begins, implying property (iii). Indeed, this gives $W_{k-1}(u) \cap W_k(u) = \emptyset, \forall u \in \overline{p}$. It also implies that $k-1$ could not influence the waiting time of k on $\overline{p}$.

The proof of (iv) follows the very same lines as in Case a, with the difference that we now refer to nodes in $\overline{p}$ instead of $\hat{p}$. □

Since in the proof $U = \delta/4$ we also have the following lower bound on the competitive ratio of any deterministic synchronous algorithm.

Theorem 5. *Any deterministic synchronous algorithm is $\Omega(\log \delta)$-competitive.* □

4.2 The Asynchronous Model

In the asynchronous model nodes are equipped with a clock and a distributed algorithm. All clocks have the same time unit, but neither the time nor the start of a new time unit on clocks is synchronized. We assume that $\tau(a) = 1$ for all a, such that τ_{v_j} is equal to the number of nodes on the $v_j - s$-path.

We propose algorithm Spread Latency (SL) for this model, which divides the latency minus transmission time of each message j equally over the nodes on the $v_j - s$-path: at each node of this path the message is assigned a waiting time of $(L_j - \tau_{v_j})/\tau_{v_j}$ time units. As soon as messages appear simultaneously at the same node they get aggregated into a packet, which is sent over the outgoing arc as soon as the waiting time of at least one of its messages at that node has passed. In this way, no message is delayed due to aggregation and thus the algorithm yields a feasible solution.

Let, as in the previous subsection, $U := \frac{\max_j |I_j|}{\max\{1, \min_j |I_j|\}} = \frac{\max_j (L_j - \tau_{v_j})}{\max\{1, \min_j (L_j - \tau_{v_j})\}}$.

Theorem 6. *The algorithm* SL *is $O(\delta \max\{\log U, 1\})$-competitive.*

Proof. We prove that for all $a \in A$ the number of packets SL sends through a is at most $O(\delta \max\{\log U, 1\})$ times that number in an optimal solution. This proves the theorem.

Let $\lambda := \max\{1, \min_j (L_j - \tau_{v_j})\}$. Consider a packet P of messages sent by an optimal solution through (u, v) at t. To bound the number of packets sent by SL that contain at least one message from P, define $P_k := \{j \in P \mid 2^{k-1}\lambda \leq L_j - \tau_{v_j} < 2^k \lambda\}$, for $k = 1, \ldots, \lceil \log U \rceil$. We charge any sent packet to the message that caused the packet to be sent due to its waiting time being over. It suffices to prove that the number of packets charged to messages in P_k is $O(\delta)$.

Since the waiting time of messages $j \in P_k$ at node u is at least $2^{k-1}\lambda/\delta$, the delay between any two packets that are charged to messages in P_k is at least $2^{k-1}\lambda/\delta$. Since the optimal solution sends packet P at t through (u, v), we get

$t \in I_j(u)\ \forall j \in P$ and thus $I_j(u) \subseteq [t - 2^k\lambda, t + 2^k\lambda]\ \forall j \in P_k$. Thus, the number of packets charged to messages in P_k is at most $2 \cdot 2^k\lambda/(2^{k-1}\lambda/\delta) = 4\delta$. □

SL determines the waiting time of each message at the nodes it traverses independently of all other messages. We call such an algorithm a WI-algorithm. To be precise, in a WI-algorithm node v determines the waiting time of message j based only on the message characteristics (v_j, r_j, d_j), transit time to the sink τ_v and clock time. The following lower bound shows that the competitive ratio of SL cannot be beaten by more than a factor $\max\{\log U, 1\}$ by any other WI-algorithm. In the derivation of the lower bound we restrict to WI-algorithms that employ the same algorithm in all nodes with the same transit time to s. This is not a severe restriction, given that transit time to s is the only information about the network that a node has.

Theorem 7. *Any deterministic asynchronous WI-algorithm is $\Omega(\delta^{1-\epsilon})$- competitive for any $\epsilon > 0$.*

Proof. Consider a binary intree with root s and all leaves at distance δ from s. Let $0 \leq \lambda < 1$ be such that $\delta^{1-\lambda} \geq 3$. An adversary releases message 1 with latency L at time r_1 in a leaf v_1. Notice that there are at most δ^λ nodes where the waiting time is at least $(L - \tau_{v_1})/\delta^\lambda$. Hence, the $v_1 - s$ path contains a sub-path consisting of at least $\delta^{1-\lambda} - 2$ nodes where in each node message 1 waits less than $(L - \tau_{v_1})/\delta^\lambda$. Choose such a sub-path and let u be the node on this sub-path closest to s.

Let V' be the set of leaves of the subtree with root u and depth $\delta^{1-\lambda} - 2$. Then $|V'| \geq 2^{\delta^{1-\lambda}-2} \geq \delta^\lambda/4$ for any fixed $\lambda \in [0, 1)$ and δ large enough. The adversary releases messages $j = 2, \ldots, \delta^\lambda/4$ with latency L at times $r_j = r_1 + j(L - \tau_{v_j})/\delta^\lambda$ in leaf v_j, such that each $v_j - s$ path passes through a different vertex of V'. Because $\tau_{v_j} = \tau_{v_1}\ \forall j$ and we assumed that any WI-algorithm applies the same algorithm in nodes at equal distance, all messages are sent non-aggregated to and from u, whereas they are aggregated as early as possible in an optimal solution, in particular at u. □

The lower bound does not hold for arbitrary algorithms as a node may adjust the waiting time of subsequent messages that traverse that node. The following theorem shows that the lower bound remains $\Omega(\delta^{1-\epsilon})$ if release nodes do not delay subsequent messages longer than preceding messages.

Theorem 8. *Any asynchronous WI-algorithm for which the waiting time of message j at its release node is at most $\frac{L-\tau_{v_j}}{K}$ is $\Omega(K)$-competitive.*

Proof. Consider a chain which consists of two nodes v and s. We assume constant latency L for each message. The adversary releases $K - 1$ messages with an interval of $(L - \tau_{v_j})/(K - 1)$ at v. Since the waiting time of message j at v is at most $(L - \tau_{v_j})/K$, none of these messages are aggregated in the on-line solution, whereas they are all aggregated in one packet in an optimal solution. □

The theorem proves that SL is $\Omega(\delta)$-competitive. For arbitrary asynchronous algorithms we do not have any better lower bound than the one in Theorem 5.

In a full version of the paper [2] we design an algorithm with improved competitive ratio of $O(\log^3 \delta)$ for the asynchronous problem on a chain with s at one of its ends.

References

1. I. Akyildiz, W. Su, Y. Sanakarasubramaniam, and E. Cayirci. Wireless sensor networks: A survey. *Computer Networks Journal*, 38(4):393–422, 2002.
2. L. Becchetti, P. Korteweg, A. Marchetti-Spaccamela, M. Skutella, L. Stougie, and A. Vitaletti. *Latency Constrained Aggregation in Sensor Networks.* SPOR-report 2006-08, TU Eindhoven, www.win.tue.nl/bs/spor, 2006.
3. A. Boulis, S. Ganeriwal, and M. B. Srivastava. Aggregation in sensor networks: an energy - accuracy tradeoff. *Ad-hoc Networks Journal*, 1(2-3):317–331, 2003.
4. C. Brito, E. Koutsoupias, and S. Vaya. Competitive analysis of organization networks or multicast acknowledgement: how much to wait? In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms (SODA)*, pages 627–635, 2004.
5. A. Z. Broder and M. Mitzenmacher. Optimal plans for aggregation. In *PODC*, pages 144–152, 2002.
6. A. Goel and D. Estrin. Simultaneous optimization for concave costs: single sink aggregation or single source buy-at-bulk. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms (SODA)*, pages 499–505, 2003.
7. W. Heinzelman, A. Chandrakasan, and H. Balakrishnan. Energy efficient communication protocols for wireless microsensor networks. In *Proceedings of Hawaiian International Conference on Systems Science*, pages 3005–3014, 2000.
8. F. Hu, X. Cao, and C. May. Optimized scheduling for data aggregation in wireless sensor networks. In *International Conference on Information Technology Coding and Computing (ITCC)*, pages 557–561, 2005.
9. C. Intanagonwiwat, D. Estrin, R. Govindan, and J. Heidemann. Impact of network density on data aggregation in wireless sensor networks. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS)*, pages 414–458, 2002.
10. C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Mobile Computing and Networking*, pages 56–67, 2000.
11. K. Kalpakis, K. Dasgupta, and P. Namjoshi. Efficient algorithms for maximum lifetime data gathering and aggregation in wireless sensor networks. *Computer Networks*, 42(6):697–716, 2003.
12. S. Lindsey and C. S. Raghavendra. Pegasis: Power-efficient gathering in sensor information systems. In *Proceedings of IEEE Aerospace Conference*, pages 1125–1130, 2000.
13. S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tag: A tiny aggregation service for ad-hoc sensor networks. In *Proceedings of the 5th ACM Symposium on Operating System Design and Implementation (OSDI)*, pages 131–146, 2002.
14. S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems (TODS)*, 30(1):122–173, 2005.
15. W. Yuan, V. S. Krishnamurthy, and S. K. Tripathi. Synchronization of multiple levels of data fusion in wireless sensor networks. In *Proceedings of IEEE Globecom*, pages 221–225, 2003.

Competitive Analysis of Flash-Memory Algorithms

Avraham Ben-Aroya and Sivan Toledo

School of Computer Science, Tel-Aviv University
{abrhambe, stoledo}@tau.ac.il

Abstract. The cells of flash memories can only endure a limited number of write cycles, usually between 10,000 and 1,000,000. Furthermore, cells containing data must be erased before they can store new data, and erasure operations erase large blocks of memory, not individual cells. To maximize the endurance of the device (the amount of useful data that can be written to it before one of its cells wears out), flash-based systems move data around in an attempt to reduce the total number of erasures and to level the wear of the different erase blocks. This data movement introduces interesting online problems called *wear-leveling problems*. We show that a simple randomized algorithm for one problem is essentially optimal. For a more difficult problem, we show that clever offline algorithms can improve upon naive approaches, but online algorithms essentially cannot.

1 Introduction

The read/write/erase behaviors of flash memory is radically different than that of other programmable memories, such as magnetic disks and volatile RAM. Most importantly, flash memory cells can be erased only a limited number of times, between 10,000 and 1,000,000, after which they wear out and become unusable.

Writing to flash involves two separate operations: erasures and programming. An erasure sets all the bits in a range of cells to '1'. These ranges are called *erase units* and are usually uniform in size. We denote the number of erase units by n. The programming operation writes a given bit sequence to an erase unit, or to a part of an erase unit by clearing some of the 1's. We assume in this paper that the computer system always programs fixed-length sequences called *blocks*. We denote the number of blocks that fit within an erase unit by k. That is, each erase unit is divided into k *slots* that can each store a single block. A slot that has been programmed cannot be programmed again until the entire erase unit is erased (there are exceptions to this rule, but they are beyond the scope of this paper). Both $k = 1$ and higher values of k occur in practice.

Clever management of a flash device can dramatically extend its functional life span. Consider a device with n erase units that can each be erased H times, which are not divided into slots (that is, $k = 1$). We consider the device useless when one of the n cells exceeds the wear limit H (our results justify this assumption). When the device becomes useless, it has been written to between $H + 1$ and

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 100–111, 2006.

$n(H+1)$ times. Clever management aims to ensure that the device can be successfully written to as close to $n(H+1)$ times as possible. Techniques that aim to achieve this goal are called in the flash literature *wear-leveling* techniques.

Wear-leveling techniques work by separating the system's naming of blocks from the physical location of the slots that contain them. The computer system views the flash device as a store of $m \leq nk$ fixed-size blocks named 1 through m. The system uses the flash by issuing a sequence of read requests and write requests. Read requests are irrelevant to endurance so we ignore them. Write requests require the flash memory manager to store the new content of a named block and to return it in the future. Initially, each data block is stored in some slot. These slots are *occupied*. The flash memory manager serves a write request by performing a sequence of erasure and programming operations. In this sequence, blocks can only be written to *clean* slots (slots that have not been written to since the containing erase unit was erased), not to *dirty* slots (slots that contain obsolete data). When a unit is erased, all its contents are lost, and all its slots become clean. The sequence always needs to achieve one goal, and in most systems, it needs to achieve two more:

- At the end of the sequence each block must be stored in some slot. This is always necessary.
- The rearrangement of blocks might also contribute to wear leveling.
- Most systems require that the rearrangement of blocks is carried out such that no data is lost if the system is shut off in the middle of the sequence. This is an atomicity requirement with respect to the block-update request.

If atomicity is not an issue, the sequence always has the same structure. The manager begins the sequence by marking the slot that contains the old copy of the block as obsolete. If the manager wishes to rearrange additional blocks, it reads them into volatile memory (RAM) and marks the slots that contained them as obsolete. Next, the manager can erase units that contain no occupied slots, only clean and dirty ones. Finally, the manager writes all the blocks that are in volatile memory, including the updated block that initiated the sequence, into erased slots. If atomicity is required, or if the amount of RAM is limited, update sequences are more complex. The mapping issue (remembering where each block is stored) is largely orthogonal to the endurance issue, and we ignore it in this paper.

Starting around 1993, a variety of wear-leveling techniques have been proposed, mostly in patents [1, 4, 5, 6, 7, 9, 10, 11, 12, 13]; for details about these techniques and about other flash-management techniques, see [8]. In this paper, we present a competitive analysis of online wear-leveling policies, including of patented randomized policies. No such analysis has ever been published. Some of our analyses, such as the lower bounds for deterministic policies, apply directly to algorithms that have been previously proposed.

When the request series begins, we have m occupied slots and $nk-m$ clean slots. Thus, the manager cannot serve more than $(nk-m)+Hnk$ requests. We can, therefore, assume that the length of all request sequences is $(nk-m)+Hnk$.

The objective of the manager is to serve as many requests as possible before the device wears out.

We use competitive analysis to quantify the effectiveness of online wear-leveling policies. Let $\ell_{\mathsf{opt}}(\sigma)$ be the number of requests that the optimal offline algorithm can serve for a given request sequence σ, and let $\ell_\alpha(\sigma)$ be the number of requests that an online algorithm α can serve. The competitive ratio of α is $\min_\sigma \ell_\alpha(\sigma)/\ell_{\mathsf{opt}}(\sigma)$, where the minimization is over all the sequences of length $(nk-m)+Hnk$. A good online policy achieves a high competitive ratio. If α is randomized then we replace $\ell_\alpha(\sigma)$ by the expected length that α can serve.

The paper is organized as follows. Section 2 analyzes the case $k = 1$ and Section 3 analyzes the case $k > 1$. It turns out that these two cases are quite different and are governed by different issues. Each of these sections describes an effective offline algorithm, bounds on the competitiveness of deterministic and randomized online algorithms, and online algorithms that match most of the bounds. Section 4 presents our conclusions. Due to lack of space, most of the proofs have been omitted; see [3] for the proofs, for additional simulation results, and for a fuller discussion of implications of the results.

2 Single-Slot Units

We begin the analysis with single-slot erase units ($k = 1$). This case models at least two real-world situations: flash devices that limit programming operations to entire erase units, and flash devices that allow variable-size programming operations, which usually leads system designers to use single-slot wear-leveling algorithms.

In this case we can simplify the rules. First, we refer only to (erase) units, not to slots. Second, units can be in only two states, clean and occupied (and not dirty): We immediately erase a unit when a block is moved from it. Any result for this simplified model applies to the full model up to a change of ± 1 to H.

2.1 Deriving Atomic Policies from Non-atomic Ones

We present a method that allows us to separate the atomicity concern from the wear-leveling concern. This method transforms many non-atomic algorithms, including all the algorithms in this section, into atomic ones with exactly the same endurance.

We denote by $h_i(t)$ the number of erasures that the algorithm already performed on unit i immediately before serving request number t. We call $h_i(t)$ the *wear* of i at time t. We denote by σ_t the index of the block requested by the tth request in the sequence σ.

Theorem 1. *Let $\mathcal{N}$ be a non-atomic algorithm for n blocks and n units that serves a request sequence η by either putting η_t back in its unit or by switching η_t with some other block. We can derive from $\mathcal{N}$ an atomic algorithm $\mathcal{A}$ for $n-1$ blocks and n units. For any request sequence σ that $\mathcal{A}$ serves, there is another*

sequence η *such that* $h_i^{\mathcal{A}(\sigma)}(t) = h_i^{\mathcal{N}(\eta)}(t)$ *for all* i *and for all* t. *If* $\mathcal{N}$ *is online then* $\mathcal{A}$ *is online.*

The transformation simulates a fixed trivial online algorithm $\mathcal{T}$ on σ and uses its state (the block-to-unit mapping) to define η. Then $\mathcal{N}$ is simulated on η. The action of $\mathcal{N}$ on η_t, which is always one of two possible actions, is used to deterministically define the action of $\mathcal{A}$ on σ_t. Therefore, if $\mathcal{A}$ is randomized, then $\mathcal{N}$ essentially uses the coin tosses of $\mathcal{A}$, and the result $h_i^{\mathcal{A}(\sigma)}(t) = h_i^{\mathcal{N}(\eta)}(t)$ holds for every sequence of coin tosses. Thus, the theorem implies that the endurance of $\mathcal{A}$ and $\mathcal{N}$ is the same, both in the worst-case sense and in probabilistic senses.

2.2 Offline Algorithms

The best-case endurance (for "easy" sequences) is $\ell(\sigma) = nH$. Offline algorithms can achieve almost this best-case endurance.

Theorem 2. *There is an atomic offline algorithm for which the wear* $h_i(t)$ *of any unit* i *at time* $t = Hn - n + 1$ *is at most* H, *for any sequence* σ, *even if* $m = n - 1$. *For* $m = n$ *there is a non-atomic algorithm that achieves this endurance.*

This implies that an offline algorithm can always achieve $\ell_{\mathsf{off}}(\sigma) \geq n(H-1)$. Since $H > 10,000$, the offline endurance is exceedingly close to the best-case endurance nH.

The non-atomic offline algorithm works as follows (the atomic one uses Theorem 1). Normally, the algorithm serves a request to block x stored in unit i by erasing i and putting x back into i. In some cases, however, the algorithm decides to exchange the contents of i with the contents of another unit j. To decide whether to switch i with j, the algorithm counts the number of remaining requests to all the blocks, but only up to request number $Hn-n$. It switches if the number of remaining requests to some block y stored in unit j matches exactly the number of erasures left for unit i, *or* if the number of remaining requests to x matches exactly the number of erasures left for unit j. The algorithm performs at most n such switches on a given sequence. Notice that a switch performs two erasures, while a write in-place performs only one.

2.3 Deterministic Online Algorithms

The endurance of deterministic algorithms depends entirely on the number $n-m$ of extra erase units. An online algorithm can achieve this endurance by always putting the requested block in the least-worn out empty unit.

Theorem 3. *Under this deterministic online algorithm, the wear of any unit after* $(n - m + 1)H$ *requests is at most* H.

This is as good as any deterministic algorithm can achieve.

Theorem 4. *For every deterministic algorithm α (even if α is non-atomic) there exists a sequence σ such that $\ell_\alpha(\sigma) \leq (n - m + 1)H$.*

In spite of this pessimistic competitive result, many flash-based systems use deterministic algorithms. This is due to the fact that in practice the request sequence is oblivious to the mapping and such algorithms usually work well.

2.4 Randomized Online Algorithms

We analyze a randomized algorithm called $\mathcal{R}_p$, which was patented by Amir Ban along with several other wear-leveling algorithms [2]. The algorithm that we analyze corresponds to Claims 2.c, 3, and 4.II in [2]. It serves a request to block x using the following rules:

- With probability p, put x in a random unit i chosen uniformly and independently, and put the block that was in i in the unit where x was stored (the algorithm may return x to the unit in which it was stored).
- Otherwise, put x back in the unit from which it was taken out.

This algorithm is not atomic, but it satisfies the conditions of Theorem 1, so we can easily derive an atomic variant with the same competitive ratio. In the analysis of the algorithm, we assume that the blocks are initially stored in random units.

For $p = 1$, the behavior of this algorithm is fairly simple. Since the block that was requested is always switched with a random block, uniformly, an adversary has no useful information about which block is in which unit. Therefore, any request sequence is equivalent to a random request sequence. For large H, the wear of all the units under a random request sequence is roughly the same. Since serving each request usually costs the algorithm two erasures (if m is close to n), the endurance should be close to $nH/2$. A full analysis of $\mathcal{R}_1$ is not difficult.

However, a small p can improve the endurance and bring it close to nH. Figure 1 shows the results of simulations of $\mathcal{R}_p$. For given n and H, we simulated $\mathcal{R}_p$ with several values of p, 50 times for each p, with a constant request sequence. Each cross on the graph indicates the length of the sequence that a particular run was able to serve. The results clearly show that the simple variant $\mathcal{R}_1$ achieves endurance of roughly $nH/2$, but for a small p, the algorithm $\mathcal{R}_p$ can get close to nH. The main goal of our analysis, which is more complex than the analysis of $\mathcal{R}_1$, is to fully analyze this phenomenon and to provide guidelines for the choice of p.

A small p only helps, however, if H is large. If H is small, $\mathcal{R}_1$ is optimal. We begin with results that bound the performance of randomized algorithms for small H and that show that $\mathcal{R}_1$ is optimal in this regime. The results from here on are asymptotic with respect to a growing n.

Theorem 5. *For any randomized online algorithm α (even if α is non-atomic) and for every constant e such that $n = m + e$, there exists a sequence σ and a constant c such that $\Pr\left[\ell_\alpha(\sigma) < n^{1-c/H} \ln n\right] \geq 1 - o(1)$. (The constants within the small-o, as well as c, depend on e).*

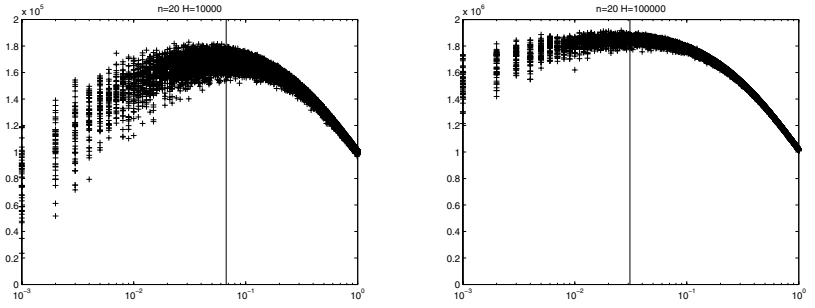


Fig. 1. Simulation results for $n = 20$ erase units and endurance limits of $H = 10,000$ (left) and $H = 100,000$ (right). Each cross represents one simulation. The x axis shows the switching probability p that was used in the simulation, the y axis shows the number of requests that were served before one of the units was erased $H + 1$ times. The y axis always extends up to exactly nH erasures, the ideal endurance. Theorem 7 shows that for switching probabilities far from 1 but significantly larger than $p = (\ln n/H)^{1/3}$ (indicated in the graphs by the vertical line), the endurance is nearly ideal.

Theorem 6. *For H in the range*

$$\Omega(\ln n/\ln\ln n) \le H \le O(\ln n)\ ,$$

there exists a constant $d > 0$ such that for every request sequence σ,

$$\Pr\left[\ell_{\mathcal{R}_1}(\sigma) < n^{1-d/H}\ln n\right] = o(1)\ .$$

When H is large, a good choice for p brings $\ell_{\mathcal{R}_p}(\sigma)$ almost to nH.

Theorem 7. *When $H = \omega(\ln n)$, for any $(\ln n/H)^{1/3} \ll p \ll 1$ and for every request sequence σ,*

$$\Pr\left[\ell_{\mathcal{R}_p}(\sigma) < nH(1 - o(1))\right] = o(1)\ .$$

3 Fractional Unit Wear Leveling

Until now, we have analyzed the single-slot case $k = 1$. Some flash based systems support fixed-size fractional writes, in which erase units are k times larger than write blocks.

In this section, we consider the *fractional wear-leveling problem*, in which $k > 1$. In the fractional case, the best-case scenario with a single spare unit ($m = (n-1)k$) is $\ell = nHk$: all the blocks in a given unit are requested contiguously and are moved to the spare unit, then the unit with the k dirty slots is erased, and so on. If the blocks are requested such that the units are emptied cyclically, we achieve $\ell = nHk$. On the other hand, an algorithm that always moves all the blocks in a unit together (and erases an entire unit as soon as one of its slots

becomes dirty) can achieve at most $\ell = nH$. Algorithms that behave like this are essentially single-slot algorithms and the bounds that we presented earlier apply to them. True fractional algorithms try to achieve endurance close to $\ell = nHk$. Clearly, to achieve such endurance, algorithms must avoid greedy movement of blocks and operate with units that contain some dirty slots.

Achieving high endurance in the fractional case is more difficult than achieving high endurance in the single-slot case; the two problems are very different. Consider, for example, a request sequence with random requests. In the single-slot case, even a naive non-atomic write-in-place deterministic algorithm achieves high endurance on such a sequence. The whole point of our randomized online algorithm was to introduce similar randomness into the process of serving an arbitrary sequence. But in the fractional problem, a random request sequence is difficult to serve, because it causes slots in many units to become dirty. When there are no more empty slots, the algorithm must erase some unit. But with high probability, no unit contains close to k dirty slots. Therefore, the algorithm will need to erase a unit with a relatively small number of dirty slots, leading to low endurance.

3.1 An Offline Algorithm for the Fractional Problem

Clearly, $\ell_{\mathsf{opt}} \leq (nk - m) + nHk$. On the other hand, a simple lower bound is given by the same algorithm that we used in the single-slot unit problem. This can be done by treating all blocks that are initially in the slots of the same unit as one big block that moves between units. As seen previously, this algorithm will achieve $\ell_{\mathsf{alg}} \geq (H-1)n$.

Obviously, the fractional unit wear leveling problem is only interesting when there are some spare empty units, since when all the units are full, the problem is equivalent to the single-slot unit wear leveling problem. First, we describe an non-atomic algorithm, which for $H \gg 1$ achieves

$$\ell_{\mathsf{off}}(\sigma) \geq (1 - o(1)) \left(\frac{kHn}{9}\right)^{2/3} = (1 - o(1)) \frac{\left(\frac{1}{9}k\right)^{2/3}}{(Hn)^{1/3}} Hn$$

using a single empty unit. This algorithm is better than the naive algorithm when k is sufficiently large (specifically, when $k \geq 9\sqrt{Hn}$). We later describe two atomic variants, one with the same ℓ using $k+2$ empty units, and another which loses a factor of $\log k$ but uses only three empty units.

The idea is to split the concerns of the algorithm. We first devise an algorithm $\mathcal{N}$ that attempts to minimize the total wear (the total number of erasures). We then apply the wear-leveling policy from Section 2.2 to even the wear among the units.

The algorithm $\mathcal{N}$ serves σ_t as follows. If there is a clean slot, it puts σ_t in it. Otherwise, it erases all the units that contain dirty slots, and sorts all the blocks stored in them in the order of their future arrival time. That is, after these erasures, there is a unit which is completely clean, and all the other erased units are completely occupied and sorted.

For simplicity we assume that initially unit n is the empty unit, and that whenever erasures are preformed, unit n is always erased and arranged such that after the arrangement it is the one empty unit (regardless of whether there are any dirty slots in it). This assures us that unit n is always the empty unit.

Since there are exactly k clean slots, $\mathcal{N}$ preforms erasures after each k consecutive requests. Thus, we split the executions of $\mathcal{N}$ into phases. Each phase consists of serving k requests and performing subsequent erasures. Phase ϕ ends just before serving $\sigma_{\phi k+1}$. Let A_ϕ denote the set of erased units at the end of the ϕth phase. Our main objective now is to bound $\sum A_\phi$. To do this, we define the following labeling scheme. A block x is associated with a set of labels denoted by S_x. Initially all these sets are empty. They are updated between phases. After the ϕth phase, the only blocks whose sets are updated are the blocks in the units of A_ϕ. First, the sets $S_{\sigma_{(\phi-1)k+1}}, \ldots, S_{\sigma_{\phi k}}$ of the requested blocks becomes empty. Then, for each of block x within the units of A_ϕ the label (ϕ, z_x) is added to S_x, where $z_x \in \{1, \ldots, k\}$ indicates the unit order of the unit that now contains x (i.e., if x's arrival time is the jth shortest one among the blocks in the units of A_ϕ, then $z_x = \lceil j/k \rceil$). Observe that a unit that contains blocks with label (ϕ, z) is not erased until all the blocks with label $(\phi, z-1)$ are requested. The sets $\{S_x\}$ change during the execution; we denote the set S_x before the ϕth phase by $S_x(\phi)$.

We say that a block x is *accessible* just before the ϕth phase begins if, for each label pair $(a, z) \in S_x(\phi)$, there is no other set $S_y(\phi)$ such that $(a, z') \in S_y(\phi)$ for some $z' < z - 1$. A unit $i \neq n$ is *erasable* just before the ϕth phase if all the blocks in it at that time are accessible. We denote by B_ϕ the set of erasable units before ϕth phase and define $\zeta_\phi = |B_{\phi+1} \setminus B_\phi|$.

Lemma 1. $A_\phi \setminus \{n\} \subseteq B_\phi$.

Lemma 2. $|B_{\phi+1}| \leq |B_\phi| - |A_\phi| + 3 + \zeta_\phi$ *for any* $\phi \geq 1$.

Proof. A unit that is erasable in the ϕth phase but was not erased (it is not in A_ϕ) is surely erasable in the $(\phi+1)$th phase. We now examine the set A_ϕ. By the previous lemma, every unit $i \neq n$ in A_ϕ was erasable before just before phase ϕ. At the end of the ϕth phase, the algorithm sorts the blocks in A_ϕ according to their next arrival times. Therefore, most of the units in this set are not erasable in the $(\phi + 1)$th phase: only the two units with the shortest arrival times are. This adds 2 to the right-hand side of the inequality. Unit n is always erased but it is not erasable (by definition): this adds 1 to the right-hand side. To bound $|B_{\phi+1}|$ we only need to add ζ_ϕ, the number of units that became erasable during the ϕth phase. □

Lemma 3. *Let $D_1, \ldots, D_\Phi$ be Φ sets of pairs of integers (a, z), where the first component in each pair is an integer between 1 and Φ. Suppose that for $i \neq j$, there is at most one integer a such that $(a, z_1) \in D_i$ and $(a, z_2) \in D_j$ for some $z_1 \neq z_2$ (i.e. a appears as the first component in a pair in D_i and in a pair in D_j). Then $\left|\bigcup_{i=1}^{\Phi} D_i\right| \leq 3\Phi\sqrt{\Phi}$.*

We can now prove a bound on $\sum_\phi \zeta_\phi$.

Lemma 4. *For any $\Phi \geq 1$ we have $\sum_{i=1}^{\Phi} \zeta_i \leq 9\Phi\sqrt{\Phi}$.*

Proof. We wish to bound the number of events in which a unit changes its state from non-erasable to erasable. A unit becomes non-erasable when it is erased and used to store blocks with label (a, z) for some $z > 2$. (The units that are used to store blocks with label (a, z) for $z = 1, 2$ are immediately erasable.) For such a unit to become erasable again, all the k blocks with labels $(a, z-2)$ must be requested. Until the label $(a, z-2)$ disappears, the unit that stores blocks labeled (a, z) does not become erasable again. Therefore, what we need to count to bound $\sum \zeta_i$ is the number of labels that completely disappears.

Only $k\Phi$ blocks are requested during the first Φ phases. However, this does not give a bound of Φ on the number of units that become erasable again during these Φ phases, because requested blocks with more than one label may contribute to the erasability of multiple units.

Let $C_t = S_{\sigma_t}(\lceil t/k \rceil)$ be the set of labels that block σ_t carries at time t. The number of labels (a, z) that appear exactly k times in the C_t's is exactly the number of units that become erasable again. Other labels, the ones that appear fewer than k times, are irrelevant and we completely ignore them in the rest of the analysis. Thus, from now on, we assume that each label appears in exactly k of the C_t's.

Let $D = \{D_1, \ldots, D_\Phi\}$ be a random sample of the C_t's, drawn uniformly and independently (with repetitions). The probability that a particular label appears in one of the D_i's is exactly $1/\Phi$, because exactly k of the $k\Phi$ sets C_t contain that label. The probability that a particular label does not appear in any of the D_i's is, therefore, $\left(1 - \frac{1}{\Phi}\right)^{\Phi} \leq \frac{1}{e} < \frac{2}{3}$. Hence, the probability that the label does appears in some of the D_i's is bounded from below by a constant. Therefore, the expected number of labels that appear in $\cup_i D_i$ is bounded from below by a $1/3$ times the number of labels in $\cup_t C_t$. This implies that there is some specific sample $D = \{D_1, \ldots, D_\Phi\}$ in which the number of labels is at least $1/3$ fraction of the labels in the (reduced) C_t's.

We say that two sets C_{t_1} and C_{t_2} are *linked by* a if $(a, z) \in C_{t_1}$ and $(a, z') \in C_{t_2}$ for some $z' \neq z$. We claim that if C_{t_1} and C_{t_2} are linked by a, then they cannot be linked by any other phase-label $b \neq a$. Suppose for contradiction that the claim is false and that the two sets are also linked by b. Without loss of generality, let $a < b$ and $z' > z$. If time t_1 occurs after phase b ends, then $(b, ?) \notin C_{t_2}$, because until after time t_1, the block associated with C_{t_2} is in a unit in which all the blocks are labeled by (a, z'). None of these blocks can be requested until after time t_1, so the block associated with C_{t_2} cannot be labeled with b. On the other hand, if time t_1 occurs before phase b ends, then $(b, ?) \notin C_{t_1}$.

Therefore, the C_t's satisfy the mutual exclusion assumption of Lemma 3. This implies that so do the D_i's in the specific set D. Lemma 3 guarantees that $\left|\bigcup_{i=1}^{\Phi} D_i\right| \leq 3\Phi\sqrt{\Phi}$. Since $\left|\bigcup_{i=1}^{\Phi} D_i\right| \geq \frac{1}{3}\left|\bigcup_{i=1}^{k\Phi} C_i\right|$ we conclude that $\left|\bigcup_{i=1}^{k\Phi} C_i\right| \leq 9\Phi\sqrt{\Phi}$. The lemma follows from the fact that $\sum_{i=1}^{\Phi} \zeta_i = \left|\bigcup_{i=1}^{k\Phi} C_i\right|$. □

If the number of units that become erasable is small, the flash endures.

Theorem 8. *If $H \gg 1$ then for $t \le (1-o(1))(kHn/9)^{2/3}$ the total number of erasures under this offline algorithm is $\sum_{i=1}^{n} h_i^{\mathcal{N}}(t) \le Hn - n$.*

Proof. It follows from Lemma 2 (by simple induction, using $|B_1| \le n$) that

$$|B_\Phi| \le n - \sum_{i=1}^{\Phi-1} |A_i| + 3(\Phi - 1) + \sum_{i=1}^{\Phi} \zeta_i \ .$$

From Lemma 1 we know that $|B_\Phi| + 1 \ge |A_\Phi|$. This implies

$$\sum_{i=1}^{\Phi} |A_i| \le 1 + n + 3(\Phi - 1) + \sum_{i=1}^{\Phi} \zeta_i \le 1 + n + 3(\Phi - 1) + 9\Phi\sqrt{\Phi} \ ,$$

where the last inequality follows from the previous lemma. Since $\sum_{i=1}^{\Phi} |A_i| = \sum_{i=1}^{n} h_i^{\mathcal{N}}(\Phi k)$ we get $\sum_{i=1}^{n} h_i^{\mathcal{N}}(\Phi k) \le 1+n+3(\Phi-1)+9\Phi\sqrt{\Phi}$ and conclude that $\sum_{i=1}^{n} h_i^{\mathcal{N}}(t) \le n + 3(t/k - 1) + 9(t/k)^{3/2}$. Thus, for $t \le (1-o(1))(kHn/9)^{2/3}$ it holds that $\sum_{i=1}^{n} h_i^{\mathcal{N}}(t) \le Hn - n$. □

We now use the algorithm for the single-slot unit wear-leveling problem to create an algorithm $\mathcal{N}_2$ which evens the wear among the units. $\mathcal{N}_2$ first simulates $\mathcal{N}$ and generates a new *single-slot* request sequence η. To avoid confusion, we call the blocks in η pseudo-blocks. We construct η as follows: whenever $\mathcal{N}$ erases unit i, we add a request for pseudo block i to η (since $\mathcal{N}$ may erase many units at once, we impose an arbitrary order on these erasures). Now $\mathcal{N}_2$ runs the offline algorithm for the single-slot case on η. When the single-slot offline algorithm switches blocks among two units, $\mathcal{N}_2$ switches the corresponding actual units.

Theorem 9. *If $H \gg 1$ then the wear $h_i^{\mathcal{N}_2}(t)$ of any unit i at time $t = (1-o(1))(2kHn/3)^{2/3}$ is at most H.*

The algorithm $\mathcal{N}_2$ is clearly not atomic. We can transform it into an atomic one in two ways.

Theorem 10. *There exist atomic algorithms $\mathcal{A}_1, \mathcal{A}_2$ such that for $H \gg 1$ the wear $h_i^{\mathcal{A}_1}(t_1), h_i^{\mathcal{A}_2}(t_2)$ of any unit i at times $t_1 = (1-o(1))(2kHn/3)^{2/3}$ and $t_2 = \Omega\left((2kHn/3)^{2/3}/\log k\right)$ is at most H. $\mathcal{A}_1$ requires $k+2$ empty units and $\mathcal{A}_2$ requires 3 empty units.*

The difficult non-atomic part of $\mathcal{N}$ is the sorting of the blocks in the dirty units. $\mathcal{A}_1$ uses a straightforward approach that requires $k+1$ empty units to preform this sorting. $\mathcal{A}_2$ does it using the merge-sort algorithm, using two extra units to hold the partial runs of sorted blocks that the algorithm constructs. It is not hard to see that two extra units are always sufficient, and that the sorting algorithm performs $O(k \log k)$ erasures. The same idea can be used to trade off any number of extra units for better endurance using multiway merge-sort. Both algorithms apply the atomic single-slot unit wear leveling algorithm, which requires another extra unit.

3.2 The Deterministic Online Fractional Problem

The endurance of any deterministic algorithm for the fractional problem depends on the number of extra slots, $nk - m$, just like in the single-slot problem.

Theorem 11. *For every deterministic algorithm α (even if α is non-atomic) there exists a sequence σ such that $\ell_\alpha(\sigma) \leq (nk - m + 1)(H + 1)$. Furthermore, there exists a deterministic non-atomic algorithm that achieves $\ell(\sigma) \geq (nk-m+1)(H+1)$ and an atomic variant that achieves $\ell(\sigma) \geq ((n-1)k-m+1)(H+1)$ using a single empty unit.*

3.3 The Randomized Online Fractional Problem

We now present a lower bound for randomized online algorithms for the fractional problem. The main ingredient that we analyze is the number of erasures preformed by the algorithm. The bound that we prove depends on the number of empty slots s. In particular, we require that $s < nk/2$. Otherwise, even a deterministic algorithm can achieve high endurance.

Theorem 12. *For every randomized online algorithm α (even if α is non-atomic) for the fractional wear leveling problem with $s = s(n)$ empty slots, such that $s \ll Hn$ and $s < nk/2$:*

- *If $s = n^{1-\epsilon}$ for some constant $0 < \epsilon < 1$ then there exists a constant $c > 0$ and σ such that $E[\ell_\alpha(\sigma)] < cHn$.*
- *If $\frac{n}{polylog(n)} \leq s \ll n \log n$ then there exists a constant $c > 0$ and σ such that*
$$E[\ell_\alpha(\sigma)] < cHn \cdot \frac{\log n}{\log(n \log n/s)} .$$
- *If $s = \Omega(n \log n)$ then there is a constant $c > 0$ and σ such that $E[\ell_\alpha(\sigma)] < cHs$.*

The proof's idea is to observe a random request sequence and split it into phases of size $2s$. Since there are only s empty slots, the algorithm must clean at least s slots during each phase. Because the requests are random, during each phase there is not likely to be a unit with many dirty slots. Thus, the algorithm will be forced to perform many erasures.

4 Conclusions

Our analysis shows that Ban's simple randomized algorithm [2] is nearly optimal (both in the competitive sense and in the absolute sense) for the single-slot wear-leveling problem. The competitive performance of deterministic algorithms for the same problem is poor, although many flash-based systems do use deterministic algorithms. The effectiveness of deterministic algorithms in practice is probably due to the fact that request sequences are oblivious to the online algorithm.

Our analysis of the fractional wear-leveling problem leads to two conclusions. First, the fact that offline algorithms outperform online algorithms implies that in practice, it is advantageous to try to cluster blocks according to the expected time of their next modification. Second, the analysis justifies the separation of the erasure-minimization policy from the wear-leveling policy.

References

1. Mahmud Assar, Siamack Nemazie, and Petro Estakhri. Flash memory mass storage architecture incorporation wear leveling technique. US patent 5,479,638, US patent 5,388,083 (slightly different title), and US patent 5,485,595 (slightly different title), all filed 1993, issued 1995/6, and assigned to Cirrus Logic, 1993.
2. Amir Ban. Wear leveling of static areas in flash memory. US patent 6,732,221, filed 2001, issued 2004, and assigned to M-Systems, 2001.
3. Avraham Ben-Aroya. Competitive analysis of flash-memory algorithms. Master's thesis, School of Computer Science, Tel-Aviv University, April 2006. Available online at `www.tau.ac.il/~abrhambe`.
4. Ricardo H. Bruce, Ronaldo H. Bruce, Earl T. Cohen, and Allan J. Christie. Unified re-map and cache-index table with dual write-counters for wear-leveling of non-volitile flash ram mass storage. US patent 6,000,006, December 1999. Filed August 25, 1997; Issued December 7, 1999; Assigned to BIT Microsystems.
5. M.-L. Chiang and R.-C. Chang. Cleaning policies in mobile computers using flash memory. *The Journal of Systems and Software*, 48(3):213–231, 1999.
6. Mei-Ling Chiang, Paul C.H. Lee, and Reui-Chuan Chang. Using data clustering to improve cleaning performance for flash memory. *Software—Practice and Experience*, 29(3), 1999.
7. Petro Estakhri, Mahmud Assar, Robert Reid, Alan, and Berhanu Iman. Method of and architecture for controlling system data with automatic wear leveling in a semiconductor non-volitile mass storage memory. US patent 5,835,935, 1998. Filed September 13, 1995; Issued November 10, 1998; Assigned to Lexar Media.
8. Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys*, 37:138–163, 2005.
9. Sang-Wook Han. Flash memory wear leveling system and method. US patent 6,016,275, January 2000. Filed November 4, 1998; Issued January 18, 2000; Assigned to LG Semiconductors.
10. Edwin Jou and James H. Jeppesen III. Flash memory wear leveling system providing immediate direct access to microprocessor. US patent 5,568,423, October 1996. Filed April 14, 1995; Issued October 22, 1996; Assigned to Unisys.
11. Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A flash-memory based file system. In *Proceedings of the USENIX 1995 Technical Conference*, pages 155–164, New Orleans, Louisiana, January 1995.
12. Karl M. J. Lofgren, Robert D. Norman, Gregory B. Thelin, and Anil Gupta. Wear leveling techniques for flash EEPROM systems. US patent 6,081,447 and US patent 6,594,183, filed 1998/1999, issued 2000/2003, and assigned to Western Digital and Sandisk, 1998.
13. Steven E. Wells. Method for wear leveling in a flash EEPROM memory. US patent 5,341,339, 1994. Filed November 1, 1993; Issued August 23, 1994; Assigned to Intel.

Contention Resolution with Heterogeneous Job Sizes*

Michael A. Bender[1], Jeremy T. Fineman[2], and Seth Gilbert[2]

[1] Dept. of Computer Science, Stony Brook University, NY 11794-4400, USA
bender@cs.sunysb.edu
[2] CSAIL, Massachusetts Institute of Technology, Cambridge, MA 02139, USA
{jfineman, sethg}@mit.edu

Abstract. We study the problem of contention resolution for different-sized jobs on a simple channel. When a job makes a run attempt, it learns only whether the attempt succeeded or failed. We first analyze binary exponential backoff, and show that it achieves a makespan of $V2^{\Theta(\sqrt{\log n})}$ with high probability, where V is the total work of all n contending jobs. This bound is significantly larger than when jobs are constant sized. A variant of exponential backoff, however, achieves makespan $O(V \log V)$ with high probability. Finally, we introduce a new protocol, size-hashed backoff, specifically designed for jobs of multiple sizes that achieves makespan $O(V \log^3 \log V)$. The error probability of the first two bounds is polynomially small in n and the latter is polynomially small in $\log V$.

1 Introduction

Randomized backoff is a common mechanism for reducing contention on a shared resource. Processes/jobs make competing attempts to access the resource, but only one can gain control of the resource at a time. If an access attempt fails due to contention, then that process waits for a random amount of time before trying again. On subsequent failed attempts, the waiting time increases, thereby reducing the probability of a collision and increasing the chance of successful resource acquisition.

Backoff is used in many contexts, for example, network access (e.g., an Ethernet bus [1]), wireless communication [2], transactional memory [3], and speculative-lock elision [4]. In these and other applications of randomized backoff, the lengths of jobs fluctuate substantially. Most theoretical analyses, however, assume unit-length jobs. In a transactional shared-memory system, for example, jobs (transactions) can vary by four to five orders of magnitude [5]. In a wireless network, jobs (packet transmissions) can vary by over three orders of magnitude. The job length is proportional to both transmission length (in bits) and

* This research was supported in part by the Singapore-MIT Alliance, NSF Grants CCR-0208670, ITR-0121277, CNS-0305606, OCI-0324974, and by USAF/AFRL Award #FA9550-04-1-0121.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 112–123, 2006.

the transmission speed; the speed of the transmitters alone varies considerably (e.g., from roughly 10Kb/s to 10Mb/s).

This paper gives the first theoretical analysis of randomized backoff when jobs have variable sizes. We analyze a system consisting of jobs $1, \ldots, n$. Job i has size $t_i \geq 1$, which indicates that i must run for t_i consecutive units of time in order to complete. We define the ***volume*** of the jobs as $V = \sum_{i=1}^{n} t_i$. Each job knows its own size, but does not know any other job size or the number of other jobs. For simplicity, we assume that t_i is integral and that time is divided into unit-sized time slots, but our analyses extend to the case of nonintegral sizes.

The jobs are competing for access to a ***simple channel*** and have no other means of communication. Whenever a job of size t_i makes a run attempt, it must execute for the full t_i consecutive timeslots. If a job's run is uncontested, then the job completes successfully. If multiple jobs make overlapping run attempts, then all attempts fail and the jobs must retry. A job i learns whether its run attempt is successful only *after* the full t_i time slots, not instantly when the collision occurs. A job gains information only by making run attempts—there is no "listening" on the channel. No other information (e.g., the number of jobs that made attempts in a time slot) is available to the job. (These assumptions are roughly the worst case, in terms of information learned when a collision occurs.)

In this paper, we consider the ***batch problem*** (also called the ***control-tower problem*** [6] or ***shopping-cart problem*** [7,8]), where all jobs arrive at time 0. We analyze the worst-case ***makespan*** of the protocols, which is the maximum completion time among all the jobs.

This paper discusses ***windowed backoff protocols*** in which time is divided into a sequence of windows $\langle W_1, W_2, W_3, \ldots \rangle$. A job makes at most one run attempt in any window. Notice that a job can make a run attempt only if the window is larger than the job size. Even if a job does fit in a window, it may choose not to make a run attempt. If the job does choose to execute, it randomly chooses a position in the window such that there is sufficient time left for the job to execute fully within the window.

Results

Binary exponential backoff and generalizations. We begin by presenting results on binary exponential backoff. Since a single large job can slow down many small jobs, the performance for heterogeneous job sizes is significantly worse than for unit-sized jobs. We show that it achieves a makespan of $V2^{\Theta(\sqrt{\log n})}$ with error probability polynomially small in n. We next give a variant of exponential backoff that backs off more slowly and yields a makespan of $\Theta(V \log V)$ also with error probability small in n. A key tool is a tight analysis of "fixed-window backoff," where all windows have size $\Theta(V)$. These protocols achieve the specified makespan with error probability polynomially small in the number of jobs.

Size-hashed backoff. The principle result in this paper is a backoff protocol that achieves makespan $O(V \log^3 \log V)$. The main technique is to group jobs by size. Thus, we "hash" jobs based on their size to specific windows in which they can

make run attempts. An explicit construction, based on the modulo hash function results in a makespan of $O(V\sqrt{\log V}\log^2\log V)$. By grouping the job sizes using specially designed "good" hash functions, we obtain makespan $O(V\log^3\log V)$. We use the probabilistic method to show that such hash functions exist. These protocols achieve the specified makespans with error probability polynomially small in $\log V$.

Related Work

The most closely related work is that of Gereb-Graus and Tsantilas [9] (see also [10]) and Bender et al. [11]. Gereb-Graus and Tsantilas [9] show that for unit-size jobs in the batch setting, there is a backoff-backon protocol (which is sometimes called "sawtooth") that achieves an optimal makespan of $O(n)$; a similar backoff-backon approach also appears in Greenberg and Leiserson [10] in the context of routing. Bender et al. [11] analyze fixed backoff, exponential backoff, polynomial backoff, and optimal monotone backoff in the batch setting; they analyze exponential backoff in an adversarial queuing-theory setting. For binary exponential backoff ($W_i = 2^i$) with unit-size jobs, they prove a makespan of $\Theta(n\log n)$. (With variable-length jobs the situation is quite different; see Theorems 3 and 4.) Batch arrivals have been considered by several other authors [12, 13, 14, 15] with the goal of routing h-relations, involving multiple channels.

In the wireless-networking literature, this batch problem is known as the ***shopping-cart problem*** [7, 8] and models a shopping cart full of items with RFID tags passing through a sensor all at the same time. Currently implemented protocols are far from achieving the linear makespan described in [9, 10].

2 Traditional Backoff with Variable-Sized Jobs

In this section we analyze classic backoff protocols. We first consider ***fixed backoff***, where the window size is fixed at $\Theta(V)$. (This models the case where an estimate of the volume is known in advance.) We then turn to ***binary exponential backoff***, where the window size repeatedly doubles, i.e., $W_{i+1} = 2W_i$. If a job fits in window, it makes a random run attempt. In both cases the makespan of these strategies is worse for variable-size jobs than for unit-size jobs, with binary exponential backoff significantly worse. We end by giving a faster monotone backoff strategy, whose performance matches fixed backoff to within constant factors, even when the volume V is not known in advance.

Fixed-Volume Backoff

We first analyze the protocol FIXED-BACKOFF$_W$, where the volume V of the jobs is known in advance. FIXED-BACKOFF$_W$ is the windowed protocol in which $W_i = W = \Theta(V)$, where W is the (unchanging) window size throughout the protocol. We first show that FIXED-BACKOFF$_{\Theta(V)}$ has the following lower bound:

Theorem 1. *Let $W = (1+\varepsilon)V$ for any constant $\varepsilon \in \mathbb{R}^+$. There exists n sufficiently large such that the makespan of* FIXED-BACKOFF$_W$ *is $\Omega(W \log n)$ with error probability polynomially small in n.*

Proof (sketch). Consider an execution with one large job of size $n+1$ and $n-1$ small jobs of size 1. As long as a polylogarithmic number of small jobs remain, at least one small job collides with the large job (w.h.p.); hence the large job does not complete. As long as the large job remains, only a constant fraction of small jobs completes (w.h.p.). Applying a Chernoff bound concludes the proof. □

We now give a matching upper bound for FIXED-BACKOFF$_W$, when $W \geq 3V$.

Theorem 2. *Let $W = \alpha V$, for any $\alpha \geq 3$. Then the makespan of* FIXED-BACKOFF$_W$ *is $O(W \log n)$ with error probability polynomially small in n.*

Proof (sketch). In each round, we argue that a constant fraction of the jobs completes. First, notice that a constant fraction of the jobs are "small," i.e., less than twice the average size. Next, notice that a constant fraction of the small jobs completes: the big jobs can only block $2V$ of the window; the remaining V space is sufficient for each small job to complete with constant probability. □

Notice the difference between fixed backoff in the variable-size and the unit-size case. If all jobs are unit size, then the makespan is $n \lg n \pm O(n)$ with high probability [11]. Moreover, the makespan *improves* when the window size dips slightly below the volume $V = n$, say to $W = 3n/\lg\lg\lg n$, at which point the makespan attains its optimal value of $\Theta(n \log\log n/\log\log\log n)$ [11]. With variable-length jobs, the makespan grows arbitrarily large if $W = V$.

Exponential Backoff

We next analyze binary exponential backoff with variable-size jobs. In binary exponential backoff, $W_i = 2^i$, for $i = 1, 2, \ldots$, and for any job j in the system, j must make a run attempt in window W_i, as long as $t_j \leq W_i$.

Theorem 3. *Consider n jobs with total volume V running binary exponential backoff. The makespan is $V 2^{O(\sqrt{\log n})}$ with error probability polynomially small in n (for sufficiently large n).*

Proof (sketch). In each round we divide the jobs into classes of "small" and "large" jobs. We show that as the window size increases, an increasingly large fraction of jobs completes (with high probability). Specifically, "small" is defined to include an increasingly large fraction of all jobs, and an increasingly large fraction of small jobs complete in each round (with high probability). Within $O(\sqrt{\log n})$ rounds after the window size is W, there are only $O(\log n)$ jobs remaining. Another argument shows that these $O(\log n)$ stragglers complete in the next $O(\sqrt{\log n})$ rounds, with high probability. □

We now give a lower bound on the performance of binary exponential backoff.

Theorem 4. *There exists an instance of* $(c+3)m\ln m+1$ *jobs for which the makespan of exponential backoff is* $\Omega(V2^{\sqrt{\lg V}/2})$ *rounds with probability* $(1-1/m^c)$*, for any* $c>1$*.*

Proof (sketch). Consider an instance with one large job of size m and $(c+3)m\ln m$ small jobs of size 1, resulting in a total volume of $V=(c+3)m\ln m+m$. There are two regimes, which we analyze separately. While $W_i<m$, the *small-window regime*, only small jobs attempt to execute. When $W_i\geq m$, the *large-window regime*, the large job also attempts to execute. No job completes in the small-window regime (w.h.p.) since the small jobs collide with each other. For the first $\Omega(\sqrt{\log m})$ windows of the large-window regime, there exists some small job that collides with the large job in each window (w.h.p.), since the large job blocks a geometrically decreasing fraction of the window. □

Optimized Exponential Backoff

We develop a variant of exponential backoff that achieves better performance by backing off more slowly, nearly matching the performance of fixed backoff.

The idea is to double window sizes (as in exponential backoff) but only after repeating a window of size W $\Theta(\log W)$ times, allowing all jobs to complete when W is an accurate guess of V. Thus, we effectively back off by a factor of only $1+O(1/\log V)$ (rather than 2 as with binary exponential backoff). This algorithm matches the asymptotic performance of FIXED-BACKOFF$_{\Theta(V)}$:

Theorem 5. *There exists a parameter choice for exponential backoff achieving makespan* $O(V\log V)$ *with high probability, i.e., error probability polynomially small in* n*.* □

3 Size-Hashed Backoff

This section describes more efficient backoff protocols that improve on the traditional ones analyzed in Section 2. The main difficulty in dealing with different-sized jobs is that larger jobs are not likely to succeed until enough of the smaller jobs complete. This fact is exploited in Theorem 1's proof, where just one large job interferes with all the other jobs. The approach in this section groups jobs by size so that jobs with different sizes cannot interfere with each other for too long. In particular, we divide jobs into $\lceil\lg V\rceil$ ***job classes*** based on size. Jobs of size t_i belong to the $(\lceil\lg t_i\rceil+1)$th job class.

We first review a "backon" protocol for constant-sized jobs, which forms a subcomponent of our new strategy. We then overview the general strategy for size-hashed backoff. Next, we discuss the mapping "hash" functions (for which the protocol is named). We present the detailed protocol and two specific mapping functions resulting in specific instantiations of size-hashed backoff. Applying our first mapping yields a protocol with makespan $O(V\sqrt{\log V}\log^2\log V)$. We then show the existence of a mapping that achieves $O(V\log^3\log V)$ makespan. Both of these versions achieve the specified makespan with probability $1-1/\log^c V$ for any constant $c>1$ with a linear dependence on c in the makespan.

Backon Protocol for Constant-Sized Jobs

A key component of our strategy is the DESCEND "backon" subprotocol. The protocol (i.e., the participating jobs) takes three parameters: (1) *jclass*, the job class, (2) W, the window size, and (3) r, a number of repetitions. It guarantees that if m processes in the same job class, having total volume V', $V' < W$, all start DESCEND at the same time, then within $O(rW)$ time all the jobs finish with probability $1 - 1/2^r$. The main idea is that once the window has size $3V'$, then a constant fraction of the jobs should complete. At this point, the protocol can "back on," using a window that is a constant fraction smaller. The process continues shrinking the window size until it has decreased to $W/\lg W$. After that point, we repeat the $W/\lg W$-sized window approximately $\lg W$ times. In order to achieve the desired probability, this entire process is repeated r times.

Since a close variant of DESCEND has been previously analyzed by Greb-Graus and Tsantilas [9], we omit the proof here. (It also follows from Lemma 3.)

Overview of Size-Hashed Backoff

As in exponential backoff, size-hashed backoff proceeds by repeated doubling on the estimated volume. We refer to each iteration as a ***round***. The algorithm completes in (or before) the first round in which the estimated volume is sufficiently large ($V' > V$). Each round of the protocol proceeds in ***phases***. When the estimated volume is sufficiently large, in each phase the number of ***nonempty*** job classes—those with jobs remaining—is reduced by a constant fraction.

In the first phase, each job class runs separately. That is, we take a time interval of size $\Theta(rV)$, where $r = \Theta(\log\log V)$ is a number of repetitions for the DESCEND protocol, and divide it into $\lg V$ size-$\Theta(rV/\log V)$ "buckets," one for each job class. Specifically, bucket i is designated for the jobs in job class i (i.e., those jobs j with size $2^{i-2} < t_j \leq 2^{i-1}$). During the ith bucket, each job in the ith job class runs the DESCEND subprotocol for time $\Theta(rV/\log V)$. If the volume in the job class is small enough—specifically, $O(V/\log V)$—then that job class ***completes***, i.e., becomes empty. Since the volume is distributed among various job classes, a constant fraction of the job classes have small enough volume to complete. In particular, a simple counting argument shows that at least 1/2 of the $\lg V$ job classes have volume at most $2V/\lg V$. We conclude that after $O(rV) = O(V\log\log V)$ time, at least half the job classes are empty.

It would be ideal if, during a second phase, we could allocate buckets for only the nonempty job classes. Since at least half the job classes are empty, we can, in principle, allocate half as many buckets of twice the size and run DESCEND for each bucket. Once again, at least half of the job classes have a small enough volume to complete. After $\lg\lg V$ phases following this process, there is a single nonempty job class in a $\Theta(rV)$-size bucket, and hence this last job class completes. Since each of the phases takes time $\Theta(V\log\log V)$, the resulting makespan is $O(V\log^2\log V)$.

The problem with this approach is that jobs have no *a priori* knowledge as to which job classes become empty during a given phase, and they cannot observe this information. Surprisingly, we can still resurrect the spirit of this idea.

To generalize, we create a mapping from $\lg V$ job classes to a set of buckets smaller than $\lg V$. Given any small set of nonempty job classes, the mapping has the property that a constant fraction are assigned to their own bucket,[1] thus allowing them to complete using DESCEND. To ensure this property, we use extra buckets, resulting in a makespan of $O(V \log^3 \log V)$ for our fastest protocol.

Mapping Job Classes to Buckets

We define the mapping problem more formally. We are given η objects $X = \{x_1, x_2, \ldots, x_\eta\}$ and some integer $m < \eta$. Consider a mapping $F_{m,\eta} : X \to \wp(B)$ of objects to subsets of buckets $B = \{B_1, B_2, \ldots\}$. For example, $F_{m,\eta}(x_1) = \{B_1, B_7, B_{10}\}$ indicates that object x_1 maps to buckets B_1, B_7, and B_{10}.

A mapping is an ***α-good mapping***, with $0 < \alpha \leq 1$, if for all size-m subsets $Y = \{y_1, y_2, \ldots, y_m\} \subseteq X$, there exists a size-$\lceil \alpha m \rceil$ subset of Y in which each object is assigned its own bucket. More formally, $\exists Z \subseteq Y$ where $|Z| = \alpha m$ and $\forall z \in Z, \exists b \in F_{m,\eta}(z)$ s.t. $b \notin \bigcup_{y \in Y \setminus z} F_{m,\eta}(y)$.

This "good mapping" property is exactly what we need for size-hashed backoff. In the backoff setting, $\eta = \lceil \lg V \rceil$ is the number of job classes. In any phase, we maintain an estimate of the number m of nonempty job classes; we do not, however, know which classes are nonempty. We want at least a constant fraction of them to end up assigned to their own buckets. We can then ensure that a constant fraction of job classes complete. For example, if a phase has buckets of size $2rV/m$ (i.e., at most $m/2$ job classes are "too big" to complete) and the mapping is a 3/4-good mapping, then at least $m/4$ of the nonempty job classes must be small enough to complete in a bucket *and* be mapped to a unique bucket.

Our size-hashed backoff algorithm considers good mappings of a simplified form, making the functions easier to think about. Rather than having arbitrary functions from objects to bucket sets, we split the buckets into "collections" of consecutive buckets. Each object is mapped to exactly one bucket in each collection. We construct our mapping $\mathcal{F}_{m,\eta}$ as a sequence of functions $\mathcal{F}_{m,\eta} = \{f_{m,\eta,1}, f_{m,\eta,2}, \ldots, f_{m,\eta,s_{m,\eta}}\}$ such that $f_{m,\eta,i} : X \to B$ maps an object to a single bucket in the ith collection. We define $s_{m,\eta} = |\mathcal{F}_{m,\eta}|$ to be the ***size*** of the set of functions in our mapping $\mathcal{F}_{m,\eta}$. Adding more functions to $\mathcal{F}_{m,\eta}$ increases the chance of achieving α-goodness. We define $r_{m,\eta,i}$ to be the ***range*** of, or the number of buckets used by, the function $f_{m,\eta,i}$.

Size-Hashed Protocol

We now give the protocol for size-hashed backoff in more detail, assuming an α-good mapping $\mathcal{F}_{m,\eta} = \{f_{m,\eta,i}\}$. (Pseudocode for the size-hashed protocol is given below.) We argue that all jobs eventually make successful run attempts with probability at least $1 - 1/\lg^c V$ for any constant $c > 1$. The makespan, however,

[1] This property is similar to the collision property of a hash function. It also appears to have close connections to expanders, specifically lossless, bipartite expanders.

```
SIZE-HASHED-BACKOFF(t_i) ▷ t_i is the job size of process i.
 1  V' ← 1
 2  jclass ← ⌈lg t_i⌉ + 1
 3  repeat  ▷ Each iteration is a round.
 4          V' ← 2V'
 5          η ← lg V'
 6          m ← η ▷ m bounds the number of nonempty job classes.
 7          repeat ▷ Each iteration is a phase.
 8                  wsize ← c_1 V'/m ▷ Window size for DESCEND.
 9                  bsize ← c_3 wsize lg lg V' ▷ Bucket for DESCEND iteration.
                    ▷ s_{m,η} is the number of functions in the mapping.
                    ▷ Iterate over subphases/functions.
10                  for i ← 1 to s_{m,η}
11                      do ▷ r_{m,η,i} is the number of buckets used by f_{m,η,i}.
                           ▷ Iterate over buckets.
12                         for bucket ← 1 to r_{m,η,i}
13                             do if f_{m,η,i}(jclass) = bucket
14                                then DESCEND(jclass, wsize, c_3 lg lg V')
15                                else Wait bsize time.
16                  m ← ⌊m/c_2⌋
17            until m = 0 ▷ End loop over phases.
18    until job i executes ▷ Ends the loop over rounds.
```

depends on the size and range of the mapping, so we defer that discussion to the particular variants later in the section.

Recall that size-hashed backoff executes in rounds (lines 3–18), and we repeatedly double the estimated volume in each round (line 4). Each round is divided into phases (lines 7–17), and in each phase we expect a constant fraction of the job classes to complete using the α-good mapping $\mathcal{F}_{m,\eta}$. Each phase is subdivided into ***subphases*** (lines 10–15) which correspond to each function $f_{m,\eta,i}$ in the mapping $\mathcal{F}_{m,\eta}$, so each job class maps to exactly one bucket in each subphase. The α-goodness property guarantees that at least αm of the m nonempty job classes are assigned to unique buckets. The buckets use the geometrically-decreasing DESCEND protocol to ensure that jobs complete when (1) the buckets are large enough, and (2) $\mathcal{F}_{m,\eta}$ assigns a unique bucket (line 14).

Consider the ith phase, during which there should be (at most) m nonempty job classes remaining. During this phase, the protocol creates $s_{m,\eta}$ subphases, where subphase j uses $r_{m,\eta,j}$ buckets of size $bsize = \Theta(rV/m)$ (lines 8–9). Thus, the total length of the ith phase is $\sum_{j=1}^{s_{m,\eta}} r_{m,\eta,j}\Theta(rV/m)$. To understand what these numbers mean, consider the "ideal" mapping in which each job knows exactly which job classes are empty; in this case $s_{m,\eta} = 1$ and $r_{m,\eta,1} = m$, giving a total phase length of $\Theta(rV) = \Theta(V \log\log V)$.

The following theorem states that SIZE-HASHED-BACKOFF completes all the jobs in $\lg V + O(1)$ rounds (i.e., when the window size is $\Theta(V)$). We later analyze the length of each round—and hence the makespan—in the context of the specific family of mappings $\mathcal{F}$, which determines the number of buckets.

Theorem 6. *Suppose n jobs with volume V execute* Size-Hashed-Backoff*, beginning at the same time. Suppose also that $\mathcal{F}$ is an α-good mapping for some constant α. If we set $c_1 = 2/\alpha$, $c_2 = 2/(2-\alpha)$, and $c_3 = c+2$, where c_1, c_2, c_3 are the constants from the pseudocode, then all n jobs complete before the $(\lg V + O(1))$th round with probability at least $1 - 1/\lg^c V$, for any $c \geq 1$.*

Proof (sketch). We show the following invariant holds with sufficient probability: if $V' > V$, then m is an upper bound on the number of nonempty job classes. Initially, there are $\leq m = \eta = \lg V'$ job classes. We proceed by induction. Since the total volume of jobs is $O(V')$, there can be at most $\lfloor m/c_1 \rfloor$ job classes with volume $> \Theta(c_1 V'/m)$. Since $\mathcal{F}$ is α-good, at most $m - \lceil \alpha m \rceil$ of the nonempty job classes *do not* map to their own bucket. Thus, there are at most $m - \lceil \alpha m \rceil + \lfloor m/c_1 \rfloor \leq \lfloor m/c_2 \rfloor$ job classes that are too large or collide. These job classes do not (necessarily) complete during the phase. Any other job class completes during the Descend protocol: each job class completes with probability $1 - 1/\lg^{c_3} V' > 1 - 1/\lg^{c_3} V$. Taking a union bound across all $\lceil \lg V \rceil$ job classes and $\Theta(\log \log V)$ phases maintains the invariant with probability at least $1 - 1/\lg^{c_3 - 2} V$. □

We now provide two α-good mappings, and analyze the resulting performance.

Analysis of a 1-Good Mapping

In this section we present a 1-good mapping based on a simple modulo function, which results in a makespan of $\Theta(V\sqrt{\log V}\log^2 \log V)$.

Let $g_{m,\eta,i}$ be the identity function: $g_{m,\eta,i}(x_j) = j$. (That is, the j^{th} object maps to bucket j.) Recall that each function $g_{m,\eta,i}$ maps objects to exactly one bucket in collection i. Notice that each collection contains η buckets. Let $f_{m,\eta,i}(x_j) = j \pmod i$. Notice that the ith collection contains i buckets. We define a 1-good mapping, parameterized by a variable t (defined later), as follows:

$$\mathcal{F}_{m,\eta} = \begin{cases} \{g_{m,\eta,1}\} & : \text{ if } m > \eta/t \\ \left\{f_{m,\eta,1}, f_{m,\eta,2}, \ldots, f_{m,\eta,\Theta(m\log\eta)}\right\} & : \text{ if } m \leq \eta/t\ . \end{cases}$$

Lemma 1. *The functions $\mathcal{F}_{m,\eta}$ are a 1-good mapping.*

Proof (sketch). Notice that if $m > \eta/t$, $\mathcal{F}$ is the identity mapping, which is 1-good. Assume $m \leq \eta/t$. Consider any two objects $x_j \neq x_k \in X$. Consider C prime numbers $p_1, p_2, \ldots, p_C$, each of which is $\geq m \lg \eta$, and suppose by contradiction they collide everywhere, i.e., $f_{m,\eta,p_\ell}(x_j) = f_{m,\eta,p_\ell}(x_k)$ for all $\ell \in 1, 2, \ldots, C$. Then the difference between j and k must be divisble by each of these prime numbers, and hence at least $(m \lg \eta)^C$. Choose $C > \lg \eta / \lg(m \lg \eta)$, implying $(m \lg \eta)^C > \eta$. This is a contradiction, since $|j - k|$ can be at most η. Thus, x_j and x_k can collide in at most $C - 1$ of the functions $\{f_{m,\eta,p_\ell}\}$.

Recall that there are $\Theta(m \log \eta)$ functions $f_{m,\eta,i}$. Thus for a sufficiently large constant in the Θ notation, there are at least $mC = m \lg \eta / \lg(m \lg \eta)$ functions $f_{m,\eta,i}$ in which i is prime and $i \geq m \lg \eta$. For a given set Y of size $\leq m$ and a given object $x_j \in Y$, there must be one of the mC functions in which x_j does not collide with any of the $\leq m$ objects in Y, implying that $\mathcal{F}$ is 1-good. □

We now calculate the running times of each round:

Lemma 2. *The running time for a single round of size-hashed backoff, with 1-good mapping $\mathcal{F}$ is $O(V'\sqrt{\log V'}\log^2 \log V')$.*

Proof (sketch). First, consider a phase in which the number of job classes $m > \eta/t$. Recall that in this case, the number of collections $s_{m,\eta} = 1$ and the number of buckets in a collection $r_{m,\eta,1} = \eta$. Thus, the running time of the phase is $r_{m,\eta,1}\, bsize = \Theta(\eta V' \log\log V'/m)$ (lines 8–9). Since m decreases geometrically (by c_2 in each phase), the sum of running times of all phases with $m > \eta/t$ can be bounded by the phase with minimum m, which is thus $O(tV' \log\log V')$.

Consider a phase where $m \leq \eta/t$. Then the number of collections $s_{m,\eta} = \Theta(m \log \eta)$ and the number of buckets per collection $r_{m,\eta,i} = i$. Thus, a phase completes in $bsize \sum_{i=1}^{s_{m,\eta}} r_{m,\eta} = bsize\, O(m^2 \log^2 \eta)$ time. Substituting for $bsize$ and η, we have $O(V' \log\log V' m^2 \log^2 \eta/m) = O(mV' \log^3 \log V')$. Since m decreases geometrically, the sum of running times of all phases with $m \leq \eta/t$ can be bounded by the phase with maximum m, which is thus $O(V' \log V' \log^3 V'/t)$.

Thus the total duration of a round is $\Theta(tV' \log\log V' + V' \log V' \log^3 V'/t)$. Setting $t = \sqrt{\log V'} \log\log V'$ yields a time of $\Theta(V'\sqrt{\log V'}\log^2 \log V')$. □

Theorem 6 shows that size-hashed backoff, when using this 1-good function, terminates of $\lg V + O(1)$ rounds. Together with Lemma 2, we can conclude:

Corollary 1. *Assume that n jobs with volume V begin executing size-hashed backoff with 1-good mapping $\mathcal{F}$ at the same time. Then all n jobs make a successful run attempt in time $O(V\sqrt{\log V}\log^2 \log V)$ with probability at least $1 - 1/\lg^c V$, for any $c \geq 1$ and sufficiently large V.* □

Analysis of a 1/2-Good Mapping

Our final version of size-hashed backoff achieves a makespan of $O(V \log^3 \log V)$. This algorithm relies on a more efficient α-good mapping, which we show exists using the probabilistic method. The goal of this section is to prove the existence of a 1/2-good mapping where $s_{m,\eta} = \Theta(\log\log V)$ and $r_{m,\eta,i} = \Theta(m)$. This results, as corollary of Theorem 6, in a makespan of $O(V \log^3 \log V)$.

Notice that there are three $\log\log V$ factors in the makespan. Two of these come from the general structure of size-hashed backoff: there are $\Theta(\log\log V)$ phases reducing the number of nonempty job classes, and DESCEND runs for $\Theta(\log\log V)$ windows. The third $\log\log V$ factor arises from the number of functions $(s_{m,\eta})$. We first present a preliminary "balls and bins" lemma:

Lemma 3. *Assume you have m balls thrown uniformly at random into cm bins, $c > 15$. Then for some $0 < \delta < 1$, the probability that fewer than $m/2$ bins have exactly one ball is $\leq \delta^{cm}$.* □

We can now show that there exist appropriate 1/2-good functions:

Theorem 7. *There exists a set of 1/2-good functions $\mathcal{F}_{m,\eta} = \{f_{m,\eta,i}\}$ where the range $r_{m,\eta,i} = \Theta(m)$ and the number of subphases $s_{m,\eta} = \Theta(\lg \eta)$.*

Proof (sketch). We show the existence of the functions $\{f_{m,\eta,i}\}$ using the probabilistic method. For each η and $m \leq \eta$, for each $i \in [1, s_{m,\eta}]$, choose $f_{m,\eta,i}$ at random: choose $f_{m,\eta,i}(j)$, $j \leq \eta$, uniformly at random from the range $[1, r_{m,\eta,i}]$. We show that with probability > 0, the resulting family of functions is 1/2-good.

First, we calculate for a fixed set Y of size at most m and a fixed $i \in [1 \ldots s_{m,\eta}]$, the probability that $f_{m,\eta,i}$ is 1/2-good. We consider each of the m values $f_{m,\eta,i}(j)$, $j \in Y$, as a ball that is thrown uniformly at random into $r_{m,\eta,i} = \Theta(m)$ bins. By Lemma 3, we know that for some $\delta < 1$, the probability that fewer than 1/2 the "balls" are in their own bin is $\leq \delta^{\Theta(n)}$. That is, the probability that $|\{j \in Y \mid \exists k \in Y, f_{m,\eta,i}(j) = f_{m,\eta,i}(k)\}| > |Y|/2$ is $\leq \delta^{\Theta(m)}$.

Therefore the probability that for all $i \in [1, s_{m,\eta}]$ 1/2-goodness is violated is $\leq \delta^{\Theta(m)s_{m,\eta}} \leq \delta^{\Theta(m \lg \eta)}$, since each function i is selected independently.

Finally, we compute the number of possible sets Y of size m. In particular, there are $\binom{m+\eta}{m} \leq \binom{2\eta}{m}$ ways to distribute m possible jobs over η job classes. This reduces to: $\binom{2\eta}{m} \leq \left(\frac{2e\eta}{m}\right)^m \leq e^m 2^{m \lg \eta}$. We apply a union bound over the possible sets Y: $\Pr[\mathcal{F}_{m,\eta}$ is not 1/2-good$] \leq \delta^{\Theta(m \lg \eta)} e^m 2^{m \lg \eta} < 1$. We conclude that with probability > 0, the randomly chosen $\mathcal{F}_{m,\eta}$ is 1/2-good. □

We now conclude by calculating the makespan as a corollary of Theorem 6:

Corollary 2. *Assume that n jobs with volume V begin executing size-hashed backoff at the same time. Suppose also that we use a 1/2-good mapping $\mathcal{F}$ with sizes $s_{m,\eta} = \Theta(\log \eta)$ and ranges $r_{m,\eta,i} = \Theta(m)$. Then all n jobs make successful run attempts in time $O(V \log^3 \log V)$ with probability at least $1 - 1/\lg^c V$ for any constant $c \geq 1$ and sufficiently large V.*

Proof (sketch). By Theorem 6, all jobs complete by round $\lg V + O(1)$ with appropriate probability. Each phase in the round takes time *bsize* $\sum_{i=1}^{s_{m,\eta}} r_{m,\eta,i} =$ *bsize* $\Theta(m \log \eta)$. Substituting for *bsize* and η yields phase length $\Theta(V \log^2 \log V)$. Summing over $\Theta(\log \log V)$ phases completes the proof. □

4 Conclusion

In this paper, we study randomized backoff protocols when jobs differ in size. We analyze binary exponential backoff and show that it performs poorly, yielding makespan $V 2^{\Theta(\sqrt{\log n})}$. A slower rate of backoff achieves makespan $\Theta(V \log V)$. Our main results are size-hashed backoff protocols, where the backoff strategy depends on the job lengths; we reduce the makespan to only $O(V \log^3 \log V)$.

These results raise many questions. First, what are the lower bounds? Is a linear makespan possible? Next, on a simple channel jobs learn about contention only by making run attempts; what if jobs can listen on the channel without running? Also, a job i learns that a run attempt has failed only after the full t_i time steps. What if jobs learn of failure as soon as a collision occurs, enabling them to abort early? Can exponential backoff and its variants perform better?

This paper considers the batch problem, where jobs arrive at time 0. Ultimately we hope to understand the online problem, where jobs arrive over time.

What can be proved for queuing-theory arrivals? Are there reasonable worst-case models, similar to those in [11], that apply to backoff with different-size jobs?

Acknowledgments. We would like to thank Martin Farach-Colton, Bradley C. Kuszmaul, and Jelani Nelson for helpful conversations and feedback.

References

1. Metcalfe, R.M., Boggs, D.R.: Ethernet: Distributed packet switching for local computer networks. CACM **19**(7) (1976) 395–404
2. Abramson, N.: The ALOHA system — another alternative for computer communications. In: Proc. of AFIPS FJCC. Volume 37. (1970) 281–285
3. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: Proc. of the 20th Intnl. Conference on Computer Architecture., San Diego, California (1993) 289–300
4. Rajwar, R., Goodman, J.R.: Speculative lock elision: Enabling highly concurrent multithreaded execution. In: Proc. of the 34th Annual Intnl. Symposium on Microarchitecture, Austin, Texas (2001) 294–305
5. Ananian, C.S., Asanović, K., Kuszmaul, B.C., Leiserson, C.E., Lie, S.: Unbounded transactional memory. In: Proc. of the 11th Intnl. Symposium on High-Performance Computer Architecture, San Franscisco, California (2005) 316–327
6. MacKenzie, P.D., Plaxton, C.G., Rajaraman, R.: On contention resolution protocols and associated probabilistic phenomena. JACM **45**(2) (1998) 324–378
7. Juels, A., Rivest, R.L., Szydlo, M.: The blocker tag: Selective blocking of RFID tags for consumer privacy. In: Conference on Computer and Communications Security. (2003) 103–111
8. Finkenzeller, K.: RFID Handbook: Fundamentals and Applications in Contactless Smart Cards and Identification. Second edn. John Wiley & Sons (2003) E-book at `books24x7.com`.
9. Geréb-Graus, M., Tsantilas, T.: Efficient optical communication in parallel computers. In: Proc. of the 4th Annual Symposium on Parallel Algorithms and Architectures. (1992) 41–48
10. Greenberg, R.I., Leiserson, C.E.: Randomized routing on fat-trees. Advances in Computing Research **5** (1989) 345–374
11. Bender, M.A., Farach-Colton, M., He, S., Kuszmaul, B.C., Leiserson, C.E.: Adversarial contention resolution for simple channels. In: 17th Annual Symposium on Parallelism in Algorithms and Architectures. (2005) 325–332
12. Greenberg, A.G., Winograd, S.: A lower bound on the time needed in the worst case to resolve conflicts deterministically in multiple access channels. JACM **32**(3) (1985) 589–596
13. Greenberg, A.G., Flajolet, P., Ladner, R.E.: Estimating the multiplicities of conflicts to speed their resolution in multiple access channels. JACM **34**(2) (1987) 289–325
14. Goldberg, L.A., Jerrum, M., Leighton, T., Rao, S.: Doubly logarithmic communication algorithms for optical-communication parallel computers. SIAM Journal on Computing **26**(4) (1997) 1100–1119
15. Goldberg, L.A., Matias, Y., Rao, S.: An optical simulation of shared memory. SIAM Journal on Computing **28**(5) (1999) 1829–1847

Deciding Relaxed Two-Colorability—A Hardness Jump*

Robert Berke and Tibor Szabó

Institute of Theoretical Computer Science, ETH Zürich, 8092 Switzerland
{berker, szabo}@inf.ethz.ch

Abstract. A coloring is proper if each color class induces connected components of order one (where the *order* of a graph is its number of vertices). Here we study relaxations of proper two-colorings, such that the order of the induced monochromatic components in one (or both) of the color classes is bounded by a constant. In a (C_1, C_2)-*relaxed coloring* of a graph G every monochromatic component induced by vertices of the first (second) color is of order at most C_1 (C_2, resp.). We are mostly concerned with $(1, C)$-relaxed colorings, in other words when/how is it possible to break up a graph into small components with the removal of an independent set.

We prove that every graph of maximum degree at most three can be $(1, 22)$-relaxed colored and we give a quasilinear algorithm which constructs such a coloring. We also show that a similar statement cannot be true for graphs of maximum degree at most 4 in a very strong sense: we construct 4-regular graphs such that the removal of any independent set leaves a connected component whose order is linear in the number of vertices.

Furthermore we investigate the complexity of the decision problem (Δ, C)-AsymRelCol: Given a graph of maximum degree at most Δ, is there a $(1, C)$-relaxed coloring of G? We find a remarkable hardness jump in the behavior of this problem. We note that there is not even an obvious monotonicity in the hardness of the problem as C grows, i.e. the hardness for component order $C + 1$ does not imply directly the hardness for C. In fact for $C = 1$ the problem is obviously polynomial-time decidable, while it is shown that it is NP-hard for $C = 2$ and $\Delta \geq 3$.

For arbitrary $\Delta \geq 2$ we still establish the monotonicity of hardness of (Δ, C)-AsymRelCol on the interval $2 \leq C \leq \infty$ in the following strong sense. There exists a critical component order $f(\Delta) \in \mathbb{N} \cup \{\infty\}$ such that the problem of deciding $(1, C)$-relaxed colorability of graphs of maximum degree at most Δ is NP-complete for every $2 \leq C < f(\Delta)$, while deciding $(1, f(\Delta))$-colorability is trivial: every graph of maximum degree Δ is $(1, f(\Delta))$-colorable. For $\Delta = 3$ the existence of this threshold is shown despite the fact that we do not know its precise value, only $6 \leq f(3) \leq 22$. For any $\Delta \geq 4$, (Δ, C)-AsymRelCol is NP-complete for arbitrary $C \geq 2$, so $f(\Delta) = \infty$.

We also study the symmetric version of the relaxed coloring problem, and make the first steps towards establishing a similar hardness jump.

* Research is supported in part by the SNF Grant 200021-108099/1.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 124–135, 2006.

1 Introduction

A function from the vertex set of a graph to a k-element set is called a *k-coloring*. The values of the function are referred to as *color*s. A coloring is called *proper* if the value of the function differs on any pair of adjacent vertices. Proper coloring and the *chromatic number* of graphs (the smallest number of colors which allow a proper coloring) are among the most important concepts of graph theory. Numerous problems of pure mathematics and theoretical computer science require the study of proper colorings and even more real-life problems require the calculation or at least an estimation of the chromatic number. Nevertheless, there is the discouraging fact that the calculation of the chromatic number of a graph or the task of finding an optimal proper coloring are both intractable problems, even fast approximation is probably not possible. This is one of our motivations to study relaxations of proper coloring, because in some theoretical or practical situations a small deviation from proper is still acceptable, while the problem could become tractable. Another reason for the introduction of relaxed colorings is that in certain problems the use of the full strength of proper coloring is an "overkill". Often a weaker concept suffices and provides better overall results.

In this paper we study various relaxations of proper coloring, which allow the presence of some small level of conflicts in the color assignment. Namely, we will allow vertices of one or more color classes to participate in one conflict or, more generally, let each conflicting connected component have at most C vertices, where C is a fixed integer, not depending on the order of the graph. Most of our results deal with the case of relaxed two-colorings.

To formalize our problem precisely we say that a two-coloring of a graph is (C_1, C_2)-*relaxed* if every monochromatic component induced by the vertices of the first color is of order at most C_1, while every monochromatic component induced by the vertices of the second color is of order at most C_2. Note that $(1, 1)$-relaxed coloring corresponds to proper two-coloring.

In the present paper we deal with the two most natural cases of relaxed two-colorings. We say *symmetric relaxed coloring* when $C_1 = C_2$ and *asymmetric* relaxed coloring when $C_1 = 1$. Symmetric relaxed colorings were first studied by Alon, Ding, Oporowski and Vertigan [1] and implicitly, even earlier, by Thomassen [18] who resolved the problem for the line graph of 3-regular graphs initiated by Akiyama and Chvátal [2]. Asymmetric relaxed colorings were introduced in [5].

Related relaxations of proper colorings. There are several other types of coloring concepts related to our relaxation of proper coloring.

In a series of papers Škrekovski [17], Havet and Sereni [8], and Havet, Kang, and Sereni [9] investigated the concept of *improper colorings* over various families of graphs. A coloring is called (k, l)-improper if none of the at most k colors induces a monochromatic component containing vertices of degree larger than l. Hence in an improper coloring the amount of error is measured in terms of the *maximum degree* of monochromatic components rather than in terms of their order.

Linial and Saks [15] studied low diameter graph decompositions, where the quality of the coloring is measured by the *diameter* of the monochromatic components. Their goal was to color graphs with as few colors as possible such that each monochromatic connected component has a small diameter.

Haxell, Pikhurko and Thomason [11] study the *fragmentability* of graphs introduced by Edwards and Farr [7], in particular for bounded degree graphs. A graph is called (α, f)-fragmentable if one can remove α fraction of the vertices and end up with components of order at most f. For comparison, in a $(1, C)$-relaxed coloring one must remove an independent set and end up with small components.

The problems. We study relaxed colorings from two points of view, extremal graph theory and complexity theory, and find that these points eventually meet for asymmetric relaxed colorings. We also make the first steps for a similar connection in the symmetric case. To demonstrate our problems, in the next few paragraphs we restrict our attention to asymmetric relaxed colorings; the corresponding questions are asked and partially answered for symmetric relaxed colorings, but there our knowledge is much less satisfactory.
On the one hand there is the purely graph theoretic question:

> For a given maximum degree Δ what is the smallest component order $f(\Delta) \in \mathbb{N} \cup \{\infty\}$ such that every graph of maximum degree Δ is $(1, f(\Delta))$-relaxed colorable?

On the other hand, for fixed Δ and C one can study the computational complexity question:

> What's the complexity of the decision problem: Given a graph of maximum degree Δ, is there a $(1, C)$-relaxed coloring?

Obviously, for the critical component order $f(\Delta)$ which answers the extremal graph theory question, the answer is *trivial* for the complexity question: every instance is a YES-instance. Note also, that for $C = 1$ the complexity question is polynomial-time solvable, as it is equivalent to testing whether a graph is bipartite.

In this paper we investigate the complexity question in the range between 1 and the critical component order $f(\Delta)$. We establish the monotonicity of the hardness of the problem in the interval $C \geq 2$ and prove a very sharp "hardness jump". By this we mean that the problem is NP-hard for every component order $2 \leq C < f(\Delta)$, while, of course, the problem becomes trivial (i.e. all instances are "YES"-instances) for component order $f(\Delta)$. It is maybe worthwhile to note that at the moment we do not see any *a priori* reason why the hardness of the decision problem should even be monotone in the component order C, i.e. why the hardness of the problem for component order $C + 1$ should imply the hardness for component order C. In fact the problem is obviously polynomial-time decidable for $C = 1$, while for $C = 2$ we show NP-completeness.

The other main contribution of the paper concerns the extremal graph theory question and obtains significant improvements over previously known bounds

and algorithms. This result becomes particularly important in light of our NP-hardness results, as the exact determination of the place of the jump from NP-hard to trivial gets within reach.

To formalize our theorems we need further definitions. Let us denote by (Δ, C)-AsymRelCol the decision problem whether a given graph G of maximum degree at most Δ allows a $(1, C)$-relaxed coloring. Analogously, let us denote by (Δ, C)-SymRelCol the decision problem whether a given graph G of maximum degree at most Δ allows a (C, C)-relaxed coloring. Note here that both $(\Delta, 1)$-AsymRelCol and $(\Delta, 1)$-SymRelCol is simply testing whether a graph of maximum degree Δ is bipartite.

The asymmetric problem. For $\Delta = 2$ already $(2, 2)$-AsymRelCol is trivial. For $\Delta = 3$, it was shown in [5] that every cubic graph admits a $(1, 189)$-relaxed coloring, making $(3, 189)$-AsymRelCol trivial. In the proof the vertex set of the graph was partitioned into a triangle-free and a triangle-full part, then the parts were colored separately, finally the two colorings were assembled amid some technical difficulties. In our first main theorem we greatly improve on this result by using a different approach, which avoids the separation. Our method also implies a quasilinear time algorithm (as opposed to the $\Theta(n^7)$ algorithm implicitly contained in [5]). One still has to deal with the inconveniences of triangles, but the obtained component order is much smaller.

Theorem 1. *Any graph G with $\Delta(G) \leq 3$ is $(1, 22)$-relaxed colorable, i.e.*

$$f(3) \leq 22.$$

Moreover there is an $O(n \log^4 n)$ algorithm which finds such a 22-relaxed coloring.

A lower bound of 6 on $f(3)$ was established in [5].

In our next theorem we show that $(3, C)$-AsymRelCol exhibits the promised hardness jump.

Theorem 2. *For the integer $f(3)$ we have that*
(i) $(3, C)$-AsymRelCol is NP-complete for every $2 \leq C < f(3)$;
(ii) any graph G of maximum degree at most 3 is $(1, f(3))$-relaxed colorable.

In [5] it was shown that for any $\Delta \geq 4$ and positive C, (Δ, C)-AsymRelCol never becomes "trivial", i.e. for every finite C there is a "NO" instance, so $f(4) = \infty$. We show here however that the monotonicity of the hardness of $(4, C)$-AsymRelCol still exists for $C \geq 2$.

Theorem 3. *$(4, C)$-AsymRelCol is NP-complete for every $2 \leq C < f(4) = \infty$.*

Obviously, this implies that (Δ, C)-AsymRelCol is NP-complete for every $\Delta > 4$ and $2 \leq C < f(\Delta) = \infty$.

Remark. Let $f(\Delta, n)$ be the smallest integer f such that every n-vertex graph of maximum degree Δ is $(1, f)$-relaxed colorable. Then $f(\Delta) = \sup f(\Delta, n)$. While $f(3)$ is finite, our graph G_k on Figure 2 provides a simple example for $f(4)$ being

non-finite in a strong sense: in any asymmetric relaxed coloring of G_k there is a monochromatic component whose order is linear in the number of vertices. This is in sharp contrast with the examples of [1, 5] where the monochromatic component order is only logarithmic in the number of vertices. It would be interesting to determine the exact asymptotics of the function $f(4, n)$; we only know of the trivial upper bound $f(4, n) \leq \frac{3}{4}n$ and the lower bound $f(4, n) \geq \frac{2}{3}n$ because of G_k.

Combining arguments of [5] and the present paper we are able to prove a tight upper bound on the component order for graphs of maximum degree 3 in which every vertex is contained in a triangle.

Theorem 4. *Let G be a graph of maximum degree 3, in which every vertex is contained in a triangle. Then G has a $(1, 6)$-relaxed coloring.*

The proof of this theorem will appear in the full version of the paper [4]. An example in [5] shows that the component order 6 is best possible. We note that a 6-relaxed coloring of triangle-free graphs was already proved in [5].

The symmetric problem. Investigations about relaxed vertex colorings were originally initiated for the symmetric case by Alon, Ding, Oporowski and Vertigan [1]. They showed that any graph of maximum degree 4 has a two-coloring such that each monochromatic component is of order at most 57. This was improved by Haxell, Szabó and Tardos [10], who showed that a two-coloring is possible even with monochromatic component order of 6, and such a $(6, 6)$-relaxed coloring can be constructed in polynomial time (the algorithm of [1] is not obviously polynomial). In [10] it is also proved that the family of graphs of maximum degree 5 is $(17617, 17617)$-relaxed colorable. This coloring is using the Local Lemma and it is not known whether there is a constant C and a polynomial-time algorithm which constructs a (C, C)-relaxed coloring of graphs of maximum degree 5. Alon et al. [1] showed that a similar statement cannot be true for the family of graphs of maximum degree 6, as for every constant C there exists a 6-regular graph G_C such that in any two-coloring of $V(G_C)$ there is a monochromatic component of order larger than C.

For the problem (Δ, C)-SymRelCol we make progress in the direction of establishing a sudden jump in hardness. By taking a max-cut one can easily see that $(3, C)$-SymRelCol is trivial already for $C = 2$, so the first interesting maximum degree is $\Delta = 4$. From the result of [10] mentioned earlier it follows that $(4, 6)$-SymRelCol is trivial. Here we show that $(4, C)$-SymRelCol is NP-complete for $C = 2$ and $C = 3$. We do not know about the hardness of the problem for $C = 4$ and $C = 5$. Again, we do not know any *direct* reason for the monotonicity of the problem. I.e., at the moment it is in principle possible that $(4, 4)$-SymRelCol is in P while $(4, 5)$-SymRelCol is again NP-complete.

Theorem 5. *The problems $(4, 2)$-SymRelCol and $(4, 3)$-SymRelCol are NP-complete.*

The proof of the theorem appears in the full version of the paper [4].

Related work. Similar hardness jumps of the k-SAT problem with limited occurrences of each variable were shown by Tovey [19] for $k = 3$ and Kratochvíl, Savický and Tuza [14] for arbitrary k. Let k, s be positive integers. A Boolean formula in conjunctive normal form is called a (k, s)-formula if every clause contains *exactly* k distinct variables and every variable occurs in *at most* s clauses. Tovey showed that every $(3, 3)$-formula is satisfiable while the satisfiability problem restricted to $(3, 4)$-formulas is NP-complete. Kratochvíl, Savický and Tuza [14] generalized this by establishing the existence of a function $f(k)$, such that every $(k, f(k))$-formula is satisfiable while the satisfiability problem restricted to $(k, f(k) + 1)$-CNF formulas is NP-complete. By a standard application of the Local Lemma they obtained $f(k) \geq \left\lfloor \frac{2^k}{ek} \right\rfloor$. After some development [14, 16] the most recent upper estimate on $f(k)$ is only a log-factor away from the lower bound and is due to Hoory and Szeider [12]. Recently new bounds were also obtained on small values of the function $f(k)$ [13]. Observe that the monotonicity of the hardness of the satisfiability problem for (k, s)-formulas is given by definition.

Notation. The order of a graph G is defined to be the number of vertices of G. Similarly, the order of a connected component C of G is the number of vertices contained in C. A graph G is r-regular if all its vertices have degree r. A graph G is called k-edge-connected if there is no edge-cut (a subset of the edges of G that disconnects G) of size at most $k - 1$.

The subgraph of a graph G induced by a vertex set $U \subseteq V(G)$ is denoted throughout by $G[U]$. Connected components in an induced subgraph $G[U]$ are called U-components and neighbors of a vertex $v \in V(G)$ in the induced subgraph $G[U]$ are called U-neighbors.

2 Trivial $(3, C)$-AsymRelCol – Bounding $f(3)$

Proof (of Theorem 1.). In this section and the next one we simplify our notation by saying *C-relaxed coloring* instead of $(1, C)$-relaxed coloring.

All graphs we consider have maximum degree three. The main part of the proof is to establish the statement for 2-edge-connected 3-regular graphs. One can then easily extend this argument to arbitrary graphs of maximum degree 3. More details will be included in the full version of the paper [4].

Lemma 1. *Every 2-edge-connected, 3-regular graph has a vertex partition $I \cup X \cup B = V(G)$ such that*

(i) $I \cup X$ induces a graph where each I-vertex has degree 0 and each X-vertex has degree 1.

(ii) No triangle contains two vertices from X.

(iii) Every B-component is of order at most 6.

Observe that it is easy to argue that without loss of generality G is diamond-free, where a *diamond* is a graph consisting of two triangles sharing an edge. Hence in the proof we consider only graphs where no two triangles intersect.

First let us see how Lemma 1 implies Theorem 1 for 2-edge-connected 3-regular graphs. Let I, X, B be such as promised by Lemma 1. We do a postprocessing in two phases, during which we distribute the vertices of X between I and B. For each adjacent pair vw of vertices in X we put one of them to B and the other into I. When this happens we say that we *distributed* the X-edge vw. In the first phase some vertices contained in B will be moved to I, but once a vertex is in I, it stays there during the rest of the postprocessing.

For the first phase let us say that a vertex v is *ready for a change* if $v \in B$ and all its neighbors are in $B \cup X$. Once we find a vertex v ready for a change we move v to I, and distribute the X-edges it is adjacent to by moving all X-neighbors of v into B (and their X-neighbors into I). We iteratively make this change until we find no more vertex ready for a change, at which point the first phase ends. Property (ii) ensures that the rules of our change are well-defined. It is not possible that an X-neighbor of v is instructed to be placed in B, while it could also be the X-neighbor of another X-neighbor of v which would instruct it to be in I. After each change the property (i) stays true simply because some of the edges in X had their endpoints distributed one into I and one into B.

Crucially, at the end of the first phase every B-component is a path. As a result of one change no two B-components are joined, possibly a vertex u from X which changed its color to B is now stuck to an old B-component. In case this happens both of the other neighbors of u are in I (and stay there).

Let C be a B-component after the first phase. We claim that all vertices adjacent to C are in I except possibly two: one-one at each endpoint of C. By (iii) there is an at most 6-long path C' in C which used to be in a B-component before the first phase. So we can distinguish three cases in terms of how many X-neighbors can C have besides its I-neighbors.

Observations. After the first phase every B-component is one of the following:

(a) C is either a path of length at most 6 with one X-neighbor at each of its endpoints, or
(b) C is a path of length at most 7 with one X-neighbor at one of its endpoints, or
(c) C is a path of length 8 with no X-neighbors.

In the second phase we distribute the vertices that are still in X between I and B in such a way that the connected components in $G[B]$ don't grow too much. This is done by finding a *matching transversal* in an auxiliary graph H. The graph H is defined on the vertices of X, $V(H) = X$. There is an edge between two vertices u and v in H iff u and v are incident to the same component of $G[B]$.

Claim. $\Delta(H) \leq 2$.

Proof. Let us pick an arbitrary vertex y from $X = V(H)$. We aim to show that each of the two edges e_1, e_2 that are not incident to another X-vertex is "responsible" for at most one neighbor of y in H. That is, the component in $G[B]$ incident to y via such an e_1 or e_2 is incident to at most one other vertex from X.

Indeed, by the Observation above each B-component is a path, possibly adjacent to X-vertices through its endpoints, but not more than to two.

The following Lemma guarantees a transversal inducing a matching.

Lemma 2 ([10], Corollary 4.3). *Let G be a graph with $\Delta(G) \leq 2$ together with a vertex partition $\mathcal{P} = \{P_1, \ldots, P_m\}$ into 2-element subsets. Then there is a transversal T ($(T \cap P_i) \neq \emptyset$, for all $i \in \{1, \ldots, m\}$) with $\Delta(G[T]) \leq 1$.*

We remark that the proof of this Lemma in [10] involves a linear time algorithm which constructs the transversal.

We apply Lemma 2 and find such a transversal T of H on the partition defined by the edges in $G[X]$, $\mathcal{P} = E(G[X])$.

Now the second phase of our postprocessing consists of moving all vertices of T into B and moving $X \setminus T$ into I. Since $\Delta(H(T)) \leq 1$ we connect at most three connected components $Q_1$and Q_2 and Q_3 of $G[B]$ by moving an edge $\{u, v\}$ of H into B, with u incident to Q_1 and Q_2 and v incident to Q_2 and Q_3. Obviously, Q_1 and Q_3 are incident to at least one vertex of H (u and v respectively) and Q_2 is incident to at least two vertices from H (u and v) before moving the vertices of T. According to the Observation above the largest B-component created this way is of order at most $7 + 1 + 6 + 1 + 7 = 22$. Lemma 1(i) guarantees that I is independent so the defined coloring is 22-relaxed. □

It remains to show that the partition of $V(G)$, promised in Lemma 1 indeed exists and can be found in time $O(n \log^4 n)$. The complete proof of Lemma 1 is relegated to the full version of this paper [4].

Let us here only informally discuss the main ideas of the algorithm that partitions the vertex set of G and denote it by PA(G).

In a first step the algorithm PA(G) finds a perfect matching M in G. Thus $G - M$ consists of disjoint cycles only. Moreover M is chosen such that $G - M$ is triangle-free. Such a matching can be found in $O(n \log^4 n)$ time, see Biedl, Bose, Demaine and Lubiw [6]. (Note that the algorithm in [6] only yields some perfect matching. In order to obtain a perfect matching M such that $G - M$ is triangle-free we first contract all triangles in G yielding a new graph G'. Then we apply the algorithm to G' instead and get a perfect matching M' of G'. We observe that this perfect matching M' can easily be extended to a perfect matching M of G where each triangle of G contains exactly one edge of M. Thus $G - M$ is triangle-free.) This is in fact the bottleneck of our algorithm all other parts are done in linear time. The unique neighbor of a vertex v in M is called the *partner* of v.

Next, PA(G) colors iteratively all vertices of G, one cycle of $G - M$ after another, by traversing each cycle in a predefined direction. As a default PA(G) tries to color the vertices of a cycle with the colors I and B alternatingly. Its original goal is to create a proper two-coloring this way. Of course there are several reasons which will prevent PA(G) from doing so. One main obstacle is when the partner of the currently processed vertex v is already colored, and it is done so with the same color we just gave to v. If the conflict would be in color I

then the algorithm resolves this by changing both v and its partner to X. The algorithm generally decides not to care if the conflict is in B.

Of course there is a complication with this rule when the partner is within the same triangle as v, since Lemma 1 does not allow two X-vertices in the same triangle. This and other anomalies (like the coloring of the last vertex of a cycle when the first and next-to-last vertex have distinct colors) are handled in the full version of this paper [4] by a (hopefully) well-designed set of exceptions in place.

After having colored cycle C the algorithm immediately proceeds with the cycle containing the partner v of the last vertex colored in C unless v is already colored. Otherwise the algorithm looks for vertices in C with an uncolored partner by stepping backwards along the order in which the vertices of C have been colored and eventually starts to color such a partner. If none of the vertices of C have an uncolored partner the algorithm starts with a vertex whose partner is colored.

3 Hard $(3, C)$-AsymRelCol

Proof (of Theorem 2 and 3). For a C-relaxed coloring we denote the color class forming an independent set by I and the color class spanning components of order at most C by B.

Definition 1. *Let $C \geq 2$ and $\Delta \geq 1$ be integers. A graph G is called* (Δ, C)-forcing *with forced vertex $f \in V(G)$ if*
(i) $\Delta(G) \leq \Delta$ and f has degree at most $\Delta - 1$,
(ii) G is C-relaxed colorable, and
(iii) f is contained in I for every C-relaxed two-coloring of G.

Lemma 3. *For any non-negative integer Δ and integer $C \geq 2$ the decision problem (Δ, C)-AsymRelCol is NP-complete provided a (Δ, C)-forcing graph exists.*

The proof is detailed in the full version of the paper [4]. In the proof we establish a reduction from the 3-SAT Problem using appropriate gadgets built from (Δ, C)-forcing graphs.

3.1 $(3, C)$-Forcing Graphs

All graphs we consider in this subsection have maximum degree at most three. Let $\mathcal{G}_C$ denote the family of graphs of maximum degree at most three that are not C-relaxed two-colorable.

Lemma 4. *For all $C \geq 2$, if $\mathcal{G}_C \neq \emptyset$ then there is a $(3, C)$-forcing graph.*

Proof. Let us assume first that $C \geq 6$. By a lemma of [5] we can assume that any member of $\mathcal{G}_C$ contains a triangle.

Lemma 5 ([5]). *Any triangle-free graph of maximum degree at most 3 has a 6-relaxed coloring.*

Let us fix a graph $G \in \mathcal{G}_C$ which is minimal with respect to deletion of edges. Let T be a triangle in G (guaranteed by Lemma 5) with $V(T) = \{t_1, t_2, u\}$ and $e = \{u, v\}$ be the unique edge incident to u not contained in T. We split e into e_1, e_2 with $e_1 = \{u, f\}$ and $e_2 = \{f, v\}$ and denote this new graph by H. H is C-relaxed colorable since the minimality of G ensures that $G - e$ has a C-relaxed coloring while the non-C-relaxed-colorability of G ensures that the colors of u and v are the same on *any* C-relaxed coloring of $G - e$. So *any* C-relaxed coloring χ of $G - e$ can be extended to a C-relaxed coloring of H by coloring f to the opposite of the color of u and v. Moreover any such extension is unique. If $\chi(u) = \chi(v) = I$, then obviously $\chi(f) = B$. If $\chi(u) = \chi(v) = B = \chi(f)$ and χ is a C-relaxed coloring of H, then χ restricted to $V(G)$ is a C-relaxed coloring of G, a contradiction.

Thus in any C-relaxed coloring χ_H of H, $(\chi_H(u), \chi_H(f), \chi_H(v))$ is either (I, B, I) or (B, I, B).

We denote by v_1, v_2 the neighbors of t_1 and t_2, respectively, not contained in T (might be $t_1 = t_2$). See also Figure 1. Suppose the vertices (u, f, v) of H can be colored with (I, B, I). But then $\chi_H(t_1) = \chi_H(t_2) = B$.

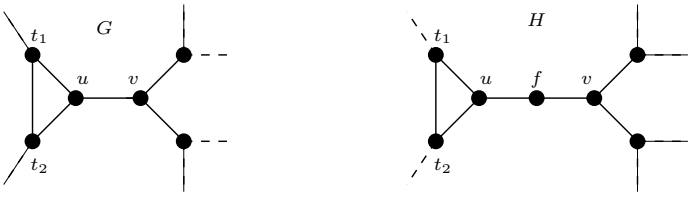


Fig. 1. Splitting $e = \{u, v\}$ into $e_1 = \{u, f\}$ and $e_2 = \{f, v\}$

[**Case** (i)**:**] If $\chi_H(v_1) = \chi_H(v_2) = I$ then we define a C-relaxed coloring χ_G for G as follows:
$\chi_G(x) = \chi_H(x)$ for all $x \in V(G) \setminus \{u\}$ and $\chi_G(u) = B$.
[**Case** (ii)**:**] Without loss of generality $\chi_H(v_1) = B$. We define a C-relaxed coloring χ_G for G as follows:
$\chi_G(x) = \chi_H(x)$ for all $x \in V(G) \setminus \{t_1, u\}$, $\chi_G(t_1) = I$, and $\chi_G(u) = B$. Indeed, the B-component containing t_2 did not increase, since $\chi_G(t_1) = \chi_G(v) = I$ and in H $\chi_H(t_1) = B$.

This contradicts the fact that G is not C-relaxed two-colorable. Thus in any C-relaxed coloring of H the vertices (u, f, v) are colored (B, I, B). The vertex f is contained in I and is of degree 2, hence H is a $(3, C)$-forcing graph with forced vertex f.

In the full version of this paper [4] we provide explicit constructions of $(3, C)$-forcing graphs with $2 \leq C \leq 5$. □

Note that $(3, C)$-AsymRelCol is obviously trivial for all C with $\mathcal{G}_C = \emptyset$, so Theorem 2 follows immediately from Lemma 4 and Lemma 3. □

3.2 $(4, C)$-Forcing Graphs

Lemma 6. *For all $\Delta \geq 4$ and all $C \geq 2$ there is a (Δ, C)-forcing graph.*

The graph $G_k - \{v_{1,1}, v_{1,2}\}$ is $(4, 2k-2)$-forcing. A proof can be found in the full version of this paper [4]. Combining Lemma 6 and Lemma 3 concludes the proof of Theorem 3. □

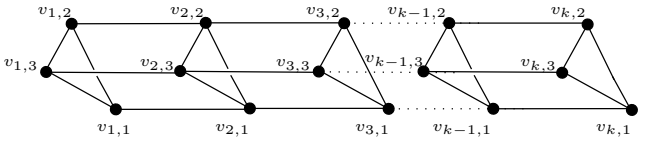


Fig. 2. G_k with one B-component of order $2k$

4 Summarizing Overview and Open Problems

It would be interesting to determine exactly the critical monochromatic component order $f(3)$ from where the problem $(3, C)$-AsymRelCol becomes trivial.

We conjecture that there is a sudden jump in the hardness of the problem $(4, C)$-SymRelCol. Such a result would particularly be interesting, since here the determination of the critical component order is even more within reach (between 4 and 6.) As a first step one could try to prove the monotonicity of the problem.

The similar problem is wide open for graphs with maximum degree 5: Does SymRelCol exhibit a monotone behavior for $C \geq 2$? Is there a "jump in hardness"? Is there a constant C and a polynomial-time algorithm which finds a (C, C)-coloring of graphs of maximum degree 5? We only know the existence of such colorings.

For colorings with more than two colors we know much less. Even the graph theoretic questions about interesting maximum degrees are open. We list here three of the most important questions: Is there a constant C such that every graph with maximum degree 9 can be three-colored such that every monochromatic component is of order at most C? The answer is "yes" for graphs with maximum degree 8 and "no" for graphs of maximum degree 10 (see [10]). Is there a constant C such that every graph of maximum degree 5 can be red/blue/green-colored such that the set of red vertices and the set of blue vertices are both independent while every green monochromatic component is of order at most C? The answer is "yes" for graphs with maximum degree 4 and "no" for graphs of maximum degree 6 (see [5]). Determine asymptotically the largest Δ_k for which there exists a constant C_k such that every graph of maximum degree Δ_k can be k-colored such that every monochromatic component is of order at most C_k. The current bounds are $3 < \Delta_k/k \leq 4$ (see [10]).

References

1. N. Alon, G. Ding, B. Oporowski, D. Vertigan, Partitioning into graphs with only small components, *Journal of Combinatorial Theory, (Series B)*, **87**(2003) 231-243.
2. J. Akiyama, V. Chvátal, A short proof of the linear arboricity for cubic graphs, *Bull. Liber. Arts Sci. NMS* 2 (1981).
3. J. Akiyama, G. Exoo, F. Harary, Coverings and packings in graphs II. Cyclic and acyclic invariants. *Math. Slovaca* **30** (1980), 405-417.
4. R. Berke, T. Szabó, Deciding Relaxed Two-Colorability — A Hardness Jump, full version, available at *http://www.inf.ethz.ch/personal/berker/publications/relcomfull.pdf*.
5. R. Berke, T. Szabó, Relaxed Coloring of Cubic Graphs, *submitted*; extended abstract in *Proceedings EuroComb* (2005), 341-344.
6. T.C. Biedl, P. Bose, E.D. Demaine, A. Lubiw, Efficient Algorithms for Petersen's Matching Theorem, *Journal of Algorithms*, **38**, (2001), 110-134.
7. K. Edwards, G. Farr, Fragmentability of graphs, *J. Comb. Th. Series B*, **82**, no. 1, (2001), 30-37.
8. F. Havet, J.-S. Sereni, Improper choosability of graphs and maximum average degree, *J. Graph Theory*, (2005), to appear.
9. F. Havet, R. Kang, J.-S. Sereni, Improper colouring of unit disk graphs. *Proceedings ICGT*, **22**, (2005), 123-128.
10. P.Haxell, T. Szabó, G. Tardos, Bounded size components — partitions and transversals, *Journal of Combinatorial Theory (Series B)*, **88** no.2. (2003) 281-297.
11. P. Haxell, O. Pikhurko, A. Thomason, Maximum Acyclic and Fragmented Sets in Regular Graphs, *submitted*.
12. S. Hoory, S. Szeider, Families of unsatisfiable k-CNF formulas with few occurrences per variable, *arXiv.org e-Print archive*, math.CO/0411167, (2004).
13. S. Hoory, S. Szeider.Computing Unsatisfiable k-SAT Instances with Few Occurrences per Variable, *Theoretical Computer Science*, **337**, no. 1-3, (2005) 347-359.
14. J. Kratochvíl, P. Savický, Zs. Tuza, One more occurrence of variables makes satisfiability jump from trivial to NP-complete, *SIAM J. Comput.* **22** (1) (1993) 203-210.
15. N. Linial, M. Saks, Low diameter graph decompositions, *Combinatorica*, **13** no.4. (1993) 441-454.
16. P. Savický, J. Sgall, DNF tautologies with a limited number of of occurrences of every variable, *Theoret. Comput. Sci.*, **238** (1-2) (2000) 495-498.
17. R. Škrekovski, List improper colorings of planar graphs, *Comb. Prob. Comp.*, **8** (1999), 293-299.
18. C. Thomassen, Two-colouring the edges of a cubic graph such that each monochromatic component is a path of length at most 5, *J. Comb. Th. Series B* **75**, (1999), 100-109.
19. C. Tovey, A simplified NP-complete satisfiability problem, *Discrete Appl. Math.*, **8** no.1 (1984) 85-89.

Negative Examples for Sequential Importance Sampling of Binary Contingency Tables

Ivona Bezáková[1], Alistair Sinclair[2,⋆], Daniel Štefankovič[3], and Eric Vigoda[4,⋆⋆]

[1] Department of Computer Science, University of Chicago, Chicago, IL 60637
ivona@cs.uchicago.edu
[2] Computer Science Division, University of California, Berkeley, CA 94720
sinclair@cs.berkeley.edu
[3] Department of Computer Science, University of Rochester, Rochester, NY 14627
stefanko@cs.rochester.edu
[4] College of Computing, Georgia Institute of Technology, Atlanta, GA 30332
vigoda@cc.gatech.edu

Abstract. The sequential importance sampling (SIS) algorithm has gained considerable popularity for its empirical success. One of its noted applications is to the binary contingency tables problem, an important problem in statistics, where the goal is to estimate the number of 0/1 matrices with prescribed row and column sums. We give a family of examples in which the SIS procedure, if run for any subexponential number of trials, will underestimate the number of tables by an exponential factor. This result holds for any of the usual design choices in the SIS algorithm, namely the ordering of the columns and rows. These are apparently the first theoretical results on the efficiency of the SIS algorithm for binary contingency tables. Finally, we present experimental evidence that the SIS algorithm is efficient for row and column sums that are regular. Our work is a first step in determining rigorously the class of inputs for which SIS is effective.

1 Introduction

Sequential importance sampling is a widely-used approach for randomly sampling from complex distributions. It has been applied in a variety of fields, such as protein folding [8], population genetics [5], and signal processing [7]. Binary contingency tables is an application where the virtues of sequential importance sampling have been especially highlighted; see Chen et al. [4]. This is the subject of this note. Given a set of non-negative row sums $r = (r_1, \ldots, r_m)$ and column sums $c = (c_1, \ldots, c_n)$, let $\Omega = \Omega_{r,c}$ denote the set of $m \times n$ 0/1 tables with row sums r and column sums c. Our focus is on algorithms for sampling (almost) uniformly at random from Ω, or estimating $|\Omega|$.

⋆ Supported by NSF grant CCR-0121555.
⋆⋆ Supported by NSF grant CCR-0455666.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 136–147, 2006.

Sequential importance sampling (SIS) has several purported advantages over the more classical Markov chain Monte Carlo (MCMC) method, such as:

Speed: Chen et al. [4] claim that SIS is faster than MCMC algorithms. However, we present a simple example where SIS requires exponential (in n, m) time. In contrast, a MCMC algorithm was presented in [6, 1] which is guaranteed to require at most time polynomial in n, m for every input.

Convergence Diagnostic: One of the difficulties in MCMC algorithms is determining when the Markov chain of interest has reached the stationary distribution, in the absence of analytical bounds on the mixing time. SIS seemingly avoids such complications since its output is guaranteed to be an unbiased estimator of $|\Omega|$. Unfortunately, it is unclear how many estimates from SIS are needed before we have a guaranteed close approximation of $|\Omega|$. In our example for which SIS requires exponential time, the estimator appears to converge, but it converges to a quantity that is off from $|\Omega|$ by an exponential factor.

Before formally stating our results, we detail the sequential importance sampling approach for contingency tables, following [4]. The general importance sampling paradigm involves sampling from an 'easy' distribution μ over Ω that is, ideally, close to the uniform distribution. At every round, the algorithm outputs a table T along with $\mu(T)$. Since for any μ whose support is Ω we have

$$E[1/\mu(T)] = |\Omega|,$$

we take many trials of the algorithm and output the average of $1/\mu(T)$ as our estimate of $|\Omega|$. More precisely, let $T_1, \ldots, T_t$ denote the outputs from t trials of the SIS algorithm. Our final estimate is

$$X_t = \frac{1}{t} \sum_i \frac{1}{\mu(T_i)}. \tag{1}$$

One typically uses a heuristic to determine how many trials t are needed until the estimator has converged to the desired quantity.

The sequential importance sampling algorithm of Chen et al. [4] constructs the table T in a column-by-column manner. It is not clear how to order the columns optimally, but this will not concern us as our negative results will hold for any ordering of the columns. Suppose the procedure is assigning column j. Let $r'_1, \ldots, r'_m$ denote the residual row sums after taking into account the assignments in the first $j-1$ columns.

The procedure of Chen et al. chooses column j from the correct probability distribution conditional on c_j, $r'_1, \ldots, r'_m$ and the number of columns remaining (but ignoring the column sums $c_{j+1}, \ldots, c_n$). This distribution is easy to describe in closed form. We assign column j the vector $(t_1, \ldots, t_m) \in \{0,1\}^m$, where $\sum_i t_i = c_j$, with probability proportional to

$$\prod_i \left(\frac{r'_i}{n' - r'_i} \right)^{t_i}, \tag{2}$$

where $n' = n - j + 1$. If no valid assignment is possible for the j-th column, then the procedure restarts from the beginning with the first column (and sets $\frac{1}{\mu(T_i)} = 0$ in (1) for this trial).[1] Sampling from the above distribution over assignments for column j can be done efficiently by dynamic programming.

We now state our negative result. This is a simple family of examples where the SIS algorithm will grossly underestimate $|\Omega|$ unless the number of trials t is exponentially large. Our examples will have the form $(1, 1, \ldots, 1, d_r)$ for row sums and $(1, 1, \ldots, 1, d_c)$ for column sums, where the number of rows is $m + 1$, the number of columns is $n + 1$, and we require that $m + d_r = n + d_c$.

Theorem 1. *Let $\beta > 0, \gamma \in (0, 1)$ be constants satisfying $\beta \neq \gamma$ and consider the input instances $r = (1, 1, \ldots, 1, \lfloor \beta m \rfloor)$, $c = (1, 1, \ldots, 1, \lfloor \gamma m \rfloor)$ with $m + 1$ rows. Fix any order of columns (or rows, if sequential importance sampling constructs tables row-by-row) and let X_t be the random variable representing the estimate of the SIS procedure after t trials of the algorithm. There exist constants $s_1 \in (0, 1)$ and $s_2 > 1$ such that for every sufficiently large m and for any $t \leq s_2^m$,*

$$\mathbf{Pr}\left(X_t \geq \frac{|\Omega_{r,c}|}{s_2^m}\right) \leq 3s_1^m.$$

In contrast, note that there are MCMC algorithms which provably run in time polynomial in n and m for *any* row/column sums. In particular, the algorithm of Jerrum, Sinclair, and Vigoda [6] for the permanent of a non-negative matrix yields as a corollary a polynomial time algorithm for any row/column sums. The fastest algorithm for the permanent of a $n \times n$ matrix requires $O(n^7 \log^4 n)$ time [3], which implies a running time of $O((nm)^7 \log^4 n)$ time for binary contingency tables. More recently, Bezáková, Bhatnagar and Vigoda [1] have presented a related MCMC algorithm that works directly with binary contingency tables and has an improved polynomial running time. Their algorithm runs in time $O((nm)^2 R^3 s_{\max} \log^5(n + m))$ where $R = \sum_i r_i$ is the sum of the row sums and $s_{\max}$ is the maximum row and column sum. We note that, in addition to being formally asymptotically faster than any exponential time algorithm, a polynomial time algorithm has additional theoretical significance in that it (and its analysis) implies non-trivial insight into the the structure of the problem.

Some caveats are in order here. Firstly, the above results imply only that MCMC outperforms SIS asymptotically *in the worst case*; for many inputs, SIS may well be much more efficient. Secondly, the rigorous worst case upper bounds on the running time of the above MCMC algorithms are still far from practical. Chen et al. [4] showed several examples where SIS outperforms MCMC methods. We present a more systematic experimental study of the performance of SIS, focusing on examples where all the row and column sums are identical as well as on the "bad" examples from Theorem 1. Our experiments suggest that SIS

[1] Chen et al. devised a more subtle procedure which guarantees that there will always be a suitable assignment of every column. We do not describe this interesting modification of the procedure, as the two procedures are equivalent for the input instances which we discuss in this paper.

is extremely fast on the balanced examples, while its performance on the bad examples confirms our theoretical analysis.

We begin in Section 2 by presenting a few basic lemmas that are used in the analysis of our negative example. In Section 3 we present our main example where SIS is off by an exponential factor, thus proving Theorem 1. Finally, in Section 4 we present some experimental results for SIS that support our theoretical analysis.

2 Preliminaries

We will continue to let $\mu(T)$ denote the probability that a table $T \in \Omega_{r,c}$ is generated by sequential importance sampling algorithm. We let $\pi(T)$ denote the uniform distribution over Ω, which is the desired distribution.

Before beginning our main proofs we present two straightforward technical lemmas which are used at the end of the proof of the main theorem. The first lemma claims that if a large set of binary contingency tables gets a very small probability under SIS, then SIS is likely to output an estimate which is not much bigger than the size of the complement of this set, and hence very small. Let $\overline{\mathcal{S}} = \Omega_{r,c} \setminus \mathcal{S}$.

Lemma 1. *Let $p \leq 1/2$ and let $\mathcal{S} \subseteq \Omega_{r,c}$ be such that $\mu(\mathcal{S}) \leq p$. Then for any $a > 1$, and any t, we have*

$$\mathbf{Pr}\left(X_t \leq a\pi(\overline{\mathcal{S}})|\Omega|\right) \geq 1 - 2pt - 1/a.$$

Proof. The probability that all t SIS trials are not in $\mathcal{S}$ is at least

$$(1-p)^t > e^{-2pt} \geq 1 - 2pt,$$

where the first inequality follows from $\ln(1-x) > -2x$, valid for $0 < x \leq 1/2$, and the second inequality is the standard $e^{-x} \geq 1 - x$ for $x \geq 0$.

Let $T_1, \ldots, T_t$ be the t tables constructed by SIS. Then, with probability $> 1 - 2pt$, we have $T_i \in \overline{\mathcal{S}}$ for all i. Notice that for a table T constructed by SIS from $\overline{\mathcal{S}}$, we have

$$\mathbf{E}\left(\frac{1}{\mu(T)} \mid T \in \overline{\mathcal{S}}\right) = |\overline{\mathcal{S}}|.$$

Let $\mathcal{F}$ denote the event that $T_i \in \overline{\mathcal{S}}$ for all i, $1 \leq i \leq t$; hence,

$$\mathbf{E}\left(X_t \mid \mathcal{F}\right) = |\overline{\mathcal{S}}|.$$

We can use Markov's inequality to estimate the probability that SIS returns an answer which is more than a factor of a worse than the expected value, conditioned on the fact that no SIS trial is from $\mathcal{S}$:

$$\mathbf{Pr}\left(X > a|\overline{\mathcal{S}}| \mid \mathcal{F}\right) \leq \frac{1}{a}.$$

Finally, removing the conditioning we get:

$$\begin{aligned}\Pr(X \le a|\overline{\mathcal{S}}|) &\ge \mathbf{Pr}\left(X \le a|\overline{\mathcal{S}}| \mid \mathcal{F}\right)\mathbf{Pr}\left(\mathcal{F}\right)\\ &\ge \left(1-\frac{1}{a}\right)(1-2pt)\\ &\ge 1-2pt-\frac{1}{a}.\end{aligned}$$

The second technical lemma shows that if in a row with large sum (linear in m) there exists a large number of columns (again linear in m) for which the SIS probability of placing a 1 at the corresponding position differs significantly from the correct probability, then in any subexponential number of trials the SIS estimator will very likely exponentially underestimate the correct answer.

Lemma 2. *Let $\alpha < \beta$ be positive constants. Consider a class of instances of the binary contingency tables problem, parameterized by m, with $m+1$ row sums, the last of which is $\lfloor \beta m \rfloor$. Let $\mathcal{A}_i$ denote the set of all valid assignments of $0/1$ to columns $1, \ldots, i$. Suppose that there exist constants $f < g$ and a set I of cardinality $\lfloor \alpha m \rfloor$ such that one of the following statements is true:*

(i) for every $i \in I$ and any $A \in \mathcal{A}_{i-1}$,

$$\pi(A_{m+1,i} = 1 \mid A) \le f < g \le \mu(A_{m+1,i} = 1 \mid A),$$

(ii) for every $i \in I$ and any $A \in \mathcal{A}_{i-1}$,

$$\mu(A_{m+1,i} = 1 \mid A) \le f < g \le \pi(A_{m+1,i} = 1 \mid A).$$

Then there exists a constant $b_1 \in (0,1)$ such that for any constant $1 < b_2 < 1/b_1$ and any sufficiently large m, for any $t \le b_2^m$,

$$\mathbf{Pr}\left(X_t \ge \frac{|\Omega_{r,c}|}{b_2^m}\right) \le 3(b_1 b_2)^m.$$

In words, in b_2^m trials of sequential importance sampling, with probability at least $1 - 3(b_1 b_2)^m$ the output is a number which is at most a b_2^{-m} fraction of the total number of corresponding binary contingency tables.

Proof. We will analyze case (i); the other case follows from analogous arguments. Consider indicator random variables U_i representing the event that the uniform distribution places 1 in the last row of the i-th column. Similarly, let V_i be the corresponding indicator variable for the SIS. The random variable U_i is dependent on U_j for $j < i$ and V_i is dependent on V_j for $j < i$. However, each U_i is stochastically dominated by U_i' which has value 1 with probability f, and each V_i stochastically dominates the random variable V_i' which takes value 1 with probability g. Moreover, the U_i' and V_i' are respectively i.i.d.

Now we may use the Chernoff bound. Let $k = \lfloor \alpha m \rfloor$. Then

$$\mathbf{Pr}\left(\sum_{i \in I} U_i' - kf \ge \frac{g-f}{2}k\right) < e^{-(g-f)^2 k/8}$$

and

$$\mathbf{Pr}\left(kg - \sum_{i \in I} V_i' \geq \frac{g-f}{2}k\right) < e^{-(g-f)^2 k/8}.$$

Let S be the set of all tables which have less than $kf+(g-f)k/2 = kg-(g-f)k/2$ ones in the last row of the columns in I. Let $b_1 := e^{-(g-f)^2\alpha/16} \in (0,1)$. Then $e^{-(g-f)^2 k/8} \leq b_1^m$ for $m \geq 1/\alpha$. Thus, by the first inequality, under uniform distribution over all binary contingency tables the probability of the set S is at least $1 - b_1^m$. However, by the second inequality, SIS constructs a table from the set S with probability at most b_1^m.

We are ready to use Lemma 1 with $\mathcal{S} = S$ and $p = b_1^m$. Since under uniform distribution the probability of S is at least $1 - b_1^m$, we have that $|\mathcal{S}| \geq (1 - b_1^m)|\Omega_{r,c}|$. Let $b_2 \in (1, 1/b_1)$ be any constant and consider $t \leq b_2^m$ SIS trials. Let $a = (b_1 b_2)^{-m}$. Then, by Lemma 1, with probability at least $1 - 2pt - 1/a \geq 1 - 3(b_1 b_2)^m$ the SIS procedure outputs a value which is at most an $ab_1^m = b_2^{-m}$ fraction of $|\Omega_{r,c}|$.

3 Proof of Main Theorem

In this section we prove Theorem 1. Before we analyze the input instances from Theorem 1, we first consider the following simpler class of inputs.

3.1 Row Sums $(1, 1, \ldots, 1, d)$ and Column Sums $(1, 1, \ldots, 1)$

The row sums are $(1, \ldots, 1, d)$ and the number of rows is $m + 1$. The column sums are $(1, \ldots, 1)$ and the number of columns is $n = m + d$. We assume that sequential importance sampling constructs the tables column-by-column. Note that if SIS constructed the tables row-by-row, starting with the row with sum d, then it would in fact output the correct number of tables exactly. However, in the next subsection we will use this simplified case as a tool in our analysis of the input instances $(1, \ldots, 1, d_r)$, $(1, \ldots, 1, d_c)$, for which SIS must necessarily fail regardless of whether it works row-by-row or column-by-column, and regardless of the order it chooses.

Lemma 3. *Let $\beta > 0$, and consider an input of the form $(1, \ldots, 1, \lfloor \beta m \rfloor)$, $(1, \ldots, 1)$ with $m + 1$ rows. Then there exist constants $s_1 \in (0,1)$ and $s_2 > 1$, such that for any sufficiently large m, with probability at least $1 - 3s_1^m$, column-wise sequential importance sampling with s_2^m trials outputs an estimate which is at most a s_2^{-m} fraction of the total number of corresponding binary contingency tables. Formally, for any $t \leq s_2^m$,*

$$\mathbf{Pr}\left(X_t \geq \frac{|\Omega_{r,c}|}{s_2^m}\right) \leq 3s_1^m.$$

The idea for the proof of the lemma is straightforward. By the symmetry of the column sums, for large m and d and $\alpha \in (0,1)$ a uniform random table will have

about αd ones in the first αn cells of the last row, with high probability. We will show that for some $\alpha \in (0,1)$ and $d = \beta m$, sequential importance sampling is very unlikely to put this many ones in the first αn columns of the last row. Therefore, since with high probability sequential importance sampling will not construct any table from a set that is a large fraction of all legal tables, it will likely drastically underestimate the number of tables.

Before we prove the lemma, let us first compare the column distributions arising from the uniform distribution over all binary contingency tables with the SIS distributions. We refer to the column distributions induced by the uniform distribution over all tables as the *true* distributions. The true probability of 1 in the first column and last row can be computed as the number of tables with 1 at this position divided by the total number of tables. For this particular sequence, the total number of tables is $Z(m,d) = \binom{n}{d} m! = \binom{m+d}{d} m!$, since a table is uniquely specified by the positions of ones in the last row and the permutation matrix in the remaining rows and corresponding columns. Therefore,

$$\pi(A_{m+1,1} = 1) = \frac{Z(m,d-1)}{Z(m,d)} = \frac{\binom{m+d-1}{d-1} m!}{\binom{m+d}{d} m!} = \frac{d}{m+d}.$$

On the other hand, by the definition of sequential importance sampling, $\Pr(A_{i,1} = 1) \propto r_i/(n - r_i)$, where r_i is the row sum in the i-th row. Therefore,

$$\mu(A_{m+1,1} = 1) = \frac{\frac{d}{n-d}}{\frac{d}{n-d} + m\frac{1}{n-1}} = \frac{d(m+d-1)}{d(m+d-1) + m^2}.$$

Observe that if $d \approx \beta m$ for some constant $\beta > 0$, then for sufficiently large m we have

$$\mu(A_{m+1,1} = 1) > \pi(A_{m+1,1} = 1).$$

As we will see, this will be true for a linear number of columns, which turns out to be enough to prove that in polynomial time sequential importance sampling exponentially underestimates the total number of binary contingency tables with high probability.

Proof (Proof of Lemma 3). We will find a constant α such that for every column $i < \alpha m$ we will be able to derive an upper bound on the true probability and a lower bound on the SIS probability of 1 appearing at the $(m+1, i)$ position.

For a partially filled table with columns $1, \ldots, i-1$ assigned, let d_i be the remaining sum in the last row and let m_i be the number of other rows with remaining row sum 1. Then the true probability of 1 in the i-th column and last row can be bounded as

$$\pi(A_{m+1,i} = 1 \mid A_{(m+1)\times(i-1)}) = \frac{d_i}{m_i + d_i} \leq \frac{d}{m+d-i} =: f(d,m,i),$$

while the probability under SIS can be bounded as

$$\mu(A_{m+1,i} = 1 \mid A_{(m+1)\times(i-1)}) = \frac{d_i(m_i + d_i - 1)}{d_i(m_i + d_i - 1) + m_i^2} \geq \frac{(d-i)(m+d-i-1)}{d(m+d-1)+m^2} =: g(d,m,i).$$

Observe that for fixed m, d, the function f is increasing and the function g is decreasing in i, for $i < d$.

Recall that we are considering a family of input instances parameterized by m with $d = \lceil \beta m \rceil$, for a fixed $\beta > 0$. We will consider $i < \alpha m$ for some $\alpha \in (0, \beta)$. Let

$$f^\infty(\alpha,\beta) := \lim_{m\to\infty} f(d,m,\alpha m) = \frac{\beta}{1+\beta-\alpha}; \tag{3}$$

$$g^\infty(\alpha,\beta) := \lim_{m\to\infty} g(d,m,\alpha m) = \frac{(\beta-\alpha)(1+\beta-\alpha)}{\beta(1+\beta)+1}; \tag{4}$$

$$\triangle_\beta := g^\infty(0,\beta) - f^\infty(0,\beta) = \frac{\beta^2}{(1+\beta)(\beta(1+\beta)+1)} > 0, \tag{5}$$

and recall that for fixed β, f^∞ is increasing in α and g^∞ is decreasing in α, for $\alpha < \beta$. Let $\alpha < \beta$ be such that $g^\infty(\alpha,\beta) - f^\infty(\alpha,\beta) = \triangle_\beta/2$. Such an α exists by continuity and the fact that $g^\infty(\beta,\beta) < f^\infty(\beta,\beta)$.

By the above, for any $\epsilon > 0$ and sufficiently large m, and for any $i < \alpha m$, the true probability is upper-bounded by $f^\infty(\alpha,\beta) + \epsilon$ and the SIS probability is lower-bounded by $g^\infty(\alpha,\beta) - \epsilon$. For our purposes it is enough to fix $\epsilon = \triangle_\beta/8$. Now we can use Lemma 2 with α and β defined as above, $f = f^\infty(\alpha,\beta) + \epsilon$ and $g = g^\infty(\alpha,\beta) - \epsilon$ (notice that all these constants depend only on β), and $I = \{1, \ldots, \lfloor \alpha m \rfloor\}$. This finishes the proof of the lemma with $s_1 = b_1 b_2$ and $s_2 = b_2$.

Note 1. Notice that every contingency table with row sums $(1, 1, \ldots, 1, d)$ and column sums $(1, 1, \ldots, 1)$ is binary. Thus, this instance proves that the column-based SIS procedure for general (non-binary) contingency tables has the same flaw as the binary SIS procedure. We expect that the negative example used for Theorem 1 also extends to general (i. e., non-binary) contingency tables, but the analysis becomes more cumbersome.

3.2 Proof of Theorem 1

Recall that we are working with row sums $(1, 1, \ldots, 1, d_r)$, where the number of rows is $m + 1$, and column sums $(1, 1, \ldots, 1, d_c)$, where the number of columns is $n + 1 = m + 1 + d_r - d_c$. We will eventually fix $d_r = \lfloor \beta m \rfloor$ and $d_c = \lfloor \gamma m \rfloor$, but to simplify our expressions we work with d_r and d_c for now.

The theorem claims that the SIS procedure fails for an arbitrary order of columns with high probability. We first analyze the case when the SIS procedure starts with columns of sum 1; we shall address the issue of arbitrary column

order later. As before, under the assumption that the first column has sum 1, we compute the probabilities of 1 being in the last row for uniform random tables and for SIS respectively. For the true probability, the total number of tables can be computed as $\binom{m}{d_c}\binom{n}{d_r}(m-d_c)! + \binom{m}{d_c-1}\binom{n}{d_r-1}(m-d_c+1)!$, since a table is uniquely determined by the positions of ones in the d_c column and d_r row and a permutation matrix on the remaining rows and columns. Thus we have

$$\begin{aligned}\pi(A_{(m+1),1}) &= \frac{\binom{m}{d_c}\binom{n-1}{d_r-1}(m-d_c)! + \binom{m}{d_c-1}\binom{n-1}{d_r-2}(m-d_c+1)!}{\binom{m}{d_c}\binom{n}{d_r}(m-d_c)! + \binom{m}{d_c-1}\binom{n}{d_r-1}(m-d_c+1)!} \\ &= \frac{d_r(n-d_r+1)+d_c d_r(d_r-1)}{n(n-d_r+1)+nd_c d_r} =: f_2(m,d_r,d_c);\end{aligned}$$

$$\mu(A_{(m+1),1}) = \frac{\frac{d_r}{n-d_r}}{\frac{d_r}{n-d_r} + m\frac{1}{n-1}} = \frac{d_r(n-1)}{d_r(n-1)+m(n-d_r)} =: g_2(m,d_r,d_c).$$

Let $d_r = \lfloor \beta m \rfloor$ and $d_c = \lfloor \gamma m \rfloor$ for some constants $\beta > 0, \gamma \in (0,1)$ (notice that this choice guarantees that $n \geq d_r$ and $m \geq d_c$, as required). Then, as m tends to infinity, f_2 approaches

$$f_2^\infty(\beta,\gamma) := \frac{\beta}{1+\beta-\gamma},$$

and g_2 approaches

$$g_2^\infty(\beta,\gamma) := \frac{\beta(1+\beta-\gamma)}{\beta(1+\beta-\gamma)+1-\gamma}.$$

Notice that $f_2^\infty(\beta,\gamma) = g_2^\infty(\beta,\gamma)$ if and only if $\beta = \gamma$. Suppose that $f_2^\infty(\beta,\gamma) < g_2^\infty(\beta,\gamma)$ (the opposite case follows analogous arguments and uses the second part of Lemma 2). As in the proof of Lemma 3, we can define α such that if the importance sampling does not choose the column with sum d_c in its first αm choices, then in any subexponential number of trials it will exponentially underestimate the total number of tables with high probability. Formally, we derive an upper bound on the true probability of 1 being in the last row of the i-th column, and a lower bound on the SIS probability of the same event (both conditioned on the fact that the d_c column is not among the first i columns assigned). Let $d_r^{(i)}$ be the current residual sum in the last row, m_i be the number of rows with sum 1, and n_i the remaining number of columns with sum 1. Notice that $n_i = n-i+1$, $m \geq m_i \geq m-i+1$, and $d_r \geq d_r^{(i)} \geq d_r - i + 1$. Then

$$\begin{aligned}\pi(A_{(m+1),i} \mid A_{(m+1)\times(i-1)}) &= \frac{d_r^{(i)}(n_i - d_r^{(i)}+1) + d_c d_r^{(i)}(d_r^{(i)}-1)}{n_i(n_i-d_r^{(i)}+1)+n_i d_c d_r^{(i)}} \\ &\leq \frac{d_r(n-d_r+1)+d_c d_r^2}{(n-i)(n-i-d_r)+(n-i)d_c(d_r-i)} \\ &=: f_3(m,d_r,d_c,i);\end{aligned}$$

$$\mu(A_{(m+1),i} \mid A_{(m+1)\times(i-1)}) = \frac{d_r^{(i)}(n_i - 1)}{d_r^{(i)}(n_i - 1) + m_i(n_i - d_r^{(i)})} \geq \frac{(d_r - i)(n - i)}{d_r n + m(n - d_r)} =: g_3(m, d_r, d_c, i).$$

As before, notice that if we fix $m, d_r, d_c > 0$ satisfying $d_c < m$ and $d_r < n$, then f_3 is an increasing function and g_3 is a decreasing function in i, for $i < \min\{n - d_r, d_r\}$. Recall that $n - d_r = m - d_c$. Suppose that $i \leq \alpha m < \min\{m - d_c, d_r\}$ for some α which we specify shortly. Thus, the upper bound on f_3 in this range of i is $f_3(m, d_r, d_c, \alpha m)$ and the lower bound on g_3 is $g_3(m, d_r, d_c, \alpha m)$. If $d_r = \lfloor \beta m \rfloor$ and $d_c = \lfloor \gamma m \rfloor$, then the upper bound on f_3 converges to

$$f_3^\infty(\alpha, \beta, \gamma) := \lim_{m\to\infty} f_3(m, d_r, d_c, \alpha m) = \frac{\beta^2}{(1 + \beta - \gamma - \alpha)(\beta - \alpha)}$$

and the lower bound on g_3 converges to

$$g_3^\infty(\alpha, \beta, \gamma) := \lim_{m\to\infty} g_3(m, d_r, d_c, \alpha m) = \frac{(\beta - \alpha)(1 + \beta - \gamma - \alpha)}{\beta(1 + \beta - \gamma) + 1 - \gamma}$$

Let

$$\triangle_{\beta,\gamma} := g_3^\infty(0, \beta, \gamma) - f_3^\infty(0, \beta, \gamma) = g_2^\infty(\beta, \gamma) - f_2^\infty(\beta, \gamma) > 0.$$

We set α to satisfy $g_3^\infty(\alpha, \beta, \gamma) - f_3^\infty(\alpha, \beta, \gamma) \geq \triangle_{\beta,\gamma}/2$ and $\alpha < \min\{1 - \gamma, \beta\}$. Now we can conclude this part of the proof identically to the last paragraph of the proof of Lemma 3.

It remains to deal with the case when sequential importance sampling picks the d_c column within the first $\lfloor \alpha m \rfloor$ columns. Suppose d_c appears as the k-th column. In this case we focus on the subtable consisting of the last $n + 1 - k$ columns with sum 1, m' rows with sum 1, and one row with sum d', an instance of the form $(1, 1, \ldots, 1, d'), (1, \ldots, 1)$. We will use arguments similar to the proof of Lemma 3.

First we express d' as a function of m'. We have the bounds $(1-\alpha)m \leq m' \leq m$ and $d - \alpha m \leq d' \leq d$ where $d = \lfloor \beta m \rfloor \geq \beta m - 1$. Let $d' = \beta' m'$. Thus, $\beta - \alpha - 1/m \leq \beta' \leq \beta/(1 - \alpha)$.

Now we find α' such that for any $i \leq \alpha' m'$ we will be able to derive an upper bound on the true probability and a lower bound on the SIS probability of 1 appearing at position $(m' + 1, i)$ of the $(n + 1 - k) \times m'$ subtable, no matter how the first k columns were assigned. In order to do this, we might need to decrease α – recall that we are free to do so as long as α is a constant independent of m. By the derivation in the proof of Lemma 3 (see expressions (3) and (4)), as m' (and thus also m) tends to infinity, the upper bound on the true probability approaches

$$f^\infty(\alpha', \beta') = \lim_{m\to\infty} \frac{\beta'}{1 + \beta' - \alpha'} \leq \lim_{m\to\infty} \frac{\frac{\beta}{1-\alpha}}{1 + \beta - \alpha - \frac{1}{m} - \alpha'} = \frac{\frac{\beta}{1-\alpha}}{1 + \beta - \alpha - \alpha'} =: f_4^\infty(\alpha, \beta, \alpha')$$

and the lower bound on the SIS probability approaches

$$\begin{aligned} g^\infty(\alpha',\beta') &= \lim_{m\to\infty} \frac{(\beta'-\alpha')(1+\beta'-\alpha')}{\beta'(1+\beta')+1} \\ &\geq \lim_{m\to\infty} \frac{(\beta-\alpha-\frac{1}{m}-\alpha')(1+\beta-\alpha-\frac{1}{m}-\alpha')}{\frac{\beta}{1-\alpha}(1+\frac{\beta}{1-\alpha})+1} \\ &= \frac{(\beta-\alpha-\alpha')(1+\beta-\alpha-\alpha')}{\frac{\beta}{1-\alpha}(1+\frac{\beta}{1-\alpha})+1} \\ &\geq \frac{(\beta-\alpha-\alpha')(1+\beta-\alpha-\alpha')}{\frac{\beta}{1-\alpha}(\frac{1}{1-\alpha}+\frac{\beta}{1-\alpha})+\frac{1}{(1-\alpha)^2}} =: g_4^\infty(\alpha,\beta,\alpha'), \end{aligned}$$

where the last inequality holds as long as $\alpha < 1$. Notice that for fixed α, β satisfying $\alpha < \min\{1, \beta\}$, the function f_4^∞ is increasing and g_4^∞ is decreasing in α', for $\alpha' < \beta - \alpha$. Similarly, for fixed α', β satisfying $\alpha' < \beta$, the function f_4^∞ is increasing and g_4^∞ is decreasing in α, for $\alpha < \min\{1, \beta - \alpha'\}$. Therefore, if we take $\alpha = \alpha' < \min\{1, \beta/2\}$, we will have the bounds

$$f_4^\infty(x,\beta,y) \leq f_4^\infty(\alpha,\beta,\alpha) \qquad \text{and} \qquad g_4^\infty(x,\beta,y) \geq g_4^\infty(\alpha,\beta,\alpha)$$

for any $x, y \leq \alpha$. Recall that $\triangle_\beta = g^\infty(0,\beta) - f^\infty(0,\beta) = g_4^\infty(0,\beta,0) - f_4^\infty(0,\beta,0) > 0$. If we choose α so that $g_4^\infty(\alpha,\beta,\alpha) - f_4^\infty(\alpha,\beta,\alpha) \geq \triangle_\beta/2$, then in similar fashion to the last paragraph of the proof of Lemma 3, we may conclude that the SIS procedure likely fails. More precisely, let $\epsilon := \triangle_\beta/8$ and let $f := f_4^\infty(\alpha,\beta,\alpha)+\epsilon$ and $g := g_4^\infty(\alpha,\beta,\alpha)-\epsilon$ be the upper bound (for sufficiently large m) on the true probability and the lower bound on the SIS probability of 1 appearing at the position $(m+1, i)$ for $i \in I := \{k+1, \ldots, k+\lfloor\alpha' m'\rfloor\}$. Therefore Lemma 2 with parameters $\alpha(1-\alpha)$, β, I of size $|I| = \lfloor\alpha' m'\rfloor \geq \lfloor\alpha(1-\alpha)m\rfloor$, f, and g implies the statement of the theorem.

Finally, if the SIS procedure constructs the tables row-by-row instead of column-by-column, symmetrical arguments hold. This completes the proof of Theorem 1. □

4 Experiments

We performed several experimental tests which show sequential importance sampling to be a promising approach for certain classes of input instances. We discuss the experiments in more detail and present supporting figures in the full version of this paper [2].

Our first set of experiments tested the SIS technique on regular input sequences, i.e., $r_i = c_j$ for all i, j. It appears the approach is very efficient for these input sequences. We considered input sequences which were 5, 10, $\lfloor 5 \log n\rfloor$ and $\lfloor n/2\rfloor$-regular, and $n \times n$ matrices with $n = 10, 15, 20, \ldots, 100$. The required number of SIS trials until the algorithm converged resembled a linear function of n.

In contrast, we examined the evolution of SIS on the negative example from Theorem 1. In our simulations we used the more delicate sampling mentioned in footnote 1, which guarantees that the assignment in every column is valid, i. e., such an assignment can always be extended to a valid table (or, equivalently, the random variable X_t is always strictly positive). We ran the SIS algorithm under three different settings: first, we constructed the tables column-by-column where the columns were ordered in decreasing order of their sums, as suggested in the paper of Chen et al. [4]; second, we ordered the columns in increasing order of their sums; and third, we constructed the tables row-by-row where the rows were ordered in decreasing order of their sums.

The experiments confirmed the poor performance described in Theorem 1. For $m = 300$, $\beta = .6$ and $\gamma = .7$, even the best of the three estimators differed from the true value by about a factor of 40, while some estimates were off by more than a factor of 1000.

References

1. I. Bezáková, N. Bhatnagar, and E. Vigoda, Sampling Binary Contingency Tables with a Greedy Start. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA), 2006.
2. I. Bezáková, A. Sinclair, D. Štefankovič, and E. Vigoda. Negative Examples for Sequential Importance Sampling of Binary Contingency Tables. Submitted. Available from Mathematics arXiv math.ST/0606650.
3. I. Bezáková, D. Štefankovič, V. Vazirani, and E. Vigoda. Accelerating Simulated Annealing for the Permanent and Combinatorial Counting Problems. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA), 2006.
4. Y. Chen, P. Diaconis, S. Holmes, and J.S. Liu, Sequential Monte Carlo Methods for Statistical Analysis of Tables. *Journal of the American Statistical Association*, 100:109-120, 2005.
5. M. De Iorio, R. C. Griffiths, R. Lebois, and F. Rousset, Stepwise Mutation Likelihood Computation by Sequential Importance Sampling in Subdivided Population Models. *Theor. Popul. Biol.*, 68:41-53, 2005.
6. M.R. Jerrum, A. Sinclair and E. Vigoda, A Polynomial-time Approximation Algorithm for the Permanent of a Matrix with Non-negative Entries. *Journal of the Association for Computing Machinery*, 51(4):671-697, 2004.
7. J. Miguez and P. M. Djuric, Blind Equalization by Sequential Importance Sampling. *Proceedings of the IEEE International Symposium on Circuits and Systems*, 845-848, 2002.
8. J. L. Zhang and J. S. Liu, A New Sequential Importance Sampling Method and its Application to the Two-dimensional Hydrophobic-Hydrophilic Model. *J. Chem. Phys.*, 117:3492-8, 2002.

Estimating Entropy over Data Streams

Lakshminath Bhuvanagiri and Sumit Ganguly

Indian Institute of Technology, Kanpur
{blnath, sganguly}@cse.iitk.ac.in

Abstract. We present an algorithm for estimating entropy of data streams consisting of insertion and deletion operations using $\tilde{O}(1)$ space.[1]

1 Introduction

Recently, there has been an emergence of *monitoring applications* in diverse areas, including, network traffic monitoring, network topology monitoring, sensor networks, financial market monitoring, web-log monitoring, etc.. In these applications, data is generated rapidly and continuously, and must be analyzed very efficiently, in real-time, to identify large trends, anomalies, user-defined exception conditions, etc.. The data streaming model [1, 12] has gained popularity as a computational model for such applications—where, incoming data (or updates) are processed very efficiently and in an online fashion using space that is much less than what is needed to store the data in its entirety.

A data stream $\mathcal{S}$ is viewed as a sequence of arrivals of the form (i, v), where, i is the identity of an item that is a member of the domain $\{0, 1, \ldots, N-1\}$, and v is the *update* to the frequency of the item. $v > 0$ indicates an insertion of multiplicity v, while $v < 0$ indicates a corresponding deletion. The frequency of an item i, denoted by f_i, is the sum of the updates to i since the inception of the stream, that is, $f_i = \sum_{(i,v) \text{ appears in } \mathcal{S}} v$. In the *strict update model*, deletions are assumed to correspond to prior insertions, and, therefore the frequency of an item is always non-negative. In the *general update model*, item frequencies may be negative or positive. Let n denote the number of items with non-zero frequencies in the stream and let m denote the sum of the absolute values of the item frequencies.

The entropy H of a data stream is defined as $H = \sum_{i:f_i>0} \frac{|f_i|}{m} \log \frac{m}{|f_i|}$. We study the problem of continuously tracking the entropy of a data stream in low space. The entropy of a data stream, or that of a frequency distribution, measures its information theoretic *randomness* and *incompressibility*. A value of entropy close to $\log n$, is indicative that the frequencies in the stream are randomly distributed, whereas, low values are indicative of "patterns" in the data. Further, monitoring changes in the entropy of a network traffic stream has been used detect anomalies [8, 14, 15].

[1] We use the $\tilde{O}$ notation to simplify complexity expressions. If N is the domain size of the stream items and m is the sum of the absolute values of the item frequencies, we say that $f(m, N) = \tilde{O}(g(m, N))$, if $f(m, N) = O\left(\frac{1}{\epsilon^{O(1)}}(\log^{O(1)} m)(\log^{O(1)} n)\ g(m, N)\right)$.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 148–159, 2006.

Prior Work. The work in [9] presents an algorithm for estimating H over insert-only streams to within a factor of $1 \pm \epsilon$ using space $\tilde{O}(n^{\frac{1}{1+\epsilon}})$. [9] also presents an algorithm for estimating H using space $\tilde{O}(1)$ bits in a *random*-streaming model, in which, the order of arrival of items is assumed to be completely random. [3] presents an algorithm for estimating entropy H over insert-only streams to within factors of $1 \pm \epsilon$ using $\tilde{O}(\min(m^{2/3}, m^{2(1-\epsilon)})$ space.

Contributions. In this paper we present an algorithm that estimates the entropy of strict update data streams to within factors of $1 \pm \epsilon$ using space $\tilde{O}(1)$. We also show how the algorithm can be generalized to the general update streaming model using space $\tilde{O}(1)$. We prove the following theorem.

Theorem 1. *There exists an algorithm for strict update streams that returns an estimate $\hat{H}$ of the entropy H of a stream such that $|\hat{H} - H| \leq \epsilon H$ with probability $1 - \delta$ using $O\left(\frac{(\log^4 m)}{\epsilon^3} \frac{(\log m + \log \frac{1}{\epsilon})}{(\log \frac{1}{\epsilon} + \log\log m)} (\log \frac{1}{\delta})\right)$ bits.*

Organization. The remainder of the paper is organized as follows. In Section 2, we present an abstract algorithm called Hss for estimating a class of data stream metrics and then use it in Section 3 to estimate entropy.

2 The Hss Algorithm

We present a procedure for obtaining a *representative sample* over the input stream, which we refer to as *Hierarchical Sampling over Sketches* (Hss) and use it for estimating a class of metrics over data-streams of the following form.

$$\Psi(\mathcal{S}) = \sum_{i:f_i>0} \psi(f_i) \tag{1}$$

Section 3 specializes this procedure to yield a straight forward algorithm for estimating the entropy of a data stream in $\tilde{O}(1)$ space. The algorithm can be naturally adapted for general update streams. A specialization of the Hss procedure was used in [2] to give an algorithm for finding the k^{th} frequency moment $F_k = \sum f_i^k$.

Preliminaries. Given a data stream, *rank*(r) returns an item with the r^{th} largest frequency (ties are broken arbitrarily). We say that an item i has rank r if *rank*$(r) = i$. For a given value of k, $1 \leq k \leq N$, the set *top*(k) is the set of items with rank $\leq k$. We use the Count-Min algorithm [6] for estimating the frequency $\hat{f}_i$ of an item i. Its guarantees are summarized in Theorem 2 and are given in terms of the quantity $m^{res}(k) = \sum_{i \notin top(k)} f_i = \sum_{r>k} f_{rank(r)}$.

Theorem 2. [6] *For $0 < \epsilon < 1$, the* Count-Min *algorithm uses space $O(\frac{k}{\epsilon} \log \frac{1}{\delta} \log m)$ bits and time $O(\log \frac{1}{\delta})$ to process each stream update. It returns an estimate $\hat{f}_i$ that satisfies $f_i \leq \hat{f}_i \leq f_i + \frac{\epsilon m^{res}(k)}{k}$ with probability $1 - \delta$.* □

The HSS *structure.* Let $T_0 > T_1 > \ldots > T_L$ (L will be fixed later) be a sequence of exponentially decreasing thresholds that partition the elements of the stream into groups $G_0 \ldots G_L$, where, $G_0 = \{i \in \mathcal{S} : \ f_i \geq T_0\}$ and $G_l = \{i \in \mathcal{S} : \ T_l \leq f_i < T_{l-1}\}$, $\ 1 \leq l \leq L$. Intuitively, the HSS algorithm works as follows. From the input stream $\mathcal{S}$, we create sub-streams $\mathcal{S}_0 \ldots \mathcal{S}_L$ such that $\mathcal{S}_0 = \mathcal{S}$ and for $1 \leq l \leq L$, $\mathcal{S}_l$ is obtained from $\mathcal{S}_{l-1}$ by sub-sampling each distinct item appearing in $\mathcal{S}_{l-1}$ independently with probability $\frac{1}{2}$ (hence $L = O(\log m)$). The sub-stream $\mathcal{S}_l$ is referred to as the sub-stream at level l, for $l = 0, 1, \ldots, L$. Let k be a space parameter. At each level l, we keep a data-structure denoted by $\mathcal{D}_l$, that takes as input the sub-stream $\mathcal{S}_l$, and returns an *approximation to the top*(k) *items of its input stream and their frequencies.* This data-structure is typically instantiated using standard synopsis structures, such as, COUNT-MIN (for estimating entropy), COUNTSKETCH [4] (for estimating frequency moments [2] and for estimating entropy when frequencies may be negative), etc. We posit the following invariant.

> **C1:** All items of G_l present in $\mathcal{S}_l$ must be discovered as frequent items by $\mathcal{D}_l$.

Approximating f_i. We assume that frequent items discovery and frequency estimation algorithms have an additive error of at most Δ in the estimated frequencies (usually, with high probability [4, 5, 6]), where Δ is a function of the space parameter k and some aggregate statistic of the input stream (e.g., $m^{res}(k)$ for the COUNT-MIN algorithm and $F_2^{res}(k)$ for the COUNTSKETCH algorithm). We use $\Delta_l = \Delta_l(k)$ to denote the error incurred by the estimates obtained from $\mathcal{D}_l$ operating on the sub-stream $\mathcal{S}_l$.

Let Q_l denote a frequency threshold defined as $Q_l = \frac{\Delta_l}{\bar{\epsilon}}$ and let $\hat{f}_{i,l}$ denote the estimate of the frequency of i as obtained from the data structure $\mathcal{D}_l$, assuming that the item i has been sampled at level l. It follows that if $\hat{f}_{i,l} > Q_l$, then, $|\hat{f}_{i,l} - f_i| \leq \bar{\epsilon} f_i$ with high probability. Lemma 1 establishes a relation between the values Δ_l for various l for a popular *top*(k) estimation algorithm, namely the COUNT-MIN sketch [2]. This relationship helps us to set the threshold Q_l to $\frac{\Delta_0(k)}{\bar{\epsilon} \cdot 2^l} = \frac{Q_0}{2^l}$ for $l > 0$.

Lemma 1. *For* COUNT-MIN *sketch algorithm,* $\Delta_l(k) \leq \frac{\Delta_0(2^{l-1}k)}{2^l}$ *with probability* $\geq 1 - 2^{-\Omega(k)}$ *for* $l \geq 1$ *and* $k \geq 36$.

Proof. By Theorem 2, $\Delta_l(k) = \frac{m^{res}(k,l)}{k}$, where, $m^{res}(k, l)$ is the (random) residual first moment of $\mathcal{S}_l$ when the top-k ranked items have been removed from $\mathcal{S}_l$. The expected number of the top-$2^{l-1}k$ ranked items in $\mathcal{S}$ appearing in $\mathcal{S}_l$ is $\frac{1}{2^l} 2^{l-1} k = \frac{k}{2}$. By Chernoff's bounds, the number of the top-$\frac{2^l k}{2}$ ranked items in $\mathcal{S}$ appearing in $\mathcal{S}_l$ is no more than $\frac{3k}{4}$, with probability at least $1 - 2^{-\Omega(k)}$. In other words, the non-top-k elements of $\mathcal{S}_l$ only includes the non-top-$2^{l-1}k$ elements of the original stream, with probability $1 - 2^{-\Omega(k)}$. Therefore, $\mathsf{E}\big[m^{res}(k, l)\big] \leq$

[2] A similar result can also be shown for COUNTSKETCH .

$\frac{1}{2^l} m^{res}(2^{l-1}k)$. By a similar argument, the largest non-top-k frequency, u_l, in $\mathcal{S}_l$ has rank at least $\frac{3 \cdot 2^l k}{4}$ in $\mathcal{S}$. Thus, $u_l \leq f_{rank(3 \cdot 2^{l-2} \cdot k)} \leq \frac{m^{res}(2^{l-1}s)}{2^{l-2}k}$. Items with ranks between $\frac{3 \cdot 2^l k}{4}$ and $2^{l-1}k$ have frequencies at least u_l, and their sum is at most $m^{res}(2^{l-1}k)$. That is, $(\frac{3}{4})(2^l k)u_l \leq m^{res}(2^{l-1}k)$, or that, $u_l \leq \frac{m^{res}(2^{l-1}k)}{3 \cdot 2^{l-2}k}$.

Hence, with probability $\geq 1-\delta$, $m^{res}(k,l) \leq \max(2\mathsf{E}\big[m^{res}(k,l)\big], 3u_l \log \frac{1}{\delta})) \leq \max(2\frac{m^{res}(2^{l-1}k)}{2^l}, 3\frac{m^{res}(2^{l-1}k)}{2^{l-2}k} \log \frac{1}{\delta})$ (from Hoeffding's bounds). Let $\delta = 2^{-\frac{k}{16}}$. Since $k \geq 36$, we get $m^{res}(k,l) \leq \frac{m^{res}(2^{l-1}k)}{2^l}$ with probability $> 1 - 2^{-\Omega(k)}$. □

Disambiguating estimated frequency. It is possible for the estimate $\hat{f}_{i,l}$ of an item i obtained from the sub-stream $\mathcal{S}_l$ to exceed the threshold Q_l for multiple l. For example, consider an item with actual frequency larger than Q_0. It crosses the threshold at level 0 and thus is estimated accurately at level 0. However, it may be sub-sampled at level 1, and in this case, its frequency estimate also crosses the threshold Q_1 (with high probability). In this manner, this item may get estimated accurately at all the levels at which it has been successfully sub-sampled. Each of these estimates $\hat{f}_{i,l}$ may be different, though they are all within factors of $1 \pm \bar{\epsilon}$ to the actual value. We therefore apply the "disambiguation-rule" of using the estimate obtained from the *lowest level* at which it crosses the threshold for that level. The estimated frequency after disambiguation is denoted as $\hat{f}_i$.

Setting T_l. As per the invariant **C1**, all elements in $G_l \cap S_l$ must be discovered as frequent items by $\mathcal{D}_l$. Since, Q_l defines the threshold for "good estimation", fixing $T_l = Q_l$ might seem a possibility. However, the elements of G_l with frequency close to $T_l (= Q_l)$ might fail to even appear in the sample $\bar{G}_l$ due to errors in estimation, thereby violating **C1**. One way to solve this problem is to choose $T_l = \sqrt{Q_l \cdot Q_{l+1}} = 2^{1/2} \cdot Q_l$. Since $Q_l(1+\bar{\epsilon}) \geq Q_l + \Delta_l$ (by definition of Q_l), we choose $\bar{\epsilon}$ such that $(1+\bar{\epsilon}) < 2^{1/2}$, and hence $T_l \geq Q_l + \Delta_l$. Thus, any $i \in G_l$ appearing in $\mathcal{S}_l$ will be present in the *top*(k) set returned by $\mathcal{D}_l$, and hence included into the sample $\bar{G}_l$ since it can suffer an additive error of at most $\pm\Delta_l$.

2.1 Algorithm

Obtaining hierarchical samples. For every stream update (i,v), we use a hash-function $h : \{1 \ldots N\} \rightarrow \{1 \ldots N\}$ to map the item onto level $u = lsb(h(i))$ [3]. The update (i,v) is then propagated to the frequent items data structures $\mathcal{D}_l$ for $0 \leq l \leq u$, in effect, i is included in the sub-streams from level 0 to level u. The hash function is assumed to be chosen randomly from a fully independent family; later we reduce the number of random bits required.

At inference time, the algorithm collects samples as follows. From each level l, the set of items whose estimated frequency crosses the threshold Q_l are identified, using the frequent items structure $\mathcal{D}_l$. If an item crosses the threshold at multiple levels, then, the disambiguation rule is applied that sets $\hat{f}_i$ to $\hat{f}_{i,l}$, where, l is the smallest of the levels r such that $\hat{f}_{i,r} \geq Q_r$. Based on their disambiguated

[3] $lsb(x)$ is the position of the least significant "1" in binary representation of x.

frequencies, the sampled items are sorted into their respective groups. In order to maintain the invariant **C1**, we include an item i in group G_l only if it hashes to level l. More, precisely, we form the sampled groups, $\bar{G}_0, \bar{G}_1, \ldots, \bar{G}_L$, as follows.

$$\bar{G}_0 = \{i : \hat{f}_i \geq T_0\} \text{ and } \bar{G}_l = \{i : T_{l-1} < \hat{f}_i \leq T_l \text{ and } i \in \mathcal{S}_l\}, 1 \leq l \leq L \ .$$

Note that any item belonging to G_l and $\mathcal{S}_l$ is "discovered" at level l, with high probability . However, if f_i is close to the right or the left boundary of G_l, the $\pm\bar{\epsilon}f_i$ estimation error could cause i to be misclassified into its adjacent group. We consider this issue in the next section.

Estimator. The sample is used to compute the estimate $\hat{\Psi}$. We also define an idealized estimator $\bar{\Psi}$ that assumes that the frequent items structure is an oracle that does not make errors.

$$\hat{\Psi} = \sum_{l=0}^{L} \sum_{i \in \bar{G}_l} \psi(\hat{f}_i) \cdot 2^l \qquad \bar{\Psi} = \sum_{l=0}^{L} \sum_{i \in \bar{G}_l} \psi(f_i) \cdot 2^l \tag{2}$$

2.2 Analysis

Let $x_{i,r}$ denote a random variable which takes the value 1 iff $i \in \mathcal{S}_r$ and is also classified by the algorithm into $\bar{G}_r$. Thus, equation (2) can be written as follows.

$$\bar{\Psi} = \sum_{i \in \mathcal{S}} \psi(f_i) \sum_{r=0}^{L} x_{i,r} \cdot 2^r, \qquad \hat{\Psi} = \qquad \sum_{i \in \mathcal{S}} \psi(\hat{f}_i) \sum_{r=0}^{L} x_{i,r} \cdot 2^r$$

Lemma 2 shows that the expected value of $\bar{\Psi}$ is close to Ψ.

Lemma 2. *Suppose that for* $0 \leq i \leq N-1$ *and* $0 \leq l \leq L$, $|\hat{f}_{i,l} - f_i| \leq \epsilon f_i$ *with probability* $\geq 1 - 2^{-t}$. *Then* $|\mathsf{E}[\bar{\Psi}] - \Psi| \leq \Psi \cdot 2^{-t+\log L}$.

Proof. $\mathsf{E}[\bar{\Psi}] = \sum_{i \in \mathcal{S}} \mathsf{E}[\psi(f_i) \sum_{r=0}^{L} x_{i,r} \cdot 2^r] = \sum_{i \in \mathcal{S}} \psi(f_i) \sum_{r=0}^{L} \mathsf{E}[x_{i,r} \cdot 2^r]$, where, $\sum_{r=0}^{L} \mathsf{E}[x_{i,r} \cdot 2^r] = \sum_{r=0}^{L} \mathsf{Pr}\{x_{i,r} = 1\} \cdot 2^r$.

Consider an item $i \in G_l$. The frequency group (or interval) G_l is partitioned into three sub-regions, namely, $lr(G_l) = [T_l, T_l + \bar{\epsilon}Q_l]$, $rr(G_l) = [T_{l-1} - \bar{\epsilon}Q_l, T_{l-1}]$ and $mr(G_l) = [T_l + \bar{\epsilon}Q_l, T_{l-1} - \bar{\epsilon}Q_l]$, that, respectively denote the *left-region*, *right-region* and *middle-region* of the group G_l. An item i is said to belong to one of these regions if its true frequency lies in that region. As seen earlier, if $i \in G_l$ and $i \in \mathcal{S}_l$, then $\hat{f}_{i,l} > Q_l$, and hence, it is discovered by $\mathcal{D}_l$ (with high probability). We now consider items that lie in the middle-region and the left-region respectively (the argument for right-region is analogous to that of the left-region).

Let $i \in mr(G_l)$. Then, $T_l \leq f_i - \Delta_l \leq \hat{f}_{i,l} \leq f_i + \Delta_l \leq T_{l-1}$, that is, the error Δ_l is not large enough to cause $\hat{f}_{i,l}$ to cross either T_l or T_{l-1}. Hence,

with probability $1-2^{-t}$, if $i \in \mathcal{S}_l$, then, is correctly classified into group $\bar{G}_l$. Therefore $\frac{1}{2^l}(1-2^{-t}) \leq \mathsf{Pr}\{x_{i,l}=1\} \leq \frac{1}{2^l}$, and $\mathsf{Pr}\{x_{i,r}=1\} \leq \frac{1}{2^r}\cdot 2^{-t}$ for $r \neq l$. Thus, $\sum_{r=0}^{L} \mathsf{E}\big[x_{i,r}2^r\big] \leq \frac{1}{2^l}\cdot 2^l + \sum_{r\neq l} 2^{-t}\cdot\frac{1}{2^r}\cdot 2^r \leq 1 + L\cdot 2^{-t}$, and $\sum_{r=0}^{L}\mathsf{E}\big[x_{i,r}2^r\big] \geq \frac{1}{2^l}\cdot(1-2^{-t})\cdot 2^l$. Hence, $|\sum_{r=0}^{L}\mathsf{E}\big[x_{i,r}2^r\big]-1| \leq 2^{-t+\log L}$.

Let $i \in lr(G_l)$. Then, with high probability, i will not be discovered at any level $l' < l$, since $\hat{f}_{i,l'} \leq f_i + \bar{\epsilon}Q_{l'} \leq (1+\bar{\epsilon})T_l + \bar{\epsilon}Q_{l'} < Q_{l'}$. Therefore, the estimate $\hat{f}_i = \hat{f}_{i,l}$ is obtained from level l. However, by virtue of its true frequency (f_i) being close to T_l, the estimate $\hat{f}_{i,l}$ might be on either side of T_l causing i to be classified into either $\bar{G}_l$ or $\bar{G}_{l+1}$. Let p be the probability that $\hat{f}_{i,l} > T_l$, that is, the item gets "correctly" classified into group $\bar{G}_l$. Therefore, $(1-2^{-t})\cdot p\cdot\frac{1}{2^l} \leq \mathsf{Pr}\{x_{i,l}=1\} \leq p\cdot\frac{1}{2^l}$. The probability of the same i getting classified into $\bar{G}_{l+1}$ at $\mathcal{D}_l$ is $1-p$, resulting in $(1-p)\cdot(1-2^{-t})\cdot 2^{-(l+1)} \leq \mathsf{Pr}\{x_{i,l+1}=1\} \leq (1-p)\cdot\frac{1}{2^{l+1}}$. Note that this argument assumes that the random sub-sampling choices for an item are made independent of the choices of the other items. Therefore, $\sum_{r=0}^{L}\mathsf{E}\big[x_{i,r}\ 2^r\big] \leq (p\ \frac{1}{2^l})\ 2^l + (1-p)\ \frac{1}{2^{l+1}}\ 2^{l+1} + \sum_{r\notin\{l,l-1\}}\frac{1}{2^r}\ 2^r\ 2^{-t}$. Therefore, $1-2^{-t} \leq \sum_{r=0}^{L}\mathsf{E}\big[x_{i,r}\ 2^r\big] < 1 + L\cdot 2^{-t}$. Therefore, $|\mathsf{E}\big[\sum_{r=0}^{L} x_{i,r}\ 2^r - 1\big]| \leq 2^{-t+\log L}$. A similar argument can be made for $rr(G_l)$. Combining, we get

$$|\mathsf{E}\big[\bar{\Psi}\big] - \Psi(\mathcal{S})| = \sum_{i\in\mathcal{S}} \psi(f_i)\ |\mathsf{E}\big[x_{i,r}\ 2^r\big] - 1| \leq \Psi(\mathcal{S})\cdot 2^{-t+\log L}\ .$$

□

We now present a bound on the variance of the idealized estimator. For any item i with non-zero frequency, we denote by $l(i)$ the group index l such that $i \in G_l$.

Lemma 3. *Suppose that for all $0 \leq i \leq N-1$ and $0 \leq l \leq L$, $|\hat{f}_{i,l} - f_i| \leq \epsilon f_i$ with probability $\geq 1-2^{-t}$. Then,*

$$\mathsf{Var}\big[\bar{\Psi}\big] \leq \sum_{i\in\mathcal{S}} 2^{-t+L+2}\cdot\psi^2(f_i) + \sum_{i\notin(G_0-lm(G_0))} \psi^2(f_i)\cdot 2^{l(i)+1}\ .$$

Proof. The proof, analogous to that of Lemma 2, is given in Appendix A. □

Corollary 1. *If the function $\psi(\cdot)$ is increasing in the interval $[0\ldots T_0+\Delta_0]$, then, choosing $t = L + \log\frac{1}{\epsilon^2} + 2$ we get*

$$\mathsf{Var}\big[\bar{\Psi}\big] \leq \sum_{i\in\mathcal{S}} \epsilon^2\psi^2(f_i) + \sum_{l=1}^{L}\sum_{i\in G_l}\psi(T_{l-1})\psi(f_i)2^{l+1} + 2\sum_{i\in lm(G_0)}\psi(T_l+\Delta_0)\psi(f_i) \tag{3}$$

Proof. If the monotonicity condition is satisfied, then $\psi(T_{l-1}) > \psi(f_i)$ for all $i \in G_l$, $l \geq 1$ and $\psi(f_i) \leq \psi(T_0+\Delta_0)$ for $i \in lm(G_0)$. Therefore, $\psi^2(f_i) \leq \psi(T_{l-1})\cdot\psi(f_i)$, in the first case and $\psi^2(f_i) \leq \psi(T_0+\Delta_0)$ in the second case. By Lemma 3 and the chosen value for t gives the desired result. □

2.3 Error in the Estimate

The error incurred by our estimate $\hat{\Psi}$ is $|\hat{\Psi} - \Psi|$, and can be written as the sum of two error components using triangle inequality.

$$|\hat{\Psi} - \Psi| \leq |\hat{\Psi} - \bar{\Psi}| + |\bar{\Psi} - \Psi| = \mathcal{E}_1 + \mathcal{E}_2$$

Here, $\mathcal{E}_1 = |\Psi - \bar{\Psi}|$ is the error due to sampling and $\mathcal{E}_2 = |\hat{\Psi} - \bar{\Psi}|$ is the error due to the estimation of the frequencies. By Chebychev's inequality,

$$\mathcal{E}_1 = |\Psi - \bar{\Psi}| \leq |E[\bar{\Psi}] - \Psi| + 3\sqrt{\mathsf{Var}[\bar{\Psi}]} \text{ with probability } \frac{8}{9} \ .$$

Using Lemma 2 and Corollary 1, and choosing $t = L + \log \frac{1}{\epsilon^2} + 2$, the expression for $\mathcal{E}_1$ can be simplified as follows.

$$\mathcal{E}_1 \leq \frac{\epsilon^2 L \Psi}{m} + 3\Big(\sum_{i \in \mathcal{S}} \epsilon^2 \psi^2(f_i) + \sum_{i \in G_l, l \geq 1} \psi(T_{l-1})\psi(f_i)2^{l+1} + \sum_{i \in lm(G_0)} 2\psi(T_l + \Delta_0)\psi(f_i)\Big)^{1/2} \tag{4}$$

with probability $\frac{8}{9}$. We now present an upper bound on $\mathcal{E}_2$.

Lemma 4. *Suppose that for $0 \leq i \leq N-1$ and $0 \leq l \leq L$, $|\hat{f}_{i,l} - f_i| \leq \epsilon f_i$ with probability $\geq 1 - 2^{-t}$. Then, $\mathcal{E}_2 \leq 16 \cdot \bar{\epsilon} \cdot Q_0 \sum_{l=0}^{L} \sum_{i \in G_l} \frac{|\psi'(\xi_i)|}{2^l}$ with probability $\geq \frac{9}{10} - 2^{-t}$, where for an $i \in G_l$, ξ_i lies between f_i and $\hat{f}_i$, and maximizes $\psi'()$.*

Proof. Let $y_{i,l}$ denote the indicator random variable that is 1 if $i \in \mathcal{S}_l$ and is 0 otherwise. Note that $y_{i,l+1}$ is 1 only if $y_{i,l}$ is 1. Let $i \in G_l$. By arguing similarly to that of Lemma 2, we have the following cases. Consider the set of items i such that $i \in mr(G_l)$, for some $l \geq 1$, or, $i \in G_0 - lr(G_0)$. If $r = l$, then, $x_{i,r} = y_{i,r}$, with probability $1 - 2^{-t}$, and otherwise, $x_{i,r} = 0$, with probability at most 2^{-t}. Therefore, for such an item i , $\sum_{r=0}^{L} x_{i,r} 2^r = y_{i,l} 2^l$, with probability $1 - 2^{-t}$.

Suppose $i \in lr(G_l)$, for some $l \geq 0$. Then, $x_{i,l} + x_{i,l+1} = y_{i,l}$, with probability $1 - 2^{-t}$, and $x_{i,r} = 0$, with probability at most 2^{-t}, for $r \notin \{l, l+1\}$. Therefore, for such items i, $\sum_{r=0}^{L} x_{i,r} 2^r = x_{i,l} 2^l + x_{i,l+1} 2^{l+1} \leq (x_{i,l} + x_{i,l+1}) 2^{l+1} = y_{i,l} 2^{l+1}$, with probability $1 - 2^{-t}$.

Finally, consider those items i such that $i \in rr(G_l)$ for some $l \geq 1$. Using a similar argument, we can show that $x_{i,l} + x_{i,l-1} = y_{i,l-1}$, with probability $1 - 2^{-t}$, and $x_{i,r} = 0$, with probability 2^{-t} for $r \notin \{l, l-1\}$. Therefore, $\sum_{r=0}^{L} x_{i,r} 2^r \leq y_{i,l-1} 2^l$. Since, $y_{i,l}$ is 1 only if $y_{i,l-1} = 1$, for $l \geq 1$, in all cases, we have,

$$\sum_{r=0}^{L} x_{i,r} \cdot 2^r \leq y_{i,l(i)-1} \cdot 2^{l+1}, \text{ with probability } \geq 1 - 2^{-t}, \text{ if } l(i) \geq 1 \ . \tag{5}$$

By triangle inequality, $\mathcal{E}_2 \leq \sum_{l=0}^{L} \sum_{i \in G_l} |\psi(\hat{f}_i) - \psi(f_i)| \cdot (\sum_{r=0}^{L} x_{i,r} \cdot 2^r)$. Using Taylor's expansion, $\mathcal{E}_2 \leq \sum_{l=0}^{L} \sum_{i \in G_l} |\Delta_l \cdot \psi'(\xi_i)| \cdot (\sum_{r=0}^{L} x_{i,r} \cdot 2^r)$, where ξ_i is

the value between f_i and $\hat{f}_i$ at which $\psi'()$ takes its maximum absolute value. Using (5)

$$\mathcal{E}_2 \leq 2\Delta_0 \sum_{i \in G_0} |\psi'(\xi_i)| y_{i,0} + \sum_{l=1}^{L} \Delta_l \sum_{i \in G_l} |\psi'(\xi_i)|\ y_{i,l-1}\ 2^{l+1}.$$

Since $\mathcal{S}_0 = \mathcal{S}$, we have $y_{i,0} = 1,\ \forall i \in G_0$. Applying Hoeffding's bounds to the second term above , we obtain with probability $\geq \frac{9}{10} - 2^{-t}$,

$$\begin{aligned}\mathcal{E}_2 &\leq 2\Delta_0\big(\sum_{i \in G_0} |\psi'(\xi_i)|\big) + 2 \cdot \big(4 \sum_{l=1}^{L} \Delta_l \sum_{i \in G_l} |\psi'(\xi_i)| + 4 \max_{i \in \mathcal{S} - G_0} |\Delta_{l(i)}\ \psi'(\xi_i)|\big) \\ &\leq 16 \sum_{l=0}^{L} \Delta_l \sum_{i \in G_l} |\psi'(\xi_i)| \leq 16 \cdot \bar{\epsilon} \cdot Q_0 \sum_{l=0}^{L} \sum_{i \in G_l} \frac{|\psi'(\xi_i)|}{2^l},\ \text{since, } \Delta_l = \frac{\bar{\epsilon} Q_0}{2^l}.\ \square\end{aligned}$$

Reducing random bits. The number of random bits used by the algorithm can be reduced to $O(s \log m)$, where, s is the space used by the Hss structure, by using a classical result of Nisan [13] on pseudo-generators for space bounded computation, as adapted for use by data stream algorithms by Indyk [10]. Since, this approach has been adequately treated in [10] and [11], we do not discuss it in greater detail.

3 Estimating Entropy

In this section, we apply the Hss algorithm to estimate the entropy $H = \sum_{i:f_i>0} \frac{f_i}{m} \log \frac{m}{f_i}$ of a data stream. We assume that the stream follows the strict update model (i.e., $f_i \geq 0$). Later we remark how the algorithm can be modified for general update streams. For any $0 \leq x \leq m$, let $h(x)$ denote $\frac{x}{m} \log \frac{m}{x}$ (we assume that $h(0) = 0$). In this section, the function $\psi(x) = h(x)$ and the statistic $\Psi = \sum_i h(f_i) = H$.

We instantiate the Hss algorithm using Count-min sketch [6] as the frequent items structure $\mathcal{D}_l$ with $\frac{8k}{\bar{\epsilon}}$ buckets in each hash table, where $\bar{\epsilon} = \epsilon$ itself. We also estimate $m^{res}(k)$ to within accuracy factors of $1 \pm \epsilon$ with probability $1 - \delta$. This is done using an algorithm similar to that of estimating $F_2^{res}(k)$ presented in [7], and uses space $O(\frac{k}{\epsilon} \log \frac{m}{\delta} \log m)$ bits. For brevity, we state the theorem without proof.

Theorem 3. *For a given integer $k \geq 1$ and $0 < \epsilon < 1$, there exists an algorithm for strict update streams that returns an estimate $\hat{m}^{res}(k)$ satisfying $(1 - \epsilon)m^{res}(k) \leq \hat{m}^{res}(k) \leq m^{res}(k)$ with probability $1 - \delta$ using $O(\frac{k}{\epsilon}(\log \frac{k}{\delta})(\log m))$ bits.* $\square$

We use the algorithm of Theorem 3 with $\delta = (20mL)^{-1}$ to obtain an estimate $\hat{m}^{res}(k)$ and use it to compute the thresholds Q_l and T_l, for levels $l = 0, 1, \ldots, L$, as follows, where $L = \lceil \log \frac{m}{k} \rceil$: $Q_0 = \frac{\hat{m}^{res}(k)}{k}$, $Q_l = \frac{Q_0}{2^l}$ and $T_l = 2^{1/2} Q_l$, for $0 \leq l \leq L$. We now bound the errors $\mathcal{E}_1$ and $\mathcal{E}_2$.

Lemma 5. *Let* $k = \frac{4\sqrt{2}\log m}{\epsilon^2(\log\frac{1}{\epsilon}+\log\log m)}$ *and* $0 < \epsilon < \frac{1}{2}$. *Then,* $\mathcal{E}_1 \leq 5\epsilon H$.

Proof. We use (4) to bound $\mathcal{E}_1$. For $l \geq 1$, $h(T_{l-1}) \cdot 2^{l+1} = \left(\frac{T_{l-1}}{m}\log\frac{m}{T_{l-1}}\right) \cdot 2^{l+1} = \left(\frac{\sqrt{2}\hat{m}^{res}(k)}{m\cdot(k\cdot 2^{l-1})}\log\frac{m}{T_l}\right)\cdot 2^{l+1} \leq \frac{4\sqrt{2}m^{res}(k)}{mk}\log m$. Further, since the frequency of a non-top k item is at most $\frac{m}{k}$, we have, $\hat{m}^{res}(k) = \sum_{i\notin\ top(k)} f_i \leq \frac{1}{\log k}\sum_{i\notin top(k)} f_i \cdot \log\frac{m}{f_i} < \frac{mH}{\log k}$. Therefore, $h(T_l)\cdot 2^{l+1} \leq \frac{4\sqrt{2}H\log m}{k\log k}$, for $l \geq 1$. Further, $2h(T_0+\Delta_0) \leq \frac{2(T_0+\Delta_0)}{m}\log\left(\frac{m}{T_0+\Delta_0}\right) \leq \frac{2(1+\frac{\epsilon}{\sqrt{2}})T_0}{m}(\log m) \leq \frac{4m^{res}(k)\log m}{mk} \leq \frac{4H\log m}{k\log k}$, by the argument above. Therefore, the following summation from the expression for $\mathcal{E}_1$ in (4) is,

$$\sum_{i\in G_l, l\geq 1} h(T_{l-1})h(f_i)2^{l+1} + \sum_{i\in lm(G_0)} 2h(T_l+\Delta_0)h(f_i)$$

$$\leq \frac{4\sqrt{2}H\log m}{k\log k}\left(\sum_{i\in G_l, l\geq 1} h(f_i) + \sum_{i\in lm(G_0)} h(f_i)\right) \leq \frac{4\sqrt{2}H\log m}{k\log k}\sum_i h(f_i)$$

$$\leq \frac{4\sqrt{2}H^2\log m}{k\log k} \leq \epsilon^2 H^2$$

by the choice of k as given in the statement. Substituting in (4), and using $L = \log\frac{m}{k}$, we obtain that $\mathcal{E}_1 \leq \frac{\epsilon^2(\log m)H}{m} + 3(\epsilon^2H^2+\epsilon^2H^2)^{1/2} < 5\epsilon H$. □

Lemma 6. *If* $0 < \epsilon \leq 1$ *and* $k \geq \lceil 8e\rceil$, *then* $\mathcal{E}_2 \leq 8\sqrt{2}\epsilon H$.

Proof. Since, $\bar{\epsilon} = \epsilon$, by Lemma 4, $\mathcal{E}_2 \leq 16\epsilon Q_0 \sum_{l=0}^{L}\sum_{i\in G_l}\frac{|h'(\xi_i)|}{2^l}$, with probability $\geq \frac{9}{10} - 2^{-t}$. Since we are using CountMin sketch , for an $i \in G_l$, ξ_i is the value which maximizes $h'()$ in the interval $(f_i, f_i+\Delta_l)$. By Theorem 3, $\hat{m}^{res}(k) < m^{res}(k)$, and therefore, $Q_0 = \frac{\hat{m}^{res}(k)}{k} < \frac{m^{res}(k)}{k}$. Let h_i denote $h(f_i)$, that is, the contribution of i to H. Let θ_i denote the contribution of i to $\mathcal{E}_2$, that is, $\theta_i = \frac{16\epsilon Q_0|h'(\xi_i)|}{2^{l(i)}}$. Thus, $\mathcal{E}_2 = \sum_{i:f_i>0}\theta_i$.

Case 1: $f_i \leq \frac{m}{e} - \Delta_0$. Since $h'()$ is positive and non-increasing in $[1, \frac{m}{e}]$, the value of ξ_i maximizing $|h'()|$ in $[f_i, f_i+\Delta_0]$ is f_i. Therefore $h'(\xi_i) \leq h'(f_i) = \frac{1}{m}\left(\log\frac{m}{f_i} - 1\right) < \frac{1}{m}\log\frac{m}{f_i} = \frac{h_i}{f_i} < \frac{h_i}{T_{l(i)}}$. Therefore, $\theta_i \leq \frac{16\epsilon Q_0 h_i}{2^{l(i)}T_{l(i)}} \leq 8\sqrt{2}\epsilon h_i$, since, $2^{l(i)}T_{l(i)} = T_0$ and $T_0 = Q_0\sqrt{2}$.

Case 2: $\frac{m}{e} - \Delta_0 < f_i \leq \frac{m}{e}$. Since, $k \geq \lceil 8e\rceil$, $T_0 \leq \frac{m}{k} \leq \frac{m}{8e}$ and therefore, $i \in G_0$. In this case, we consider two possibilities. First, if $\hat{f}_i < \frac{m}{e}$, then the value of $\xi_i \in \{f_i, \hat{f}_i\}$ maximizing $h'()$ will be f_i, and the analysis proceeds as in Case 1. The second possibility is: $\hat{f}_i > \frac{m}{e}$. In this case, note that $|h'(f_i - y)| > |h'(f_i + y)|$ for $0 < y < \Delta_0$. Hence, $h'(\xi_i) < h'(f_i - \Delta_0) < \frac{1}{m}\log\frac{m}{f_i-\Delta_0} = \frac{1}{m}(\log\frac{m}{f_i} - \log(1-\frac{\Delta_l}{f_i})) < \frac{h_i}{f_i} + \frac{2\Delta_0}{mf_i} = \frac{h_i}{f_i}(1+\frac{2\Delta_0}{mh_i})$. Since $mh_i = f_i\log\frac{m}{f_i} > f_i$, and $\frac{\Delta_0}{f_i} < \frac{\Delta_0}{T_0} = \frac{\epsilon}{\sqrt{2}}$ we can write $h'(\xi_i) < \frac{h_i}{f_i}(1+\epsilon\sqrt{2}) \leq \frac{h_i(1+\sqrt{2})}{f_i}$, since $\epsilon \leq 1$.

Also, $f_i \geq \frac{m}{e} - \Delta_0 = \frac{m}{e} - \frac{\epsilon T_0}{\sqrt{2}} \geq \frac{m}{e} - T_0$. Since, $T_0 \leq \frac{m}{k} \leq \frac{m}{8e}$, therefore, $f_i \geq 7T_0$. Thus, $h'(\xi_i) < \frac{h_i}{7T_0}(1+\sqrt{2}) \leq \frac{h_i}{2T_0}$. Therefore, $\theta_i \leq 8\epsilon h_i$.

Case 3: $f_i > \frac{m}{e}$. As argued in Case 2, $i \in G_0$. Also $|h'()|$ is increasing in the range $(\frac{m}{e}, m)$. Let $f_i = (1-\alpha_i)m$, where, $0 \leq \alpha_i < 1 - \frac{1}{e}$. Therefore $h_i = (1-\alpha_i)\log\frac{1}{(1-\alpha_i)} > \alpha_i(1-\alpha_i)$. Further, $|h'(\xi_i)| < |h'(f_i + \Delta_0)|$. Let $f_i + \Delta_0 = (1-\alpha_i')m$, that is, $\alpha_i' = \alpha_i - \frac{\Delta_0}{m}$. Then, $|h'((1-\alpha_i')m)| = \frac{1}{m}(1 - \log\frac{1}{1-\alpha_i'}) < \frac{1-\alpha_i'}{m}$. Further, $Q_0 \leq \frac{m^{res}(k)}{\sqrt{2}k} \leq \frac{\alpha_i m}{\sqrt{2}k}$, which gives, $\theta_i = 16\epsilon Q_0 h'(\xi_i) < 16\epsilon\frac{\alpha_i m}{k}\frac{(1-\alpha_i')}{m} = \frac{16\epsilon(1-\alpha_i')\alpha_i}{k} \leq \frac{32\epsilon(1-\alpha_i)\alpha_i}{k}$, since, $\hat{f}_i = (1-\alpha')m \leq f_i + \Delta_0 = f_i(1 + \frac{\Delta_0}{f_i}) = (1-\alpha)m(1 + \frac{\epsilon Q_0}{T_0}) < 2(1-\alpha)m$. Therefore, $\theta_i \leq \frac{32\epsilon h_i}{k} \leq 2\epsilon h_i$, for the given k.

In all cases, $\theta_i \leq 8\sqrt{2}\epsilon h_i$. Therefore, $\mathcal{E}_2 = \sum_{i:f_i>0}\theta_i \leq 8\sqrt{2}\epsilon\sum_{i:f_i>0} h_i = 8\sqrt{2}\epsilon H$. □

We can now prove the main theorem.

Proof. [Of Theorem 1] The estimation error is bounded by $\mathcal{E}_1 + \mathcal{E}_2$. By Lemmas 5 and 6, the total error is $(5 + 8\sqrt{2})\epsilon H \leq 17\epsilon H$. Replacing ϵ by $\frac{\epsilon}{17}$ and returning the median $\hat{H}^{\text{med}}$ of $O(\log\frac{1}{\delta})$ independent estimates gives $|\hat{H}^{\text{med}} - H| \leq \epsilon H$ with probability $1-\delta$.

The space used by the COUNT-MIN sketch sub-structure at each level of the HSS structure is $O(\frac{k}{\epsilon}(\log m + \log\frac{1}{\epsilon})(\log m))$ bits [4]. The number of levels is $L = \log\frac{m}{k} = O(\log m)$. The use of the pseudo-random generator contributes an additional factor of $\log m$ to the space requirement. A collection of $O(\log\frac{1}{\delta})$ copies are kept to return the median estimate. Therefore, the total space requirement is $O\big(\frac{k}{\epsilon}(\log m + \log\frac{1}{\epsilon})(\log^3 m)(\log\frac{1}{\delta})\big)$, which, for the chosen value of k, becomes $O\left(\frac{(\log^4 m)}{\epsilon^3}\frac{(\log m + \log\frac{1}{\epsilon})}{(\log\frac{1}{\epsilon} + \log\log m)}(\log\frac{1}{\delta})\right)$ bits. □

Generalizing to streams with negative frequencies. We briefly outline how the algorithm can be applied to the general update streaming model. First, the COUNTSKETCH algorithm for finding frequent items is used instead of COUNT-MIN algorithm. The role of $m^{res}(k)$ is replaced by $F_2^{res}(k)$; otherwise, the algorithm and its analysis is quite similar. The space complexity of the algorithm is polynomial in $\frac{1}{\epsilon}$ and $(\log F_2 + \log N)$.

References

1. B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. "Models and Issues in Data Stream Systems". In *Proc. of ACM PODS*, 2002.
2. L. Bhuvanagiri, S. Ganguly, D. Kesh, and C. Saha. "Simpler algorithm for estimating frequency moments of data streams". In *Proc. of ACM SODA*, 2006.

[4] For efficient retrieval of frequent items, a COUNT-MIN structure can be kept for each of the $\log m$ dyadic levels; this would obviate the need for a sequential scan of the domain, at the expense of an additional factor of $O(\log m)$ space.

3. A. Chakrabarti, D.K. Ba, and S. Muthukrishnan. "Estimating Entropy and Entropy Norm on Data Streams". In *Proc. of STACS*, 2006.
4. M. Charikar, K. Chen, and M. Farach-Colton. "Finding frequent items in data streams". In *Proc. of ICALP*, 2002.
5. G. Cormode and S. Muthukrishnan. "What's Hot and What's Not: Tracking Most Frequent Items Dynamically". In *Proc. of ACM PODS*, May 2003.
6. G. Cormode and S. Muthukrishnan. "An improved data stream summary: The Count-Min sketch and its applications". In *Proc. of LATIN, Springer LNCS Vol. 2976*, 2004.
7. S. Ganguly, D. Kesh, and C. Saha. "Practical Algorithms for Tracking Database Join Sizes". In *Proc. of FSTTCS, Springer LNCS Vol. 3821*, 2005.
8. Y. Gu, A. McCallum, and D. Towsley. "Detecting Anomalies in Network Traffic Using Maximum Entropy Estimation". In *Proc. of Internet Measurement Conference*, 2005.
9. S. Guha, A. McGregor, and S. Venkatsubramanian. "Streaming and Sublinear Approximation of Entropy and Information Distances". In *Proc. of SODA*, 2006.
10. P. Indyk. "Stable Distributions, Pseudo Random Generators, Embeddings and Data Stream Computation". In *Proc. of IEEE FOCS*, 2000.
11. P. Indyk and D. Woodruff. "Optimal Approximations of the Frequency Moments". In *Proc. of ACM STOC*, 2005.
12. S. Muthukrishnan. *"Data Streams: Algorithms and Applications"*. Foundations and Trends in Theoretical Computer Science, *Vol. 1, Issue 2*, 2005.
13. N. Nisan. "Pseudo-Random Generators for Space Bounded Computation". In *Proc. of ACM STOC*, 1990.
14. A. Wagner and B Plattner. "Entropy based worm and anomaly detection in fast IP networks". In *14th IEEE WET ICE, STCA Security Workshop*, 2005.
15. K. Xu, Z. Zhang, and S. Bhattacharyya. "Profiling internet backbone traffic: behavior models and applications". *SIGCOMM Comput. Commun. Rev.*, 35(4), 2005.

A Proof of Lemma 3

Proof.

$$
\begin{aligned}
\mathsf{E}\big[\bar{\Psi}\big]^2 &= \mathsf{E}\big[\big(\sum_i \psi(f_i) \sum_{r=0}^{L} x_{i,r} \cdot 2^r\big)^2\big] \\
&= \mathsf{E}\big[\sum_i \psi^2(f_i)\big(\sum_{r=0}^{L} x_{i,r} \cdot 2^r\big)^2 + \sum_{i \neq j} \psi(f_i) \cdot \psi(f_j) \sum_{r_1=0}^{L} x_{i,r_1} \cdot 2^{r_1} \sum_{r_2=0}^{L} x_{i,r_2} \cdot 2^{r_2}\big] \\
&= \mathsf{E}\big[\sum_i \psi^2(f_i)\big(\sum_{r=0}^{L} x_{i,r} \cdot 2^r\big)^2\big] + \mathsf{E}\big[\sum_{i \neq j} \psi(f_i) \cdot \psi(f_j) \sum_{r_1=0}^{L} x_{i,r_1} \cdot 2^{r_1} \sum_{r_2=0}^{L} x_{j,r_2} \cdot 2^{r_2}\big] \\
&= \mathsf{E}\big[\sum_i \psi^2(f_i) \sum_{r=0}^{L} x_{i,r}^2 \cdot 2^{2r}\big] + \mathsf{E}\big[\sum_i \psi^2(f_i) \sum_{r_1 \neq r_2} x_{i,r_1} \cdot x_{i,r_2} \cdot 2^{r_1 + r_2}\big] \\
&\qquad + \mathsf{E}\big[\sum_{i \neq j} \psi(f_i) \cdot \psi(f_j) \sum_{r_1=0}^{L} x_{i,r_1} \cdot 2^{r_1} \sum_{r_2=0}^{L} x_{j,r_2} \cdot 2^{r_2}\big]
\end{aligned}
$$

We note that: (a) $x_{i,r}^2 = x_{i,r}$. (b) an item i is classified into a unique group G_r, and therefore, $x_{i,r_1} \cdot x_{i,r_2} = 0$, for $r_1 \neq r_2$, and, (c) for $i \neq j$, x_{i,r_1} and x_{j,r_2} are independent of each other, regardless of the values of r_1 and r_2. Thus,

$$\mathsf{E}\big[\bar{\Psi}\big]^2 = \sum_i \mathsf{E}\big[\psi^2(f_i) \sum_{r=0}^{L} x_{i,r} \cdot 2^{2r}\big] + \sum_{i \neq j} \mathsf{E}\big[\psi(f_i) \sum_{r_1=0}^{L} x_{i,r_1} \cdot 2^{r_1}\big] \mathsf{E}\big[\psi(f_j) \sum_{r_2=0}^{L} x_{j,r_2} \cdot 2^{r_2}\big]$$

As a result, the expression for $\mathsf{Var}\big[\bar{\Psi}\big]$ simplifies to

$$\mathsf{Var}\big[\bar{\Psi}\big] = \mathsf{E}\big[\bar{\Psi}^2\big] - \mathsf{E}\big[\bar{\Psi}\big]^2 = \sum_i \mathsf{E}\big[\psi^2(f_i) \sum_{r=0}^{L} x_{i,r} \cdot 2^{2r}\big] - \sum_i \mathsf{E}\big[\psi(f_i) \sum_{r=0}^{L} x_{i,r} \cdot 2^{r}\big]^2$$

$\mathsf{E}\big[\psi(f_i) \sum_{r=0}^{L} x_{i,r} \cdot 2^r\big]$ is given by Lemma 2. $\mathsf{E}\big[\psi^2(f_i) \sum_{r=0}^{L} x_{i,r} \cdot 2^{2r}\big]$ is calculated in an almost similar manner; we briefly outline the calculation. Let $i \in G_l$. We decompose groups into the left-region (lr), middle-region (mr) and right regions (rr) as in Lemma 2.

Suppose $i \in G_0 \setminus lr(G_0)$: Then, $1 - 2^{-t} < \mathsf{Pr}\{\hat{f}_i > T_0\} \leq 1$ and probability of i being classified into any other $G_r, r \neq 0$ is at most $2^{-t} \cdot \frac{1}{2^r}$. Therefore, $\sum_r \mathsf{E}\big[x_{i,r} 2^{2r}\big] < 1 + \sum_{r>0} 2^{-t} \cdot 2^r < 1 + 2^{-t+L+1}$.

Suppose $i \in lr(G_0)$: In this case, i can get classified into either $\bar{G}_0$ or $\bar{G}_1$, with probability at least $1 - 2^{-t}$. Given that i is classified into one of $\bar{G}_0$ or $\bar{G}_1$, let p be the conditional probability that $\mathsf{Pr}\{i \in \bar{G}_0 \mid i \in \bar{G}_0 \cup \bar{G}_1\}$. Therefore $\mathsf{Pr}\{x_{i,0}\} \leq p$, $\mathsf{Pr}\{x_{i,1}\} \leq \frac{(1-p)}{2}$ and $\mathsf{Pr}\{x_{i,r}\} \leq 2^{-t}$ for $r \notin \{0, 1\}$. Therefore, $\sum_r \mathsf{E}\big[x_{i,r} 2^{2r}\big] < p + \frac{1-p}{2} \cdot 2^2 + \sum_{r>1} 2^{-t} \cdot 2^r < 2 + 2^{-t+L+1}$.

Suppose $i \in lr(G_l)$ for $l > 1$: The analysis for this case is similar to that of $i \in lr(G_0)$, except that $\mathsf{Pr}\{x_{i,l}\} \leq \frac{p}{2^l}$, $\mathsf{Pr}\{x_{i,l+1}\} \leq \frac{1-p}{2^{l+1}}$. Therefore, $\sum_r \mathsf{E}\big[x_{i,r} 2^{2r}\big] < \frac{p}{2^l} \cdot 2^{2l} + \frac{1-p}{2^{l+1}} \cdot 2^{2(l+1)} + \sum_{r \notin \{l, l+1\}} 2^{-t} \cdot 2^r < 2^{l+1} + 2^{-t+L+1}$.

Suppose $i \in mr(G_l)$ for $l > 1$: Such elements will be classified into $\bar{G}_l$ with probability $\geq 1 - 2^{-t}$, resulting in $\sum_r \mathsf{E}\big[x_{i,r} 2^{2r}\big] < \frac{1}{2^l} \cdot 2^{2l} + \sum_{r \neq l} 2^{-t} \cdot 2^r < 2^l + 2^{-t+L+1}$.

Suppose $i \in rr(G_l)$ for $l > 1$: Using an argument similar to that for $ll(G_l)$, we get $\sum_i \mathsf{E}\big[x_{i,r} 2^{2r}\big] < 2^l + 2^{-t+L+1}$. Combining the above cases, we obtain

$$\sum_i \mathsf{E}\big[\psi^2(f_i) \sum_{r=0}^{L} x_{i,r} \cdot 2^{2r}\big] \leq \sum_{i \in G_0 \setminus lr(G_0)} (1 + 2^{-t+L+1}) \cdot \psi^2(f_i) + \sum_{i \notin G_0 \setminus lr(G_0)} \psi^2(f_i) \cdot (2^{l+1} + 2^{-t+L+1}) \ .$$

In conjunction with Lemma 2, we get

$$\mathsf{Var}\big[\bar{\Psi}\big] \leq \sum_{i \in G_0 \setminus lr(G_0)} (1 + 2^{-t+L+1}) \cdot \psi^2(f_i) + \sum_{i \notin G_0 \setminus lr(G_0)} \psi^2(f_i) \cdot (2^{l+1} + 2^{-t+L+1}) - \sum_i (1 - 2^{-t+\log L}) \cdot \psi^2(f_i) \leq \sum_{i \in \mathcal{S}} 2^{-t+L+2} \cdot \psi^2(f_i) + \sum_{i \notin G_0 \setminus lr(G_0)} \psi^2(f_i) \cdot 2^{l+1}.$$

□

Necklaces, Convolutions, and $X + Y$

David Bremner[1,⋆], Timothy M. Chan[2,⋆], Erik D. Demaine[3,⋆⋆], Jeff Erickson[4], Ferran Hurtado[5], John Iacono[6,⋆⋆], Stefan Langerman[7], and Perouz Taslakian[8]

[1] Faculty of Computer Science, University of New Brunswick, Fredericton, New Brunswick, Canada
bremner@unb.ca
[2] School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada
tmchan@uwaterloo.ca
[3] Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, USA
edemaine@mit.edu
[4] Computer Science Department, University of Illinois, Urbana-Champaign, IL, USA
jeffe@cs.uiuc.edu
[5] Departament de Matemàtica Aplicada II, Universitat Politècnica de Catalunya, Barcelona, Spain
Ferran.Hurtado@upc.es
[6] Department of Computer and Information Science, Polytechnic University, Brooklyn, NY, USA
http://john.poly.edu
[7] Chercheur qualifié du FNRS, Départment d'Informatique, Université Libre de Bruxelles, Brussels, Belgium
stefan.langerman@ulb.ac.be
[8] School of Computer Science, McGill University, Montréal, Québec, Canada
perouz@cs.mcgill.ca

Abstract. We give subquadratic algorithms that, given two necklaces each with n beads at arbitrary positions, compute the optimal rotation of the necklaces to best align the beads. Here alignment is measured according to the ℓ_p norm of the vector of distances between pairs of beads from opposite necklaces in the best perfect matching. We show surprisingly different results for $p = 1$, $p = 2$, and $p = \infty$. For $p = 2$, we reduce the problem to standard convolution, while for $p = \infty$ and $p = 1$, we reduce the problem to $(\min, +)$ convolution and $(\text{median}, +)$ convolution. Then we solve the latter two convolution problems in subquadratic time, which are interesting results in their own right. These results shed some light on the classic sorting $X + Y$ problem, because the convolutions can be viewed as computing order statistics on the antidiagonals of the $X + Y$ matrix. All of our algorithms run in $o(n^2)$ time, whereas the obvious algorithms for these problems run in $\Theta(n^2)$ time.

⋆ Supported by NSERC.
⋆⋆ Supported in part by NSF grants CCF-0430849 and OISE-0334653 and by an Alfred P. Sloan Fellowship.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 160–171, 2006.

1 Introduction

How should we rotate two necklaces, each with n beads at different locations, to best align the beads? More precisely, each necklace is represented by a set of n points on the unit-circumference circle, and the goal is to find rotations of the necklaces, and a perfect matching between the beads of the two necklaces, that minimizes some norm of the circular distances between matched beads. In particular, the ℓ_1 norm minimizes the average absolute circular distance between matched beads, the ℓ_2 norm minimizes the average squared circular distance between matched beads, and the ℓ_∞ norm minimizes the maximum circular distance between matched beads. The ℓ_1 version of this necklace alignment problem was introduced by Toussaint [29] in the context of comparing rhythms in computational music theory, with possible applications to rhythm phylogeny [14, 30].

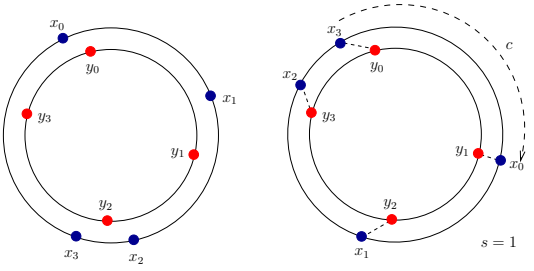


Fig. 1. An example of necklace alignment: the input (left) and one possible output (right)

Toussaint [29] gave a simple $O(n^2)$-time algorithm for ℓ_1 necklace alignment, and asked whether the problem could be solved in $o(n^2)$ time. In this paper, we solve this open problem by giving $o(n^2)$-time algorithms for ℓ_1, ℓ_2, and ℓ_∞ necklace alignment, in both the standard real RAM model of computation and the less realistic nonuniform linear decision tree model of computation.

Our approach is based on reducing the necklace alignment problem to another important problem, convolution, for which we also obtain improved algorithms. The $(+, \cdot)$ convolution of two vectors $\mathbf{x} = \langle x_0, x_1, \ldots, x_{n-1}\rangle$ and $\mathbf{y} = \langle y_0, y_1, \ldots, y_{n-1}\rangle$, is the vector $\mathbf{x} * \mathbf{y} = \langle z_0, z_1, \ldots, z_{n-1}\rangle$ where $z_k = \sum_{i=0}^{k} x_i \cdot y_{k-i}$. While any $(\oplus, \odot)$ convolution with specified addition and multiplication operators (here denoted $\mathbf{x} \overset{\odot}{\underset{\oplus}{*}} \mathbf{y}$) can be computed in $O(n^2)$ time, the $(+, \cdot)$ convolution can be computed in $O(n \lg n)$ time using the Fast Fourier Transform [10, 21, 22], because the Fourier transform converts convolution into elementwise multiplication. Indeed, fast $(+, \cdot)$ convolution was one of the early breakthroughs in algorithms, with applications to polynomial and integer multiplication [3], batch polynomial evaluation [11, Problem 30-5], 3SUM [15, 1], string matching [17, 23, 9], matrix multiplication [7], and even juggling [5].

As we show in Theorems 1, 3, and 11, respectively, ℓ_2 necklace alignment reduces to standard $(+, \cdot)$ convolution, ℓ_∞ necklace alignment reduces to $(\min, +)$ [and $(\max, +)$] convolution, and ℓ_1 necklace alignment reduces to $(\text{median}, +)$ convolution (whose kth entry is $\text{median}_{i=0}^{k}(x_i + y_{k-i})$). The $(\min, +)$ convolution problem has appeared frequently in the literature, already

appearing in Bellman's early work on dynamic programming in the early 1960s [2, 16, 24, 25, 26, 28]. Its name varies among "minimum convolution", "min-sum convolution", "inf-convolution", "infimal convolution", and "epigraphical sum".[1] To date, however, no worst-case $o(n^2)$-time algorithms for this convolution, or the more complex (median, +) convolution, has been obtained. In this paper, we develop worst-case $o(n^2)$-time algorithms for (min, +) and (median, +) convolution, in the real RAM and the nonuniform linear decision tree.

Necklace alignment problem. More formally, in the *necklace alignment problem*, the input is a number p representing the ℓ_p norm, and two sorted vectors of n real numbers, $\mathbf{x} = \langle x_0, x_1, \ldots, x_{n-1} \rangle$ and $\mathbf{y} = \langle y_0, y_1, \ldots, y_{n-1} \rangle$, representing the two necklaces. See Figure 1. Canonically, we assume that each number x_i and y_i is in the range $[0, 1)$, representing a point on the unit-circumference circle (parameterized clockwise from some fixed point).

The optimization problem involves two parameters. The first parameter, the *offset* $c \in [0, 1)$, is the clockwise rotation angle of the first necklace relative to the second necklace. The second parameter, the *shift* $s \in \{0, 1, \ldots, n\}$, defines the perfect matching between beads: bead i of the first necklace matches with bead $(i + s) \bmod n$ of the second necklace. (Here we use the property that an optimal perfect matching between the beads does not cross itself.)

The goal of the ℓ_p necklace alignment problem is to find the offset $c \in [0, 1)$ and the shift $s \in \{0, 1, \ldots, n\}$ that minimize $\sum_{i=0}^{n-1} \left| x_i - y_{(i+s) \bmod n} + c \right|^p$ or, in the case $p = \infty$, that minimize $\max_{i=0}^{n-1} \left| x_i - y_{(i+s) \bmod n} + c \right|$.

Although not obvious from the definition, the ℓ_1, ℓ_2, and ℓ_∞ necklace alignment problems all have trivial $O(n^2)$ solutions. In each case, as we show, the optimal offset c can be computed in linear time for a given shift value s (sometimes even independent of s). The optimization problem is thus effectively over just $s \in \{0, 1, \ldots, n\}$, and the objective costs $O(n)$ time to compute for each s, giving an $O(n^2)$-time algorithm.

Related work. Although necklaces are studied throughout mathematics, mainly in combinatorial settings, we are not aware of any work on the necklace alignment problem before Toussaint [29]. He introduced ℓ_1 necklace alignment, calling it the *cyclic swap-distance* or *necklace swap-distance* problem, with a restriction that the beads lie at integer coordinates. Colannino et al. [8] consider some different distance measures between two sets of points on the real line in which the matching does not have to match every point. They do not, however, consider alignment under such distance measures.

The only subquadratic results for (min, +) convolution concern two special cases. First, the (min, +) convolution of two convex sequences or functions can be trivially computed in $O(n)$ time by a simple merge, which is the same as computing the Minkowski sum of two convex polygons [26]. This special case is already used in image processing and computer vision [16, 24]. Second, Bussieck

[1] "Tropical convolution" would also make sense, by direct analogy with tropical geometry, but we have never seen this terminology used in print.

et al. [4] proved that the $(\min, +)$ convolution of two *randomly permuted* sequences can be computed in $O(n \lg n)$ expected time. Our results are the first to improve the worst-case running time for $(\min, +)$ convolution.

Connections to $X + Y$. The necklace alignment problems, and their corresponding convolution problems, are also intrinsically connected to problems on $X + Y$ matrices. Given two lists of n numbers, $X = \langle x_0, x_1, \ldots, x_{n-1} \rangle$ and $Y = \langle y_0, y_1, \ldots, y_{n-1} \rangle$, $X + Y$ is the matrix of all pairwise sums, whose (i, j)th entry is $x_i + y_j$. A classic unsolved problem [12] is whether the entries of $X+Y$ can be sorted in $o(n^2 \lg n)$ time. Fredman [19] showed that $O(n^2)$ comparisons suffice in the nonuniform linear decision tree model, but it remains open whether this can be converted into an $O(n^2)$-time algorithm in the real RAM model. Steiger and Streinu [27] gave a simple algorithm that takes $O(n^2 \log n)$ time while using only $O(n^2)$ comparisons.

The $(\min, +)$ convolution is equivalent to finding the minimum element in each antidiagonal of the $X + Y$ matrix, and similarly the $(\max, +)$ convolution finds the maximum element in each antidiagonal. We show that ℓ_∞ necklace alignment is equivalent to finding the antidiagonal of $X + Y$ with the smallest *range* (the maximum element minus the minimum element). The $(\text{median}, +)$ convolution is equivalent to finding the median element in each antidiagonal of the $X + Y$ matrix. We show that ℓ_1 necklace alignment is equivalent to finding the antidiagonal of $X+Y$ with the smallest *median cost* (the total distance between each element and the median of the elements). Given the apparent difficulty in sorting $X+Y$, it seems natural to believe that the minimum, maximum, and median elements of every antidiagonal cannot be found, and that the corresponding objectives cannot be minimized, any faster than $O(n^2)$ total time. Figure 2 shows a sample $X+Y$ matrix with the maximum element in each antidiagonal marked, with no apparent structure. Nonetheless, we show that $o(n^2)$ algorithms are possible.

Our results. In the standard real RAM model, we give subquadratic algorithms for the ℓ_1, ℓ_2, and ℓ_∞ necklace alignment problems, and for the $(\min, +)$ and $(\text{median}, +)$ convolution problems, using techniques of Chan [6]. Despite the roughly logarithmic factor improvements for ℓ_1 and ℓ_∞, these results do not use word-level bit tricks of word-RAM fame.

1. $O(n \lg n)$-time algorithm on the real RAM for ℓ_2 necklace alignment (Section 2).
2. $O(n^2 / \lg n)$-time algorithm on the real RAM for ℓ_∞ necklace alignment and $(\min, +)$ convolution (Section 3).
3. $O(n^2 (\lg \lg n)^2 / \lg n)$-time algorithm on the real RAM for ℓ_1 necklace alignment and $(\text{median}, +)$ convolution (Section 4).

In the nonuniform linear decision tree model, we give faster algorithms for the ℓ_1 and ℓ_∞ necklace alignment problems, using techniques of Fredman [19, 20]:

4. $O(n\sqrt{n})$-time algorithm in the nonuniform linear decision tree model for ℓ_∞ necklace alignment and $(\min, +)$ convolution (Section 3).
5. $O(n\sqrt{n \lg n})$-time algorithm in the nonuniform linear decision tree model for ℓ_1 necklace alignment and $(\text{median}, +)$ convolution (Section 4).

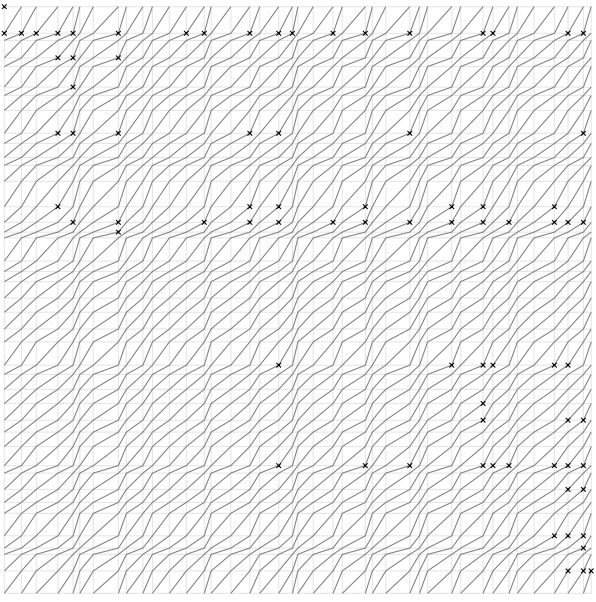


Fig. 2. An $X + Y$ matrix. Each polygonal line denotes an antidiagonal of the matrix, with a point at coordinates (x, y) denoting the value $x + y$ for $x \in X$ and $y \in Y$. An $\times$ denotes the maximum element in each antidiagonal.

(Although we state our results here in terms of $(\min, +)$ and $(\text{median}, +)$ convolution, our results discuss $-$ instead of $+$ for synergy with necklace alignment.)

2 ℓ_2 Necklace Alignment and $(+, \cdot)$ Convolution

In this section, we show how ℓ_2 necklace alignment reduces to standard convolution, leading to an $O(n \lg n)$-time algorithm.

Theorem 1. *The ℓ_2 necklace alignment problem can be solved in $O(n \lg n)$ time on a real RAM.*

Proof. The objective $\sum_{i=0}^{n-1} \left(x_i - y_{(i+s) \bmod n} + c\right)^2$ expands algebraically to

$$\sum_{i=0}^{n-1} \left(x_i^2 + 2cx_i + c^2\right) + \sum_{i=0}^{n-1} \left(y_{(i+s) \bmod n}^2 - 2cy_{(i+s) \bmod n}\right) - 2\sum_{i=0}^{n-1} x_i y_{(i+s) \bmod n}$$
$$= \left[\sum_{i=0}^{n-1} \left(x_i^2 + y_i^2\right) + 2c\sum_{i=0}^{n-1} (x_i - y_i) + nc^2\right] - 2\sum_{i=0}^{n-1} x_i y_{(i+s) \bmod n}.$$

The first term depends solely on the inputs and the variable c, while the second term depends solely on the inputs and the variable s. Thus the two terms can be optimized separately. The first term can be optimized in $O(n)$ time by solving

for when the derivative, which is linear in c, is zero. The second term can be computed, for each $s \in \{0, 1, \ldots, n-1\}$, in $O(n \lg n)$ time using $(+, \cdot)$ convolution (and therefore optimized in the same time). Specifically, define the vectors

$$\mathbf{x}' = \langle x_0, x_1, \ldots, x_{n-1}; \underbrace{0, 0, \ldots, 0}_{n} \rangle; \quad \mathbf{y}' = \langle y_{n-1}, y_{n-2}, \ldots, y_0; y_{n-1}, y_{n-2}, \ldots, y_0 \rangle.$$

Then, for $s' \in \{0, 1, \ldots, n-1\}$, the $(n + s')$th entry of the convolution $\mathbf{x}' * \mathbf{y}'$ is $\sum_{i=0}^{n+s'} x'_i y'_{n+s'-i} = \sum_{i=0}^{n-1} x_i y_{(i-s'-1) \bmod n}$, which is the desired entry if we let $s' = n - 1 - s$. We can compute the entire convolution in $O(n \lg n)$ time using the Fast Fourier Transform. □

3 ℓ_∞ Necklace Alignment and $(\min, +)$ Convolution

First we show the relation between ℓ_∞ necklace alignment and $(\min, +)$ convolution. We need the following basic fact:

Fact 2. *For any vector* $\mathbf{z} = \langle z_0, z_1, \ldots, z_{n-1} \rangle$*, the minimum value of* $\max_{i=0}^{n-1} |z_i + c|$ *is* $\frac{1}{2}\left(\max_{i=0}^{n-1} z_i - \min_{i=0}^{n-1} z_i\right)$*, which is achieved when* $c = -\frac{1}{2}\left(\min_{i=0}^{n-1} z_i + \max_{i=0}^{n-1} z_i\right)$.

Instead of using $(\min, +)$ convolution directly, we use two equivalent forms, $(\min, -)$ and $(\max, -)$ convolution:

Theorem 3. *The* ℓ_∞ *necklace alignment problem can be reduced in* $O(n)$ *time to one* $(\min, -)$ *convolution and one* $(\max, -)$ *convolution.*

Proof. For two necklaces $\mathbf{x}$ and $\mathbf{y}$, we apply the $(\min, -)$ convolution to the following vectors:

$$\mathbf{x}' = \langle x_0, x_1, \ldots, x_{n-1}; \underbrace{\infty, \ldots, \infty}_{n} \rangle; \quad \mathbf{y}' = \langle y_{n-1}, y_{n-2}, \ldots, y_0; y_{n-1}, y_{n-2}, \ldots, y_0 \rangle.$$

Then, for $s' \in \{0, 1, \ldots, n-1\}$, the $(n + s')$th entry of $\mathbf{x}' \underset{\min}{\bar{*}} \mathbf{y}'$ is $\min_{i=0}^{n+s'}(x'_i - y'_{n+s'-i}) = \min_{i=0}^{n-1}(x_i - y_{(i-s'-1) \bmod n})$, which is $\min_{i=0}^{n-1}(x_i - y_{(i+s) \bmod n})$ if we let $s' = n - 1 - s$. By symmetry, we can compute the $(\max, -)$ convolution $\mathbf{x}'' \underset{\max}{\bar{*}} \mathbf{y}'$, where $\mathbf{x}''$ has $-\infty$'s in place of ∞'s, and use it to compute $\max_{i=0}^{n-1}(x_i - y_{(i+s) \bmod n})$ for each $s \in \{0, 1, \ldots, n-1\}$. Applying Fact 2, we can therefore minimize $\max_{i=0}^{n-1} |x_i - y_{(i+s) \bmod n} + c|$ over c, for each $s \in \{0, 1, \ldots, n-1\}$. By brute force, we can minimize over s as well using $O(n)$ additional comparisons and time. □

For our nonuniform linear decision tree results, we use the main theorem of Fredman's work on sorting $X + Y$:

Theorem 4. [19] *For any fixed set* Γ *of permutations of* N *elements, there is a comparison tree of depth* $O(N + \lg |\Gamma|)$ *that sorts any sequence whose rank permutation belongs to* Γ.

Theorem 5. *The* $(\min, -)$ *convolution of two vectors of length* n *can be computed in* $O(n\sqrt{n})$ *time in the nonuniform linear decision tree model.*

Proof. Let $\mathbf{x}$ and $\mathbf{y}$ denote the two vectors of length n, and let $\mathbf{x} \underset{\min}{\bar{*}} \mathbf{y}$ denote their $(\min, -)$ convolution, whose kth entry is $\min_{i=0}^{k}(x_i - y_{k-i})$.

First we sort the set $D = \{x_i - x_j, y_i - y_j : |i-j| \leq d\}$ of pairwise differences between nearby x_i's and nearby y_i's, where $d \leq n$ is a value to be determined later. This set D has $N = O(nd)$ elements. The possible sorted orders of D correspond to cells in the arrangement of hyperplanes in $\mathbb{R}^{2n}$ induced by all $\binom{N}{2}$ possible comparisons between elements in the set, and this hyperplane arrangement has $O(N^{4n})$ cells. By Theorem 4, there is a comparison tree sorting D of depth $O(N + n \lg N) = O(nd + n \lg n)$.

The comparisons we make to sort D enable us to compare $x_i - y_{k-i}$ versus $x_j - y_{k-j}$ for free, provided $|i-j| \leq d$, because $x_i - y_{k-i} < x_j - y_{k-j}$ precisely if $x_i - x_j < y_{k-i} - y_{k-j}$. Thus, in particular, we can compute $M_k(\lambda) = \min\{x_i - y_{k-i} : i = \lambda, \lambda+1, \ldots, \min\{\lambda + d, n\} - 1\}$ for free (using the outcomes of the comparisons we have already made).

We can rewrite the kth entry $\min_{i=0}^{k}(x_i - y_{k-i})$ of $\mathbf{x} \underset{\min}{\bar{*}} \mathbf{y}$ as $\min\{M_k(0), M_k(d), M_k(2d), \ldots, M_k(\lceil k/d \rceil d)\}$, and thus we can compute it in $O(k/d) = O(n/d)$ comparisons between differences. Therefore all n entries can be computed in $O(nd + n \lg n + n^2/d)$ total time.

This asymptotic running time is minimized when $nd = \Theta(n^2/d)$, i.e., when $d^2 = \Theta(n)$. Substituting $d = \sqrt{n}$, we obtain a running time of $O(n\sqrt{n})$ in the nonuniform linear decision tree model. □

Combining Theorems 3 and 5, we obtain the following result:

Corollary 6. *The* ℓ_∞ *necklace alignment problem can be solved in* $O(n\sqrt{n})$ *time in the nonuniform linear decision tree model.*

Our results on the real RAM use the following geometric lemma from Chan's work on all-pairs shortest paths:

Lemma 7. [6, Lemma 2.1] *Given* n *points* $p_1, p_2, \ldots, p_n$ *in* d *dimensions, each colored either red or blue, we can find the* P *pairs* (p_i, p_j) *for which* p_i *is red,* p_j *is blue, and* p_i *dominates* p_j *(i.e., for all* k*, the* k*th coordinate of* p_i *is at least the* k*th coordinate of* p_j*), in* $2^{O(d)} n^{1+\varepsilon} + O(P)$ *time for arbitrarily small* $\varepsilon > 0$.

Theorem 8. *The* $(\min, -)$ *convolution of two vectors of length* n *can be computed in* $O(n^2/\lg n)$ *time on a real RAM.*

Proof. Let $\mathbf{x}$ and $\mathbf{y}$ denote the two vectors of length n, and let $\mathbf{x} \underset{\max}{\bar{*}} \mathbf{y}$ denote their $(\max, -)$ convolution. (Symmetrically, we can compute the $(\min, -)$ convolution.) For each $\delta \in \{0, 1, \ldots, d-1\}$, for each $i \in \{0, d, 2d, \ldots, \lfloor n/d \rfloor d\}$, and for each $j \in \{0, 1, \ldots, n-1\}$, we define the d-dimensional points

$$p_{\delta,i} = (x_{i+\delta} - x_i, x_{i+\delta} - x_{i+1}, \ldots, x_{i+\delta} - x_{i+d-1}),$$
$$q_{\delta,j} = (y_{j-\delta} - y_i, y_{j-\delta} - y_{i-1}, \ldots, y_{j-\delta} - y_{j-d-1}).$$

(To handle boundary cases, define $x_i = \infty$ and $y_j = -\infty$ for indices i, j outside $[0, n-1]$.) For each $\delta \in \{0, 1, \ldots, d-1\}$, we apply Lemma 7 to the set of red points $\{p_{\delta,i} : i = 0, d, 2d, \ldots, \lfloor n/d \rfloor d\}$ and the set of blue points $\{q_{\delta,j} : j = 0, 1, \ldots, n-1\}$, to obtain all dominating pairs $(p_{\delta,i}, q_{\delta,j})$.

Point $p_{\delta,i}$ dominates $q_{\delta,j}$ precisely if $x_{i+\delta} - x_{i+\delta'} \geq y_{j-\delta} - y_{j-\delta'}$ for all $\delta' \in \{0, 1, \ldots, d-1\}$ (ignoring the indices outside $[0, n-1]$). By re-arranging terms, this condition is equivalent to $x_{i+\delta} - y_{j-\delta} \geq x_{i+\delta'} - y_{j-\delta'}$ for all $\delta' \in \{0, 1, \ldots, d-1\}$. If we substitute $j = k - i$, we obtain that $(p_{\delta,i}, q_{\delta,k-i})$ is a dominating pair precisely if $x_{i+\delta} - y_{k-i-\delta} = \max_{\delta'=1}^{d-1}(x_{i+\delta'} - y_{k-i-\delta'})$. Thus, the set of dominating pairs gives us the maximum $M_k(i) = \max\{x_i - y_{k-i}, x_{i+1} - y_{k-i+1}, \ldots, x_{\min\{i+d,n\}-1} - y_{\min\{k-i+d,n\}-1}\}$ for each i divisible by d and for each k. Also, there can be at most $O(n^2/d)$ such pairs for all i, j, δ, because there are $O(n/d)$ choices for i and $O(n)$ choices for j, and if $(p_{\delta,i}, q_{\delta,j})$ is a dominating pair, then $(p_{\delta',i}, q_{\delta',j})$ cannot be a dominating pair for any $\delta' \neq \delta$. (Here we assume that the max is achieved uniquely, which can be arranged by standard perturbation techniques or by breaking ties consistently [6].) Hence, the running time of the d executions of Lemma 7 is $d2^{O(d)}n^{1+\varepsilon} + O(n^2/d)$ time, which is $O(n^2/\lg n)$ if we choose $d = \alpha \lg n$ for a sufficiently small constant $\alpha > 0$. We can rewrite the kth entry $\max_{i=0}^{k}(x_i - y_{k-i})$ of $\mathbf{x} \underset{\max}{\bar{*}} \mathbf{y}$ as $\max\{M_k(0), M_k(d), M_k(2d), \ldots, M_k(\lceil k/d \rceil d)\}$, and thus we can compute it in $O(k/d) = O(n/d)$ time. Thus all n entries can be computed in $O(n^2/d) = O(n^2/\lg n)$ time on a real RAM. □

Combining Theorems 3 and 8, we obtain the following result:

Corollary 9. *The ℓ_∞ necklace alignment problem can be solved in $O(n^2/\lg n)$ time on a real RAM.*

This approach likely cannot be improved beyond $O(n^2/\lg n)$. Such an improvement would require an improvement to Lemma 7, which would in turn improve the fastest known algorithm for all-pairs shortest paths in dense graphs, the $O(n^3/\lg n)$-time algorithm of [6].

4 ℓ_1 Necklace Alignment and (median, +) Convolution

First we show the relation between ℓ_1 necklace alignment and (median, +) convolution. We need the following basic fact:

Fact 10. *For any vector $\mathbf{z} = \langle z_0, z_1, \ldots, z_{n-1} \rangle$, $\sum_{i=0}^{n-1} |z_i + c|$ is minimized when $c = -\operatorname{median}_{i=0}^{n-1} z_i$.*

Instead of using (median, +) convolution directly, we use the equivalent form, (median, −) convolution:

Theorem 11. *The ℓ_1 necklace alignment problem can be reduced in $O(n)$ time to one* (median, −) *convolution.*

Proof. For two necklaces $\mathbf{x}$ and $\mathbf{y}$, we apply the (median, −) convolution to the following vectors, as in the proof of Theorem 3:

$$\mathbf{x}' = \langle x_0, x_0, x_1, x_1, \ldots, x_{n-1}, x_{n-1}; \underbrace{\infty, -\infty, \infty, -\infty, \ldots, \infty, -\infty}_{2n} \rangle,$$

$$\mathbf{y}' = \langle y_{n-1}, y_{n-1}, y_{n-2}, y_{n-2}, \ldots, y_0, y_0; y_{n-1}, y_{n-1}, y_{n-2}, y_{n-2}, \ldots, y_0, y_0 \rangle.$$

Then, for $s' \in \{0, 1, \ldots, n-1\}$, the $2(n+s')+1$st entry of $\mathbf{x}' \underset{\text{med}}{\bar{*}} \mathbf{y}'$ is $\text{median}_{i=0}^{2(n+s')+1}(x'_i - y'_{2(n+s')+1-i}) = \text{median}_{i=0}^{n-1}(x_i - y_{(i-s'-1) \bmod n})$, which is $\text{median}_{i=0}^{n-1}(x_i - y_{(i+s) \bmod n})$ if we let $s' = n-1-s$. Applying Fact 10, we can therefore minimize $\text{median}_{i=0}^{n-1} |x_i - y_{(i+s) \bmod n} + c|$ over c, for each $s \in \{0, 1, \ldots, n-1\}$. By brute force, we can minimize over s as well using $O(n)$ additional comparisons and time. □

Our results for (median, −) convolution use the following result of Frederickson and Johnson:

Theorem 12. [18] *The median element of the union of k sorted lists, each of length n, can be computed in $O(k \lg n)$ time and comparisons.*

We begin with our results for the nonuniform linear decision tree model:

Theorem 13. *The* (median, −) *convolution of two vectors of length n can be computed in $O(n\sqrt{n \lg n})$ time in the nonuniform linear decision tree model.*

Proof. As in the proof of Theorem 6, we sort the set $D = \{x_i - x_j, y_i - y_j : |i-j| \le d\}$ of pairwise differences between nearby x_i's and nearby y_i's, where $d \le n$ is a value to be determined later. By Theorem 4, this step requires $O(nd + n \lg n)$ comparisons between differences. These comparisons enable us to compare $x_i - y_{k-i}$ versus $x_j - y_{k-j}$ for free, provided $|i-j| \le d$, because $x_i - y_{k-i} < x_j - y_{k-j}$ precisely if $x_i - x_j < y_{k-i} - y_{k-j}$. In particular, we can sort each list $L_k(\lambda) = \langle x_i - y_{k-i} : i = \lambda, \lambda+1, \ldots, \min\{\lambda + d, n\} - 1 \rangle$ for free. By Theorem 12, we can compute the median of $L_k(0) \cup L_k(d) \cup L_k(2d) \cup \cdots \cup L_k(\lceil k/d \rceil d)$, i.e., $\text{median}_{i=0}^{k}(x_i - y_{k-i})$, in $O((k/d) \lg d) = O((n/d) \lg d)$ comparisons. Also, in the same asymptotic number of comparisons, we can binary search to find where the median fits in each of the $L_k(\lambda)$ lists, and therefore which differences are smaller and which differences are larger than the median. This median is the kth entry of $\mathbf{x} \underset{\text{med}}{\bar{*}} \mathbf{y}$. Therefore, we can compute all n entries of $\mathbf{x} \underset{\text{med}}{\bar{*}} \mathbf{y}$ in $O(nd + n \lg n + (n^2/d) \lg d)$ comparisons. This asymptotic running time is minimized when $nd = \Theta((n^2/d) \lg d)$, i.e., when $d^2 / \lg d = \Theta(n)$. Substituting $d = \sqrt{n \lg n}$, we obtain a running time of $O(n\sqrt{n \lg n})$ in the nonuniform linear decision tree model. □

Combining Theorems 11 and 13, we obtain the following result:

Corollary 14. *The ℓ_1 necklace alignment problem can be solved in $O(n\sqrt{n \lg n})$ time in the nonuniform linear decision tree model.*

Now we turn to the analogous results for the real RAM:

Theorem 15. *The* (median, −) *convolution of two vectors of length n can be computed in $O(n^2 (\lg \lg n)^2 / \lg n)$ time on a real RAM.*

Proof. Let $\mathbf{x}$ and $\mathbf{y}$ denote the two vectors of length n, and let $\mathbf{x} \underset{\text{med}}{\bar{*}} \mathbf{y}$ denote their (median, $-$) convolution. For each permutation π on the set $\{0, 1, \ldots, d-1\}$, for each $i \in \{0, d, 2d, \ldots, \lfloor n/d \rfloor d\}$, and for each $j \in \{0, 1, \ldots, n-1\}$, we define the $(d-1)$-dimensional points

$$p_{\pi,i} = (x_{i+\pi(0)} - x_{i+\pi(1)}, x_{i+\pi(1)} - x_{i+\pi(2)}, \ldots, x_{i+\pi(d-2)} - x_{i+\pi(d-1)}),$$
$$q_{\pi,j} = (y_{j-\pi(0)} - y_{j-\pi(1)}, y_{j-\pi(1)} - y_{j-\pi(2)}, \ldots, y_{j-\pi(d-2)} - y_{j-\pi(d-1)}),$$

(To handle boundary cases, define $x_i = \infty$ and $y_j = -\infty$ for indices i, j outside $[0, n-1]$.) For each permutation π, we apply Lemma 7 to the set of red points $\{p_{\pi,i} : i = 0, d, 2d, \ldots, \lfloor n/d \rfloor d\}$ and the set of blue points $\{q_{\pi,j} : j = 0, 1, \ldots, n-1\}$, to obtain all dominating pairs $(p_{\pi,i}, q_{\pi,j})$.

Point $p_{\pi,i}$ dominates $q_{\pi,j}$ precisely if $x_{i+\pi(\delta)} - x_{i+\pi(\delta+1)} \geq y_{j-\pi(\delta)} - y_{j-\pi(\delta+1)}$ for all $\delta \in \{0, 1, \ldots, d-2\}$ (ignoring the indices outside $[0, n-1]$). By rearranging terms, this condition is equivalent to $x_{i+\pi(\delta)} - y_{j-\pi(\delta)} \geq x_{i+\pi(\delta+1)} - y_{j-\pi(\delta+1)}$ for all $\delta \in \{0, 1, \ldots, d-2\}$, i.e., π is a sorting permutation of $\langle x_i - y_j, x_{i+1} - y_{j-1}, \ldots, x_{i+d-1} - y_{j-d+1} \rangle$. If we substitute $j = k - i$, we obtain that $(p_{\pi,i}, q_{\pi,k-i})$ is a dominating pair precisely if π is a sorting permutation of the list $L_k(i) = \langle x_i - y_{k-i}, x_{i+1} - y_{k-i+1}, \ldots, x_{\min\{i+d,n\}-1} - y_{\min\{k-i+d,n\}-1} \rangle$. Thus, the set of dominating pairs gives us the sorted order of $L_k(i)$ for each i divisible by d and for each k. Also, there can be at most $O(n^2/d)$ total dominating pairs $(p_{\pi,i}, q_{\pi,j})$ over all i, j, π, because there are $O(n/d)$ choices for i and $O(n)$ choices for j, and if $(p_{\pi,i}, q_{\pi,j})$ is a dominating pair, then $(p_{\pi',i}, q_{\pi',j})$ cannot be a dominating pair for any permutation $\pi' \neq \pi$. (Here we assume that the sorted order is unique, which can be arranged by standard perturbation techniques or by breaking ties consistently [6].) Hence, the running time of the $d!$ executions of Lemma 7 is $d!\, 2^{O(d)} n^{1+\varepsilon} + O(n^2/d)$ time, which is $O(n^2 \lg\lg n / \lg n)$ if we choose $d = \alpha \lg n / \lg\lg n$ for a sufficiently small constant $\alpha > 0$. By Theorem 12, we can compute the median of $L_k(0) \cup L_k(d) \cup L_k(2d) \cup \cdots \cup L_k(\lceil k/d \rceil d)$, i.e., $\text{median}_{i=0}^{k}(x_i - y_{k-i})$, in $O((k/d) \lg d) = O((n/d) \lg d)$ comparisons. Also, in the same asymptotic number of comparisons, we can binary search to find where the median fits in each of the $L_k(\lambda)$ lists, and therefore which differences are smaller and which differences are larger than the median. This median is the kth entry of $\mathbf{x} \underset{\text{med}}{\bar{*}} \mathbf{y}$. Therefore all n entries can be computed in $O(n^2(\lg d)/d) = O(n^2(\lg\lg n)^2/\lg n)$ time on a real RAM. □

Combining Theorems 11 and 15, we obtain the following result:

Corollary 16. *The ℓ_1 necklace alignment problem can be solved in $O(n^2(\lg\lg n)^2/\lg n)$ time on a real RAM.*

As before, this approach likely cannot be improved beyond $O(n^2/\lg n)$, because such an improvement would require an improvement to Lemma 7, which would in turn improve the fastest known algorithm for all-pairs shortest paths in dense graphs [6]. In contrast to (median, $+$) convolution, (mean, $+$) convolution is trivial to compute in linear time.

5 Conclusion

The convolution problems we consider here have connections to many classic problems, and it would be interesting to explore whether the structural information extracted by our algorithms could be used to devise faster algorithms for these classic problems. For example, does the antidiagonal information of the $X + Y$ matrix lead to a $o(n^2 \lg n)$-time algorithm for sorting $X + Y$? We believe that any further improvements to our convolution algorithms would require progress and/or have interesting implications on all-pairs shortest paths [6].

Our $(\min, -)$-convolution algorithms give subquadratic algorithms for *polyhedral 3SUM*: given three lists, $A = \langle a_0, a_1, \ldots, a_{n-1} \rangle$, $B = \langle b_0, b_1, \ldots, b_{n-1} \rangle$, and $C = \langle c_0, c_1, \ldots, c_{2n-2} \rangle$, such that $a_i + b_j \leq c_{i+j}$ for all $0 \leq i, j < n$, decide whether $a_i + b_j = c_{i+j}$ for any $0 \leq i, j < n$. This problem is a special case of 3SUM, and this special case has an $\Omega(n^2)$ lower bound in the 3-linear decision tree model [15]. Our results solve polyhedral 3SUM in $O(n^2/\lg n)$ time in the 4-linear decision tree model, and in $O(n\sqrt{n})$ time in the nonuniform 4-linear decision tree model, solving an open problem of Erickson [13]. Can these algorithms be extended to solve 3SUM in subquadratic time in the (nonuniform) decision tree model?

Acknowledgments. This work was initiated at the 20th Bellairs Winter Workshop on Computational Geometry held January 28–February 4, 2005. We thank the other participants of that workshop—Greg Aloupis, Justin Colannino, Mirela Damian-Iordache, Vida Dujmović, Francisco Gomez-Martin, Danny Krizanc, Erin McLeish, Henk Meijer, Patrick Morin, Mark Overmars, Suneeta Ramaswami, David Rappaport, Diane Souvaine, Ileana Streinu, David Wood, Godfried Toussaint, Remco Veltkamp, and Sue Whitesides—for helpful discussions and contributing to a fun and creative atmosphere. We particularly thank the organizer, Godfried Toussaint, for posing the problem to us.

References

1. I. Baran, E. D. Demaine, and M. Pătraşcu. Subquadratic algorithms for 3SUM. In *Proc. 9th Worksh. Algorithms & Data Structures*, LNCS 3608, pp. 409–421, 2005.
2. R. Bellman and W. Karush. Mathematical programming and the maximum transform. *J. SIAM*, 10(3):550–567, 1962.
3. D. J. Bernstein. Fast multiplication and its applications. In J. Buhler and P. Stevenhagen, eds., *Algorithmic Number Theory*. Cambridge University Press. To appear.
4. M. Bussieck, H. Hassler, G. J. Woeginger, and U. T. Zimmermann. Fast algorithms for the maximum convolution problem. *Oper. Res. Lett.*, 15(3):133–141, 1994.
5. J. Cardinal, S. Kremer, and S. Langerman. Juggling with pattern matching. *Theory Comput. Syst.*, 39(3):425–437, 2006.
6. T. M. Chan. All-pairs shortest paths with real weights in $O(n^3/\log n)$ time. In *Proc. 9th Worksh. Algorithms & Data Structures*, LNCS 3608, pp. 318–324, 2005.
7. H. Cohn, R. Kleinberg, B. Szegedy, and C. Umans. Group-theoretic algorithms for matrix multiplication. In *Proc. 46th IEEE Symp. Found. Computer Science*, pp. 379–388, 2005.

8. J. Colannino, M. Damian, F. Hurtado, J. Iacono, H. Meijer, S. Ramaswami, and G. Toussaint. An $O(n \log n)$-time algorithm for the restriction scaffold assignment. *J. Comput. Biol.*, 13(4):979–989, 2006.
9. R. Cole and R. Hariharan. Verifying candidate matches in sparse and wildcard matching. In *Proc. 34th ACM Symp. Theory of Computing*, pp. 592–601, 2002.
10. J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comp.*, 19:297–301, 1965.
11. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.
12. E. D. Demaine, J. S. B. Mitchell, and J. O'Rourke. Problem 41: Sorting $X + Y$ (pairwise sums). In *The Open Problems Project*. http://cs.smith.edu/~orourke/TOPP/P41.html.
13. E. D. Demaine and J. O'Rourke. Open problems from CCCG 2005. In *Proc. 18th Canadian Conference on Computational Geometry*, 2006.
14. J. M. Díaz-Báñez, G. Farigu, F. Gómez, D. Rappaport, and G. T. Toussaint. El compás flamenco: A phylogenetic analysis. In *Proc. BRIDGES: Mathematical Connections in Art, Music, and Science*, pp. 61–70, 2004.
15. J. Erickson. Lower bounds for linear satisfiability problems. *Chic. J. Theoret. Comput. Sci.*, 1999.
16. P. F. Felzenszwalb and D. P. Huttenlocher. Distance transforms of sampled functions. TR2004-1963, Faculty of Computing and Information Science, Cornell Univ.
17. M. J. Fischer and M. S. Paterson. String-matching and other products. In *Complexity of computation*, Proc. SIAM-AMS Applied Math. Symp., 1973, pp. 113–125.
18. G. N. Frederickson and D. B. Johnson. The complexity of selection and ranking in $X + Y$ and matrices with sorted columns. *J. Comput. System Sci.*, 24(2):197–208, 1982.
19. M. L. Fredman. How good is the information theory bound in sorting? *Theoret. Comput. Sci.*, 1(4):355–361, 1976.
20. M. L. Fredman. New bounds on the complexity of the shortest path problem. *SIAM J. Comput.*, 5(1):83–89, 1976.
21. C. F. Gauss. *Werke*, vol. 3. Königlichen Gesellschaft der Wissenschaften, 1866.
22. M. T. Heideman, D. H. Johnson, and C. S. Burrus. Gauss and the history of the fast Fourier transform. *Arch. Hist. Exact Sci.*, 34(3):265–277, 1985.
23. P. Indyk. Faster algorithms for string matching problems: Matching the convolution bound. In *Proc. 39th Symp. Found. Computer Science*, pp. 166–173, 1998.
24. P. Maragos. Differential morphology. In S. Mitra and G. Sicuranza, eds., *Nonlinear Image Processing*, ch. 10, pp. 289–329. Academic Press, 2000.
25. J.-J. Moreau. Inf-convolution, sous-additivité, convexité des fonctions numériques. *J. Math. Pures Appl. (9)*, 49:109–154, 1970.
26. R. T. Rockafellar. *Convex Analysis*. Princeton Univ. Press, 1970.
27. W. L. Steiger and I. Streinu. A pseudo-algorithmic separation of lines from pseudolines. *Infor. Process. Lett.*, 53(5):295–299, 1995.
28. T. Strömberg. The operation of infimal convolution. *Dissertationes Math. (Rozprawy Mat.)*, 352:58, 1996.
29. G. Toussaint. The geometry of musical rhythm. In *Proc. Japan Conference on Discrete and Computational Geometry*, LNCS 3742, pp. 198–212, 2004.
30. G. T. Toussaint. A comparison of rhythmic similarity measures. In *Proc. 5th International Conference on Music Information Retrieval*, pp. 242–245, 2004.

Purely Functional Worst Case Constant Time Catenable Sorted Lists

Gerth Stølting Brodal[1], Christos Makris[2], and Kostas Tsichlas[2]

[1] Department of Computer Science, University of Aarhus,
BRICS, Basic Research in Computer Science,
funded by the Danish National Research Foundation.
gerth@brics.dk
www.brics.dk
[2] Department of Computer Engineering and Informatics, University of Patras,
26500 Patras, Greece
{makri, tsihlas}@ceid.upatras.gr

Abstract. We present a purely functional implementation of search trees that requires $O(\log n)$ time for search and update operations and supports the join of two trees in worst case constant time. Hence, we solve an open problem posed by Kaplan and Tarjan as to whether it is possible to envisage a data structure supporting simultaneously the join operation in $O(1)$ time and the search and update operations in $O(\log n)$ time.

Keywords: data structures, sorted lists, purely functional programming.

1 Introduction

The balanced search tree is one of the most common data structures used in algorithms and constitutes an elegant solution to the *dictionary* problem. In this problem, one needs to maintain a set of elements in order to support the operations of insertion, deletion and searching for the predecessor of a query element. In a series of applications [9, 10, 13, 16] search trees are also equipped with the operations of *join* and *split*. The most efficient search trees use linear space and support insertion, deletion, search, join and split operations in logarithmic time; the most prominent examples of them are: AVL-trees, red-black trees, (a, b)-trees, BB[α]-trees and Weight Balanced B-trees.

In commonly used data structures, update operations such as insertions and deletions change the data structure in such a way, that the old version (the version before the update) is destroyed. These data structures are called *ephemeral*. A data structure that does not destroy its previous versions after updates, is called *a persistent* data structure. Depending on the operations allowed by a persistent structure, the following types of persistence can be distinguished:

- *Partial* persistence: only the latest version of the structure can be updated and all the versions can be queried.
- *Full* persistence: all the versions can be updated and queried.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 172–183, 2006.

– *Confluent* persistence: all the versions can be updated and queried and additionally, two versions can be combined to produce a new version. Note that in this case it is possible to create in polynomial time an exponentially sized structure by repeatedly joining it with itself.

The history of a persistent data structure is represented by a *version graph* $G = (V, E)$; a node $v \in V$ corresponds to a version of the data structure while an edge $e \in E$ between nodes v and v' denotes that one of the structures involved in the operation creating v' was v. For the partial persistence case the version graph is a linear list, while for the full persistence case it is a tree. In the case of confluently persistent data structures the version graph is a directed acyclic graph.

Notably, persistent data structures are also met under the name *purely functional* data structures. This term indicates data structures built using operations that correspond to the LISP commands *car, cdr, cons*. These commands create nodes which are immutable and hence fully persistent. However, a full persistent data structure is not necessarily purely functional.

The problem of devising a general framework for turning ephemeral pointer-based data structures into their partial and full persistent counterparts was successfully handled in [7]. The proposed construction works for linked data structures of bounded in-degree and allows the transformation of an ephemeral structure into a partial or full persistent one with only a constant amortized time and space cost. The amortized bounds for the partial persistent construction were turned into worst case in [2]. Additionally, in [16] Okasaki presented simpler constructions by applying the lazy evaluation technique met in functional languages.

The aforementioned general techniques fail to apply in the confluent persistence setting; in this setting the version graph becomes a DAG making the navigation method of [7] to fail. Fiat and Kaplan presented efficient methods to transform general linked data structures to confluently persistent and have showed that if the total number of assignments is U then the update creating version v will cost $O(d(v) + \log U)$ and the space requirement will be $O(d(v) \log U)$ bits, where $d(v)$ is the depth of v in the version graph.

The aforementioned general framework was surpassed in practice by ad hoc solutions for specific data structures. In particular in [8, 4, 11, 12] a set of solutions for constructing confluent persistent deques was presented leading to an optimal solution that could handle every operation in worst case constant time and space cost. The supported set of operations included push, pop, inject, eject and catenate. One of the most interesting examples of purely functional data structure is sorted lists implemented as finger trees. Kaplan and Tarjan described in [13] three implementations, the more efficient of which achieved logarithmic access, insertion and deletion time, and double-logarithmic catenation time. In fact, they supported the search and update operations in $O(\log d)$ time, where d is the number of elements between the queried element and the smallest or the largest element in the tree. They asked whether the join operation can be implemented in $O(1)$ worst-case time even in an ephemeral setting,

while supporting searches and updates in logarithmic time. They sketched a data structure supporting the join operation in $O(1)$ time but the time complexity of search and update operations was $O(\log n \log\log n)$.

In this paper we focus on the problem of efficiently implementing purely functional sorted lists from the perspective of search trees and not finger trees as in [13]. We present a purely functional implementation of search trees that supports join operations in worst-case constant time, while simultaneously supporting search and update operations in logarithmic time. In Figure 1 we provide the complexities of our data structure and compare them with previous results. In Section 2, we introduce the reader to the problem as well as to some basic notions used throughout the paper. In Section 3, we present the main structural elements of the construction and depict how to make the structure purely functional, and finally we conclude in Section 4 with some final remarks.

	Traditional (e.g. AVL, (a,b)-trees)	Kaplan & Tarjan (STOC '96 [13])	This Paper
Search	$O(\log n)$	$O(\log n)$ - $O(\log d)$	$O(\log n)$
Join	$O(\log n)$	$O(\log\log n)$	$O(1)$
Insert/Delete	$O(\log n)$	$O(\log n)$ - $O(\log d)$	$O(\log n)$

Fig. 1. Comparison of the complexities of our data structure with previous results. Here n denotes the number of stored elements, while d denotes the number of elements between the queried element (defined by the insert, delete or search operations) and the smallest or largest element.

2 Definitions

A *biased tree* T [15], [1] is a leaf-oriented search tree storing elements equipped with weights. The *weight* $w(v)$ of a node v in T is the sum of the weights of all the leaves in its subtree. The *weight* $w(T)$ of the tree T is the weight of the root of T. The left (right) spine of the tree T is the path from the root of T to the smallest (largest) element of the tree.

In this paper, we consider the problem of maintaining a set of elements each of weight 1, represented as a collection of trees, subject to the following operations:

1. **Insert**(T_i,x), inserts element x in the tree T_i.
2. **Delete**(T_i,x), deletes the element x, if it exists, from tree T_i.
3. **Search**(T_i,x), returns the position of x in the tree T_i. If x does not exist in T_i, then it returns the position of its predecessor.
4. **CreateTree**(T_i,x), creates a new tree T_i with element x. A tree T_i ceases to exist when it has no elements.
5. **Join**(T_i,T_j), joins the two trees in one tree. Trees T_i and T_j are ordered in the sense that all elements of T_j are either smaller or larger than the smallest or largest element of T_i. Assume without loss of generality that $w(T_i) \geq w(T_j)$. In this case, tree T_j is attached to tree T_i, and the result of this operation is the tree T_i. T_j is attached to a node on the spine of T_i.

There exist various implementations of biased trees differing in the used balance criterion; our construction is based on the biased $2, b$ trees presented in [1] which are analogous to 2,3 trees and B-trees. A $2, b$ tree is a search tree with internal nodes having degree at least 2 and at most b; in a biased $2, b$ tree the rank $r(v)$ of a leaf v is equal to $\lfloor \log w(v) \rfloor$, while the rank of an internal node is one plus the maximum of the ranks of its children. The rank $r(T)$ of the tree is the rank of the root. In [1] it is described how to maintain a $2, b$ tree so that accessing a leaf v takes $O(\log(w(T)/w(v))$ query time; inserting an item i takes $O(\log(w(T)/(w_{i-} + w_{i+})) + \log(w'(T)/w_i))$ time and deleting an item i takes $O(\log(w'(T)/(w_{i-} + w_{i+})) + \log(w(T)/w_i))$ time, where $w(T)$, $w'(T)$, are the weights of the tree before and after the operation and $i-$, $i+$ are the largest item smaller than i and the smallest item larger than i respectively. These bounds are achieved by maintaining a balance criterion termed *global bias*. A *globaly biased* $2, b$ tree is a $2, b$ tree with the property that any neighboring leaf of a node v whose parent w has rank larger than $r(v) + 1$, has rank $r(w) - 1$.

We will employ in our construction biased $2, 4$ trees, using the same definitions of weights, ranks and balance (global bias) as in [1].

3 The Fast Join-Tree

In this section we provide a description of our structure, called the *Fast Join-tree*. Initially we present an overview, introducing the building components of the structure and the way these various components are linked together. Then, we discuss how these components are combined when joining trees in order to produce a worst-case constant time implementation. Finally, we provide the necessary machinery for the update operations as well as the necessary changes that have to be performed in the structure in order to make it purely functional.

3.1 An Overview

The main goal of the proposed data structure is to support the join operation in $O(1)$ worst-case time. The main obstacle in achieving this complexity for the join operation is the location of the appropriate position on the spine (see Section 2, definition of Join operation). This problem can be overcome by performing joins in a lazy manner. In particular, if two trees T and T' such that $r(T) \geq r(T')$ are joined, then T' is inserted in a temporary structure along with other trees that have been joined with T. This temporary structure is called a *tree-collection*. Thus, a tree-collection is a set of elements structured as a forest of trees. During the insertion of trees in the tree-collection, the spine is traversed so that the tree-collection is finally inserted in the appropriate position. The tree-collection is implemented as a simple linked linear list of trees.

A tree-collection can be considered as a weighted element to be inserted in a tree structure. The weight of a tree-collection x is the number of leaves of the trees in x. A tree-collection can be inserted in a fast Join tree in worst-case constant time by employing a preventive top-down rebalancing scheme on the spines.

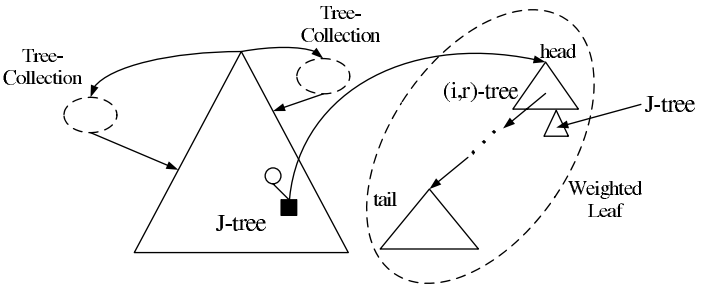


Fig. 2. The structure of the fast Join-tree at one recursive level

The fast Join-tree is better understood as a recursive data structure. A biased search tree constitutes a recursive level of the fast Join-tree, called henceforth a *J-tree*. As a result, the fast Join-tree is a J-tree whose leaves are tree-collections. A tree-collection, as implied earlier, is a linear list of J-trees. These J-trees constitute the second recursion level. The recursion ends when the tree-collection is a simple item. Figure 2 depicts an instance of our structure.

3.2 The Biased Tree and the Tree-Collections

The J-tree is a biased $2,4$-tree with top-down rebalancing at its spines. This tree structure is subject to insertions of *weighted elements* at its spines as well as decrements by one of the rank of one of its leaves. A weighted element can be a simple item equipped with a weight or a tree-collection, which is a forest of trees on weighted elements.

A tree-collection is structured as a simple linear linked list. The insertions of new elements in the tree-collection take place always at the tail of the linked list. The virtual rank $vr(x)$ of a tree-collection x is the number of J-trees contained in x. Its real rank $r(x)$ is equal to the logarithm of the sum of the weights of the participating trees. It will become clear below how these quantities are related to each other. We describe the weighted insertion operation as a one step procedure, but it will in fact be implemented incrementally.

Assume a weighted element x_i with weight $w(x_i)$. This element must be inserted as a weighted leaf of the last (from top to bottom) node on the spine of the tree (either left or right) that has rank larger or equal to $\lfloor \log w(x_i) \rfloor + 1$.

During the traversal of the spine all nodes with 4 children are split, so when the weighted element is inserted there are no cascading splits on the path to the root. It may be the case that the spine is too short, meaning that the weighted element should be inserted at the spine deeper than the length of the spine. In this case, unary nodes must be created in order to put the new element at the appropriate level. However, these nodes can be introduced on demand by attaching the leaf to the last node of the spine. If some other element is inserted with small weight then we attach it as leaf to the last node or introduce new nodes if the number of weighted elements attached to this node is 4.

Consider now an arbitrary leaf l. Our biased tree must support an operation that decreases the rank of the leaf by one. We call this operation *Demote(ℓ,T)*. This decrement by one of the rank of the leaf (which in term of weights is equivalent to reducing by half its weight) can be implemented by simply moving from this leaf to the root of the tree, and using similar techniques as those described in [1].

The following lemma summarizes the properties of the described biased tree.

Lemma 1. *There exists an implementation of a biased tree T, such that the tree has height $O(\log w(T))$ and supports the operations of insertion of a weighted element at its spines and demotion of a weighted element in $O\left(\log \frac{w(T)}{w(x_i)}\right)$ time, where x_i is the element inserted or demoted.*

Proof. The insert operation places the elements always at the correct level which is never changed by the insertion operation. Hence by a similar line of arguments as that in [1] we can conclude that T has height $O(\log w(T))$. From the insertion algorithm an element x_i is inserted at a level such that the path from the root to this level has length at most $O(\log w(T) - \log w(x_i))$ which is equal to $O\left(\log \frac{w(T)}{w(x_i)}\right)$. Finally by using the analysis in [1] it is proved that the demote operation takes the same time complexity. □

3.3 The Join Operation

The fast Join-tree is a biased tree with one tree-collection attached to each of its spines and tree-collections at its leaves. The tree-collection is a weighted element that must be inserted at the appropriate position on the spine of the biased tree. This insertion is incrementally implemented during the future join operations. When the appropriate node for the tree-collection has been found, it is attached to this node as a weighted leaf and the process starts again from the root with a new and possibly empty tree-collection.

The tree-collection x_L (for the left spine) maintains a pointer p_L that traverses the spine of the biased tree T starting from its root v. Assume that T is involved in a join operation with some other tree T', such that $R = r(T) \geq r(T')$. Then, tree T' is inserted in the tree-collection x_L and p_L is moved one node down the spine. The choice of moving one node down the spine is arbitrary and any constant number would do. During the traversal of the spine a simple counter is maintained, which denotes the ideal rank of each node on the spine. This counter is initialized to R, and each time we traverse a node on the spine it is reduced by one. When a node is located such that the counter is equal to $r(x_L) + 1$ the tree-collection is inserted and the process starts again from the root with a new tree-collection. The inserted tree-collection is inserted as a weighted leaf of this node.

Assume that T' and T are joined, where $r(T) \geq r(T')$, and that the tree-collection x_L points to a node u on the spine with rank $r(u)$. There are two cases as to the relation of $r(T')$ and $r(u)$:

1. $\lfloor \log(w(x_L) + w(T')) \rfloor + 1 < r(u)$: in this case the tree-collection x_L after the insertion of T' in it must be inserted somewhere down the spine hence the traversal must be continued.
2. $\lfloor \log(w(x_L) + w(T')) \rfloor + 1 \geq r(u)$: in this case the weight of T' introduced in x_L is too much and x_L should be inserted higher up the spine. If $\lfloor \log(w(x_L) + w(T')) \rfloor + 1 < r(u) + 1$ then x_L canbe safely inserted as a child of the father of node u. Otherwise, the tree-collection x_L without T' is made a child of u and a new tree-collection is created and initialized to tree T' that starts the traversal of the spine from the root. We call x_L the *stepchild* of node u.

For the second case the following lemma holds:

Lemma 2. $r(T') > r(x_L)$.

Proof. (by contradiction) Assume that $r(T') \leq r(x_L)$. Then, by the addition of T' the tree-collection will have rank at most $r(x_L) + 1$. However, this is not possible since x_L could have been attached without taken T' into account. □

A stepchild is a weighted element that was not inserted in its correct position but higher on the spine. We assume that the stepchild does not contribute to the out-degree of its father. The following property is essential:

Property 1. Each internal node of the fast Join-tree has at most one stepchild.

This property is a direct consequence of Lemma 2. Thus, node u is not anymore part of the spine and the property follows. A similar issue arises in the case when tree T' is merged with the larger tree T. As before, the two non-empty tree-collections of T' are made stepchildren of the nodes they currently point to. Tree T' is not part of the spine of T, thus Property 1 is maintained.

One problem that arises from the use of stepchildren is that Lemma 1 may not hold anymore. Fortunately, this is not the case because of Property 1 and the fact that the stepchildren cannot cause any splits on the spine.

The result of this discussion is that the tree-collections move only once the spine from root to leaf. The following property exploits this fact and relates the virtual rank $vr(x)$ of a tree-collection x with the corresponding pointer p with its rank $r(x)$ and the rank R of T:

Property 2. $R - r(x) > vr(x)$

Proof. We prove that $R - vr(x) > r(x)$. $R - vr(x)$ is the maximum rank of the node pointed by the pointer p of the tree-collection x. This is because, after $vr(x)$ insertions of trees in the tree-collection x, the pointer p has traversed $vr(x)$ nodes down the spine. Since, the tree-collection x has not been attached to any node yet, its rank must be less than the rank of the node pointed by p. The inequality follows and the property is proved. □

The join operation is performed in $O(1)$ worst-case number of steps since a constant number of nodes are traversed on the spine and a single insertion is

performed in one tree-collection. We now move to the discussion of the search operation. The search starts from the root of the fast Join-tree T and traverses a path until a weighted leaf ℓ is reached. The search continues in the forest of J-trees in tree-collection ℓ. The search in a forest of J-trees is implemented as a simple scan of a linear linked list from head to tail. Note that the head of the list is the first element ever inserted in the tree-collection.

The following lemma states that searching in a tree-collection is efficient.

Lemma 3. *If the search procedure needs $O(p)$ steps to locate the appropriate J-tree T' in a tree collection in J-tree T, then the rank of T' is at most equal to the rank of T reduced by p.*

Proof. Assume that at some recursive level of detail the J-tree T has rank $R = r(T)$. In addition, let T' be the p-th tree in the tree-collection. Since T' is the p-th tree in the collection, its insertion must occured at the p-th step of the spine's traversal. Since when traversing the spine we visit nodes of reduced rank we get as a result that T', being the J-tree in the p-th position has rank smaller than T by at least p, and the lemma follows. □

The following theorem states the logarithmic complexity of the search operation by using Lemma 3.

Theorem 1. *The search operation in a fast Join-tree T is carried out in $O(\log w(T))$ steps.*

Proof. Assume that the search procedure enters a weighted leaf which is a child of a node u in the J-tree T. Additionally, assume that in the forest the search procedure explores the p-th J-tree T'.

We show that:

$$r(T') \leq \min\{r(T) - p, r(u)\} \tag{1}$$

Since the search has reached node u we get that $r(T') \leq r(u)$. This observation in conjunction with Lemma 3 proves Equation 1.

Equation 1 states that the rank of the search space is decreased by 1 after $O(1)$ steps. As a result, to find a single element in the J-tree T we need $O(\log w(T))$ steps. □

3.4 Supporting Update Operations

We now describe how the J-tree supports insertions and deletions of single elements. The update operations may cause the fast Join-tree to become unbalanced; thus rebalancing strategies must be employed in order to ensure that this is not the case. We first show how insertions are implemented and then we move to the case of deletions.

Insertions. We implement insertions by using a two level data structure. The first level of the structure is the fast Join-tree while the second one is a traditional degree balanced $(2, 4)$-tree as described in [15]. Each leaf of the fast

Join-tree is a degree balanced $(2, 4)$-tree. Hence, the structure can be seen as a forest of $(2, 4)$-trees over which a secondary structure is built to incorporate join operations. Consequently, the first level of the structure implements efficiently the join operation while the second level implements the insertion operation. Figure 3 depicts the structure.

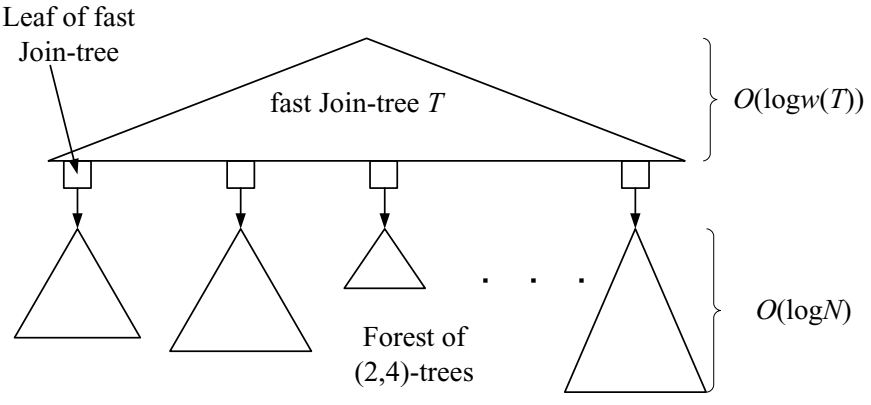


Fig. 3. A high level view of the structure for insertions

When inserting an element we first need to locate its position in the structure. This is accomplished by searching the fast Join-tree for the appropriate leaf by using Theorem 1. The leaf represents a $(2, 4)$-tree in the second level of the structure, in which the element is inserted without affecting the fast Join-tree. This means that insertions do not affect the weights of the internal nodes of the fast Join-tree. The weight of the fast Join-tree is the number of leaves, that is the number of $(2, 4)$-trees in the second level.

When the position of the insertion is located in a $(2, 4)$-tree in the second level, the element is inserted. Finally, rebalancing operations are performed in the $(2, 4)$-tree and the insertion procedure terminates. Thus, the fast Join-tree is by no means affected by this insertion operation. As a result, there may be the case that a very light fast Join-tree has very large number of stored elements because of insertion operations, and we get the following property:

Property 3. The number of leaves of the fast Join-tree is a lower bound for the number of elements stored in the forest of $(2, 4)$-trees in the second level of the structure.

By Theorem 1, a leaf in a fast Join-tree T is located in $O(\log w(T))$ steps. When the leaf is located a second search operation is initiated in the $(2, 4)$-tree attached to this leaf. If the number of elements stored in the forest of $(2, 4)$-trees is N, then by Property 3 we get that $w(T) \leq N$. As a result, the total time complexity for the search operation is bounded by $O(\log w(T)) + O(\log N)$, which is equal to $O(\log N)$. Consequently, insertion operations are supported efficiently by applying this two level data structure.

Deletions. The delete operation cannot be tackled in the same way as the insertion operation. This is because when a leaf of the fast Join-tree becomes empty, the weight of internal nodes is reduced by one. We devise a rebalancing strategy for the fast Join-tree in the case of deletions. Additionally, we introduce a *fix* operation, which incrementally joins leaves from the forest of $(2, 4)$-trees.

As in the case of insertions, first the element must be located. The search procedure locates the appropriate leaf of the fast Join-tree and then locates the element in the $(2, 4)$-tree attached to this leaf. This element is removed from the $(2, 4)$-tree by applying standard rebalancing operations (fuse or share). If the $(2, 4)$-tree is non-empty after the deletion, then the procedure terminates and the fix operation is initiated. If it is empty, then rebalancing operations must be forced on the fast Join-tree.

Since this leaf is empty it is removed from the fast Join-tree. If this leaf belonged to a tree-collection x then the virtual rank has decreased by one and potentially the rank of x has decreased by one. By employing a demote operation on the biased tree whose leaf is this tree-collection the change of the rank is remedied. This change may propagate up to the J-tree of the first recursive level. During this traversal, the weight of all nodes on the path to the root is updated accordingly. When the root of the fast Join-tree is reached, the deletion operation terminates. Note that this operation does not violate Property 2. This is because both the rank and the virtual rank of the tree-collection are reduced.

As shown above the deletion operation may reduce by one the virtual rank of the tree-collection. Thus, a tree-collection may be completely empty after a deletion operation. This means that the leaf is removed and rebalancing operations (fuse or share) must be performed in the biased tree to maintain its structural properties.

A final detail is how deletions interact with stepchildren. In this case, when a tree-collection is demoted then if one of its adjacent brothers is a stepchild, we also demote the stepchild. If the stepchild reached its correct level then it ceases to be a stepchild and it is inserted as an ordinary tree-collection. This procedure may be seen as an insertion of a weighted element in a J-tree, thus inducing rebalancing operations which may propagate up to the root of the fast Join-tree. The time complexity of the delete operation remains unaffected.

We now switch to the description of the fix operation. The fix operation is used as a means of globally correcting the data structure, that is it is not mandatory, however it is used to give a better shape to the structure. Each time an update operation is performed, the fix operation picks 4 leaves of the fast Join-tree and merges them together. Each one of these leaves represents a $(2, 4)$-tree. The merge of these trees can be performed in $O(\log n)$ time, where n is the number of elements stored in the forest of $(2, 4)$-trees. From this merge three leaves of the fast Join-tree become empty. Thus, rebalancing operations must be employed on the fast Join-tree for these leaves. All in all, the time complexity for the delete operation is $O(\log n)$.

The problem posed by this delete operation is that Property 2 may be violated. As a result, the search bounds and consequently the update bounds will not be

logarithmic anymore. Assume that N is the number of leaves of the fast Join-tree T before the fix operation is initiated, where $n \geq N$. Thus, the rank of T is $R = \log N$. During the fix operation, until all the leaves are merged into a single $(2,4)$-tree and the fast Join-tree is a single node, the number of elements in the forest of $(2,4)$-trees will never decrease below $\frac{n}{2}$. By Theorem 1, the time complexity for the search will be $O(R) + O(\log n)$, and since $R = O(\log n)$ the time complexity for the deletion follows.

By Theorem 1 and the previous discussion, we get the following theorem:

Theorem 2. *There exists a search tree supporting search and update operations in $O(\log n)$ worst case time and meld operations in worst case constant time.*

3.5 Purely Functional Implementation

The nodes that compose our structure have all degrees that are bounded by a constant. Hence, in order to make our structure work efficiently in a purely functional setting we need the following ingredients:

- a purely functional implementation of the linear list with which we implement the tree collection. This structure is implemented purely functionally in [3, 16].
- a purely functional implementation of the left and right tree spines. The presence of the pointer designating the movement of the tree-collection is quite awkward, since in a functional setting explicitly assigning values is not permitted. However, the pointer movement can be modeled by partitioning each spine into two lists the border of which designates the pointer position. These lists should be implemented purely functionally and should support both catenation and split operations. Details of such an implementation can be found in [14].

By Theorem 2 and the previous discussion, we get:

Theorem 3. *There exists a purely functional implementation of search trees supporting search and update operations in $O(\log n)$ worst case time and join operations in worst case constant time.*

4 Conclusion

We have presented a purely functional implementation of catenable sorted lists, supporting the *join* operation in worst case constant time, the *search* operation in $O(\log n)$ time and the insertion and deletion operations in $O(\log n)$ time. This is the first purely functional implementation of search trees supporting the *join* operation in worst case constant time.

It would be very interesting to implement efficiently the *split* operation. It seems quite hard to do this in the proposed structure because of the dependence of Property 2 on the rank of the tree. Splitting will invalidate this property for every tree-collection and will lead to $(\log n \log \log n)$ search and update times.

It would also be interesting to come up with an efficient purely functional implementation of sorted lists, implemented as finger trees (as in [13]) that could support join in worst case constant time. In this structure, it is quite unlikely to implement finger searching due to the relaxed structure of the fast Join-tree.

References

1. Bent, S., Sleator, D., Tarjan R. Biased Search Trees, *SIAM Journal of Computing* 14:545–568, 1985.
2. Brodal, G.S. Partially Persistent Data Structures of Bounded Degree with Constant Update Time, *Nordic Journal of Computing*, 3(3):238-255, 1996.
3. Brodal, G.S., and Okasaki, C. Optimal Purely Functional Priority Queues. *Journal of Functional Programming*, 6(6):839-857, 1996.
4. Buchsbaum, A. and R. E. Tarjan. Confluently persistent deques via data structural bootstrapping. *Journal of Algorithms*, 18:513-547, 1995.
5. Dietz P.F. Fully Persistent Arrays. *In Proc. of Workshop on Algorithms and Data Structures (WADS)*, pp. 67-74, 1989.
6. Dietz, P. and Raman, R. Persistence. Amortization and Randomization. *In Proc. of the 2nd ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 78-88, 1991.
7. Driscoll, J.R., Sarnak, N., Sleator, D., and Tarjan, R.E. Making Data Structures Persistent. *Journal of Computer and System Sciences*, 38(1):86-124, 1989.
8. Driscoll, J.R., Sleator, D., and Tarjan, R.E. Fully Persistent Lists with Catenation. *Journal of the ACM*, 41(5):943-959, 1994.
9. Fiat, A., and Kaplan, H. Making Data Structures Confluently Persistent. *Journal of Algorithms*, 48(1):16-58, 2003.
10. Kaplan, H. Persistent Data Structures. *Handbook of Data Structures*, CRC Press, Mehta, D., and Sahni, S., (eds.), 2004.
11. Kaplan, H., Okasaki, C., and Tarjan, R. E. Simple Confluently Persistent Catenable Lists. *SIAM Journal of Computing*, 30(3):965-977, 2000.
12. Kaplan, H., and Tarjan, R.E. Purely Functional, Real-Time Deques with Catenation. *Journal of the ACM*, 46(5):577-603, 1999.
13. Kaplan H., and Tarjan, R.E. Purely Functional Representations of Catenable Sorted Lists. *In Proc. of the 28th Annual ACM Symposium on Theory of Computing (STOC)*, 202-211, 1996.
14. Kaplan, H., and Tarjan, R.E. Persistent Lists with Catenation via Recursive Slowdown. *In Proc. of the 27th Annual ACM Symposium on Theory of Computing*, pp. 93-102, 1995.
15. Mehlhorn, K. Data Structures and Algorithms 1: Sorting and Searching. *EATCS Monographs on Theoretical Computer Science*, Springer-Verlang, 1984.
16. Okasaki, C. Purely Functional Data Structures, Cambridge University Press, 1998.
17. Okasaki, C. Purely Functional Random-Access Lists. *In Conf. on Functional Programming Languages and Computer Architecture (FPCA)*, pp. 86-95, 1995.

Taxes for Linear Atomic Congestion Games*

Ioannis Caragiannis, Christos Kaklamanis, and Panagiotis Kanellopoulos

Research Academic Computer Technology Institute and
Dept. of Computer Engineering and Informatics
University of Patras, 26500 Rio, Greece

Abstract. We study congestion games where players aim to access a set of resources. Each player has a set of possible strategies and each resource has a function associating the latency it incurs to the players using it. Players are non–cooperative and each wishes to follow strategies that minimize her own latency with no regard to the global optimum. Previous work has studied the impact of this selfish behavior to system performance. In this paper, we study the question of how much the performance can be improved if players are forced to pay taxes for using resources. Our objective is to extend the original game so that selfish behavior does not deteriorate performance. We consider atomic congestion games with linear latency functions and present both negative and positive results. Our negative results show that optimal system performance cannot be achieved even in very simple games. On the positive side, we show that there are ways to assign taxes that can improve the performance of linear congestion games by forcing players to follow strategies where the total latency suffered is within a factor of 2 of the minimum possible; this result is shown to be tight. Furthermore, even in cases where in the absence of taxes the system behavior may be very poor, we show that the total disutility of players (latency plus taxes) is not much larger than the optimal total latency. Besides existential results, we show how to compute taxes in time polynomial in the size of the game by solving convex quadratic programs. Similar questions have been extensively studied in the model of non-atomic congestion games. To the best of our knowledge, this is the first study of the efficiency of taxes in atomic congestion games.

1 Introduction

We study the well-known *congestion games* introduced by Rosenthal [22]. In a congestion game Π there is a set E of resources and a set N of n players. Each player i has a positive unsplittable demand (or weight) w_i and a set of actions $\mathcal{P}_i \subseteq 2^E$ (each action of player i is a set of resources). Each resource e has a non-negative and non-decreasing latency function f_e defined over non-negative numbers. A resource e used by players with total demand w causes a latency of $f_e(w)$ to each of them. Players are non–cooperative and each wishes

* This work was partially supported by the European Union under IST FET Integrated Project 015964 AEOLUS.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 184–195, 2006.

to minimize her own cost (the cumulative latency experienced at the resource used) with no regard to the global optimum. *Network congestion games* can be used to model non-cooperative users in a communication network, where each user i aims to communicate an amount of traffic w_i through a least congested single path connecting two particular nodes s_i and t_i. In this setting, resources correspond to network links and the actions of user i are all the paths connecting node s_i to t_i.

In general, players follow *mixed strategies*, i.e., player i selects a probability distribution $y_i = \{y_{ip} | p \in \mathcal{P}_i\}$ over her actions. Mixed strategies where $y_{ip} \in \{0, 1\}$ are called *pure strategies*. Each player is aware of the strategies selected by all other players. We denote by y_{ie} the probability that player i uses resource e. Clearly, $y_{ie} = \sum_{p \in \mathcal{P}_i : e \in p} y_{ip}$. We use the term *assignment* to refer to the vector of players' strategies. In a pure assignment, all players follow pure strategies. Given an assignment y, we denote by $L_{ip}(y; \Pi)$ the expected latency of player i when selecting action p. Then the expected latency of player i is $L_i(y; \Pi) = \sum_{p \in \mathcal{P}_i} y_{ip} L_{ip}(y; \Pi)$. An assignment y is a (mixed or pure) *Nash equilibrium* if no player has an incentive to unilaterally change her strategy, i.e., $L_i(y; \Pi) \leq L_i(y_{-i}, x_i; \Pi)$ for any player i and for any probability distribution x_i over the actions in $\mathcal{P}_i$, where y_{-i}, x_i denotes the assignment obtained by y when player i deviates from y_i to x_i. The *weighted total latency* defined as $W(y; \Pi) = \sum_i w_i L_i(y; \Pi)$ has been used as a measure of performance of assignment y in game Π. Another natural measure of performance is the *total latency* defined as $T(y; \Pi) = \sum_i L_i(y; \Pi)$. The *price of anarchy* [17, 21] (with respect to the weighted total latency) of a game Π is the maximum of the ratio of $W(y; \Pi)/W(x; \Pi)$ where y is a Nash equilibrium and x is any assignment for Π. Similarly, we may define the price of anarchy with respect to the total latency. We use the terms *unweighted* and *weighted* for congestion games in order to denote whether players have equal weights or not. Clearly, in unweighted congestion games, the weighted total latency equals the total latency.

[9, 11, 12, 13, 16, 17, 19] study various games which can be thought of as special cases of congestion games with respect to the complexity of computing equilibria of best/worst social cost and the price of anarchy when the social cost is defined as the maximum latency experienced by any player. These include *linear congestion games*, i.e., games with latency functions of the form $f_e(w) = \alpha_e w + b_e$ with non-negative constants α_e and b_e, and *load balancing games*, i.e., linear congestion games where the actions of players are singleton sets. In load balancing terminology, we refer to the resources of a load balancing game as machines. The performance measure of the weighted total latency has been studied in [1, 5, 6, 18, 24]. Awerbuch et al. [1] and, independently, Christodoulou and Koutsoupias [6] prove tight bounds on the price of anarchy of congestion games. Among other results concerning polynomial latency functions, they show that the price of anarchy of pure Nash equilibria in unweighted linear congestion games is 5/2 while for mixed Nash equilibria or pure Nash equilibria of weighted players it is 2.618. Bounds on the price of anarchy of pure Nash equilibria were recently proved to be tight even for load balancing games [5] while better bounds

exist only for load balancing games on machines with identical latency functions [5, 24]. The authors of [18] study symmetric load balancing games where all machines are actions for all players.

In order to mitigate the impact of selfish behavior on system performance, we introduce *taxes* to the resources. We use a *tax function* $\delta : E \times Q^+ \to Q^+$ that assigns a tax $\delta_e(w)$ to each player of weight w that wishes to use e. Assuming selfish behavior of the players, we obtain a new *extended game* (Π, δ) where each player now aims to minimize the expected latency she suffers plus the taxes she pays. The tax paid by player i when selecting action p is $\Delta_{ip}(\Pi, \delta) = \sum_{e \in p} \delta_e(w_i)$. Given an assignment y, the expected tax paid by player i is $\Delta_i(y; \Pi, \delta) = \sum_{p \in \mathcal{P}_i} y_{ip} \Delta_{ip}(\Pi, \delta)$. Again, y is a Nash equilibrium for the extended game if no player has an incentive to unilaterally change her strategy, i.e., $L_i(y; \Pi) + \Delta_i(y; \Pi, \delta) \leq L_i(y_{-i}, x_i; \Pi) + \Delta_i(y_{-i}, x_i; \Pi, \delta)$. We use two measures of performance in the extended game (Π, δ) extending the measures of total latency and weighted total latency in congestion games without taxes. The *total cost* of an assignment y is $T(y; \Pi, \delta) = \sum_i (L_i(y; \Pi) + \Delta_i(y; \Pi, \delta))$, while the *weighted total cost* of an assignment y is $W(y; \Pi, \delta) = \sum_i w_i (L_i(y; \Pi) + \Delta_i(y; \Pi, \delta))$.

Motivated by [8], we distinguish between refundable and non-refundable taxes. In the former case, we assume that the collected taxes can be feasibly returned (directly or indirectly) to the players (e.g., as a "lump-sum refund") and therefore do not contribute to the overall system disutility. However, refunding the collected taxes could be logistically or economically infeasible; the latter case models this scenario.

Definition 1. *A function $\delta : E \times Q^+ \to Q^+$ is a ρ-mixed-efficient refundable tax for the congestion game Π with respect to the total latency (resp. weighted total latency) if $T(y; \Pi, 0) \leq \rho \cdot T(x; \Pi, 0)$ (resp. $W(y; \Pi, 0) \leq \rho \cdot W(x; \Pi, 0)$) for any mixed Nash equilibrium y in the extended game (Π, δ) and any assignment x. A function $\delta : E \times Q^+ \to Q^+$ is a ρ-mixed-efficient non-refundable tax for the congestion game Π with respect to the total cost (resp. weighted total cost) if $T(y; \Pi, \delta) \leq \rho \cdot T(x; \Pi, 0)$ (resp. $W(y; \Pi, \delta) \leq \rho \cdot W(x; \Pi, 0)$) for any mixed Nash equilibrium y in the extended game (Π, δ) and any assignment x.*

Similarly, we define ρ-pure-efficient refundable and non-refundable taxes by constraining y to be a pure Nash equilibrium. We use the terms pure-optimal and mixed-optimal to refer to 1-pure-efficient and 1-mixed-efficient taxes, respectively.

The bounds on the price of anarchy of congestion games with respect to the weighted total latency can be also expressed using the above definition. Any tight bound of ρ on the price of anarchy over mixed (resp. pure) Nash equilibria implies that the *trivial tax function* that assigns no tax to the resources is ρ-mixed-efficient (resp. ρ-pure-efficient) and no better in general. Another issue which is related to our study is that of *network design* for selfish players (or *resource removal*). In this setting, the question is whether the performance of the game can be improved by removing some of the resources; this is equivalent to a tax function which assigns to each resource a tax of either 0 or ∞ for all players. [3] proves that deciding whether resource removal for a weighted linear

congestion game Π can yield price of anarchy better than 2.618 is NP-complete. Furthermore, there are games where this is not feasible at all, implying that taxes of this type are not better than 2.618-pure-efficient.

The problem of computing optimal taxes has been extensively considered in the model of non-atomic congestion games [23]. The main difference of these games from the atomic ones we study in this paper is that each player controls an infinitesimally small demand related to the total demand on the system, thus, the actions of a single player have negligible effect on the system performance. This difference is substantial enough so that the related results (see [8, 10, 15] and the references therein) do not carry over to our model. In fact, even nearly-optimal taxes do not always exist in our model.

In this paper we show the following results. We first study symmetric load balancing games, where we show how to compute pure-optimal taxes for unweighted players. We present lower bounds stating that optimal taxes may not be feasible even in very simple games. In particular, there are unweighted load balancing games on identical machines that do not admit $(11/10 - \epsilon)$-pure-efficient taxes, weighted load balancing games on identical machines that do not admit $(9/8 - \epsilon)$-pure-efficient taxes (note that this bound matches the upper bound on the price of anarchy for these games [18]), and unweighted load balancing games on identical machines that do not admit $(2 - \epsilon)$-mixed-efficient taxes. Even simple non-load-balancing congestion games with unweighted players may not admit $(6/5 - \epsilon)$-pure-efficient taxes either. For unweighted congestion games, we present a *universal tax function* by showing that, for a particular value of the parameter τ which is shown to be best possible, the function $\delta_e = \alpha_e \tau$ is $(1 + 2/\sqrt{3})$-pure-efficient, thus beating the lower bound of 5/2 on the price of anarchy of pure Nash equilibria. This is an interesting result since the tax function does not depend at all on the game played on the resources; it depends only on the resources themselves. Next we exploit solutions of convex quadratic programs to compute 2-mixed-efficient taxes for congestion games with respect to both the weighted total latency and the total latency. Note that the first result beats the lower bound of 2.618 on the price of anarchy while when the total latency is of concern, the price of anarchy is unbounded. Both bounds are tight.

We also consider the case of non-refundable taxes. When considering the weighted total cost, it seems that there is not much room for beating the lower bounds on the price of anarchy. However, we show that weighted load balancing games on identical machines admit $(1 + \sqrt{2})$-mixed-efficient non-refundable taxes. This is an existential result since the tax defined uses an *optimal assignment* (i.e., the pure assignment minimizing the weighted total latency). It can be made algorithmic and yield a marginally worse $1 + \sqrt{2} + \epsilon$ bound when the number of machines is constant exploiting a PTAS from [4] for approximating the optimal assignment. This result should be compared to the lower bound of 5/2 on the price of anarchy over pure Nash equilibria proved recently in [5]. Recall that the price of anarchy of weighted congestion games is unbounded when the total latency is of concern. Somehow surprisingly, we show that any congestion game admits 4-mixed-efficient non-refundable taxes with respect to

the total latency. Furthermore, 6-mixed-efficient non-refundable taxes for these games can be computed in polynomial time. Here, we exploit semi-pure assignments with particular properties which are obtained by rounding the fractional solutions of a convex quadratic program to half-integral ones. The use of convex quadratic programming is motivated by [2] where integral solutions of such programs have been used to approximate scheduling on unrelated machines. However, we rarely need integrality; even fractional or half-integral solutions suffice in order to compute taxes. For the analysis of the upper bounds we develop and use two inequalities that characterize Nash equilibria of congestion games with taxes.

Some details related to the extended game as well as convex quadratic programs are presented in Section 2. The results on refundable and non-refundable taxes are presented in Sections 3 and 4, respectively. We conclude with open problems in Section 5. Due to lack of space, many proofs have been omitted from this extended abstract.

2 Preliminaries

Properties of the extended game. For a weighted congestion game Π and a tax function δ, the extended game (Π, δ) can be seen as a congestion game with player-specific latency functions [14, 20]. Although this is not always true for such games, we can show that the extended game always has a pure Nash equilibrium and hence pure-efficient taxes are well defined. In order to prove this fact, we define a potential function over pure assignments of the extended game by slightly modifying the potential function of weighted linear congestion games in [12].

In our proofs, we use the equivalent expressions of the (weighted) total cost of assignments in the extended game given in the next lemma. The proof easily follows by the definitions.

Lemma 1. *For each assignment y in a weighted congestion game Π with linear latency functions of the form $f_e(w) = \alpha_e(w) + b_e$ and a tax function δ, the following equations hold*

$$W(y;\Pi,\delta) = \sum_e \left(\alpha_e \left(\left(\sum_i y_{ie} w_i \right)^2 + \sum_i y_{ie} \left(1 - y_{ie}\right) w_i^2 \right) + b_e \sum_i y_{ie} w_i + \sum_i y_{ie} w_i \delta_e(w_i) \right)$$

$$T(y;\Pi,\delta) = \sum_e \left(\alpha_e \left(\left(\sum_i y_{ie} \right) \left(\sum_i y_{ie} w_i \right) + \sum_i y_{ie} \left(1 - y_{ie}\right) w_i \right) + b_e \sum_i y_{ie} + \sum_i y_{ie} \delta_e(w_i) \right)$$

where w_i denotes the weight of player i.

In our analysis, we use the two inequalities stated in the following, which characterize Nash equilibria of the extended game. Although complicated at first glance, when examined carefully (and together with the expressions in Lemma 1), these inequalities provide insight about what efficient taxes should look like.

Lemma 2. *Given a weighted congestion game Π and a tax function δ, consider a mixed Nash equilibrium y and any assignment x of (Π, δ). Then*

$$W(y;\Pi,\delta) \leq \sum_e \left(\alpha_e \left(\left(\sum_i x_{ie} w_i \right) \left(\sum_i y_{ie} w_i \right) + \sum_i x_{ie} \left(1 - y_{ie}\right) w_i^2 \right) + b_e \sum_i x_{ie} w_i + \sum_i x_{ie} w_i \delta_e(w_i) \right) \tag{1}$$

$$T(y;\Pi,\delta) \leq \sum_e \left(\alpha_e \left(\left(\sum_i x_{ie} \right) \left(\sum_i y_{ie} w_i \right) + \sum_i x_{ie} \left(1 - y_{ie}\right) w_i \right) + b_e \sum_i x_{ie} + \sum_i x_{ie} \delta_e(w_i) \right) \tag{2}$$

Computation of taxes. In most cases, in order to compute taxes, we wish to compute assignments that satisfy some property; these correspond to solutions of programs of the form:

$$\begin{aligned} \text{(QP1)} \quad & \text{minimize } g(x) \\ & \text{subject to } x_{ie} \geq \sum_{p \in \mathcal{P}_i : e \in p} x_{ip}, \qquad i \in N, e \in E \\ & \sum_{p \in \mathcal{P}_i} x_{ip} \geq 1, \qquad i \in N \\ & x_{ie}, x_{ip} \geq 0, \qquad i \in N, e \in E, p \in \mathcal{P}_i \end{aligned}$$

where $g(x)$ is a convex quadratic function. Convex quadratic programs can be solved within any additive error ϵ in time polynomial in the size of the program and $1/\epsilon$. So, programs like (QP1) are solvable in polynomial time when the total number of actions is polynomial. In many interesting cases like in network congestion games, actions may be exponentially many. However, we can overcome this difficulty for these games and efficiently solve (QP1) in time polynomial in the number of resources and the number of players by considering it as a flow problem. Details will appear in the final version of the paper.

One would hope to solve (QP1) with the objective functions $W(x;\Pi,0)$ or $T(x;\Pi,0)$ and obtain optimal assignments, i.e., mixed assignments of minimum (weighted) total latency. Unfortunately, these functions are non-convex and, furthermore, they are always optimized at pure assignments. This is not difficult to see since the (weighted) total latency of a mixed assignment can be seen as the expectation of the (weighted) total latency of the pure assignments implied by the corresponding probability distributions. Hence, optimizing these functions would also contradict hardness results in [4].

3 Refundable Taxes

We start with an encouraging result concerning pure-optimal refundable taxes.

Theorem 1. *Pure-optimal refundable taxes in unweighted symmetric load balancing games always exist and are computable in polynomial time.*

Proof. (*Sketch*) Consider an unweighted symmetric load balancing game Π with latency functions $f_e(w) = \alpha_e w + b_e$. Let e' be the machine with the smallest a_e among all machines e with non-zero α_e. Let $\epsilon = \alpha_{e'}/2$. Also, let o_e be the number of players that select machine e in an optimal assignment. Let e^* be the machine with maximum $\alpha_e o_e + b_e$ among all machines. For each machine e with $\alpha_e > 0$, we define $\delta_e = \alpha_{e^*} o_{e^*} + b_{e^*} - \alpha_e o_e - b_e$. Let e_0 be the machine with minimum b_e among all machines e with $\alpha_e = 0$. We define $\delta_{e_0} = \alpha_{e^*} o_{e^*} + b_{e^*} + \epsilon$ and $\delta_e = \infty$ for all other machines e with $\alpha_e = 0$. We can show that the function δ is a pure-optimal refundable tax for game Π.

Polynomial time computability follows since optimal assignments are easy to compute through a reduction to a minimum cost flow problem. We construct a network F as follows. For each resource e of the game, F has two nodes u_e and v_e connected through n parallel directed edges g_e^i of unit capacity and cost $\alpha_e (2i-1) + b_e$, for $i = \{1, \cdots, n\}$. s is connected through directed edges to nodes u_e and all nodes v_e are connected through directed edges to t. All edges adjacent to either s or t have zero cost and capacity n. Then, it easily follows that an optimal assignment for the original game can be obtained by computing a minimum cost flow of size n from s to t. □

Unfortunately, the next theorem rules out the possibility of obtaining optimal taxes even in simple congestion games.

Theorem 2.
a) There exists a weighted symmetric load balancing game on identical machines that does not admit ρ–pure–efficient refundable taxes with respect to the weighted total latency for any $\rho < 9/8$.
b) For any $\epsilon > 0$, there exists an unweighted symmetric load balancing game on identical machines that does not admit $(2-\epsilon)$–mixed–efficient refundable taxes.
c) There exists an unweighted load balancing game on identical machines that does not admit ρ–pure–efficient refundable taxes for any $\rho < 11/10$.
d) There exists an unweighted congestion game that does not admit ρ–pure–efficient refundable taxes for any $\rho < 6/5$.

Next, we present a universal tax function for unweighted congestion games in the sense that it does not depend at all on the congestion game; it depends only on the resources themselves.

Theorem 3. *Let $\tau = \frac{3}{2}\sqrt{3} - 2$. For any unweighted congestion game Π with linear latency functions $f_e(w) = \alpha_e w + b_e$, the function $\delta_e = \alpha_e \tau$ is a $\left(1 + \frac{2}{\sqrt{3}}\right)$–pure–efficient refundable tax for Π.*

Our next result indicates that the selection of parameter τ in Theorem 3 is the best possible.

Theorem 4. *For any $\tau \geq 0$ and $\epsilon > 0$, there exists an unweighted load balancing game for which the function $\delta_e = \alpha_e \tau$ is not $\left(1 + \frac{2}{\sqrt{3}} - \epsilon\right)$–pure–efficient refundable tax.*

In the rest of this section we construct 2-mixed-efficient refundable taxes. Given a congestion game, we use a particular assignment in order to compute the tax function. In the case of the weighted total latency, we use the solution of the quadratic program (QP1) with the convex quadratic objective function

$$g_1(x) = \sum_e \left(\alpha_e \left(\left(\sum_i x_{ie} w_i \right)^2 + \sum_i x_i w_i^2 \right) + b_e \sum_i x_{ie} w_i \right)$$

Lemma 3. *Consider a weighted congestion game Π and let x be an assignment which is the optimal solution of (QP1) with the objective function g_1. Then, the function $\delta_e(w) = \alpha_e \sum_i x_{ie} w_i$ is a 2–mixed–efficient refundable tax for Π with respect to the weighted total latency.*

Proof. We will apply inequality (1) for a mixed Nash equilibrium y of the extended game (Π, δ) and assignment x. The last term in the sum at the definition of $W(y; \Pi, \delta)$ in Lemma 1 becomes $\alpha_e \left(\sum_i x_{ie} w_i\right) \left(\sum_i y_{ie} w_i\right)$ and cancels with the first term in the sum of the right part of (1), while the last term in the sum at the right part of (1) becomes $\alpha_e \left(\sum_i x_{ie} w_i\right)^2$. So, (1) yields

$$\begin{aligned}
W(y; \Pi, 0) &\leq \sum_e \alpha_e \sum_i x_{ie} (1 - y_{ie}) w_i^2 + \sum_e \alpha_e \left(\sum_i x_{ie} w_i \right)^2 + \sum_e b_e \sum_i x_{ie} w_i \\
&\leq \sum_e \alpha_e \sum_i x_{ie} w_i^2 + \sum_e \alpha_e \left(\sum_i x_{ie} w_i \right)^2 + \sum_e b_e \sum_i x_{ie} w_i \\
&\leq \sum_e \alpha_e \sum_i x^*_{ie} w_i^2 + \sum_e \alpha_e \left(\sum_i x^*_{ie} w_i \right)^2 + \sum_e b_e \sum_i x^*_{ie} w_i \\
&\leq 2 \left(\sum_e \alpha_e \left(\sum_i x^*_{ie} w_i \right)^2 + \sum_e b_e \sum_i x^*_{ie} w_i \right) \\
&= 2 \cdot W(x^*; \Pi, 0)
\end{aligned}$$

where x^* denotes the pure assignment minimizing the weighted total latency. The last inequality follows due to integrality of x^*. □

In the case of total latency, we use the solution of the quadratic program (QP1) with the convex quadratic objective function

$$g_2(x) = \sum_e \left(\alpha_e \left(\left(\sum_i x_{ie} \right) \left(\sum_i x_{ie} w_i \right) + \sum_i x_i w_i \right) + b_e \sum_i x_{ie} \right)$$

We can show the following result; the proof is similar to the proof of Lemma 3.

Lemma 4. *Consider a weighted congestion game Π and let x be an assignment which is the optimal solution of (QP1) with the objective function g_2. Then, the function $\delta_e(w) = \alpha_j w \sum_i x_{ie}$ is a 2–mixed–efficient refundable tax for Π with respect to the total latency.*

In order to make the above two results constructive, there is a subtle point concerning the validity of the third inequality in the proofs of Lemmas 3 and 4, since, in practice, the solution of the quadratic program has not perfect accuracy. As in [2], we can guarantee the validity of this inequality by making the accuracy parameter sufficiently small. As a corollary we obtain the following statement.

Theorem 5. *There exist polynomial time algorithms for computing 2-mixed-efficient refundable taxes with respect to the total latency and the weighted total latency in weighted congestion games.*

4 Non-refundable Taxes

In this section, we consider non-refundable taxes; we first focus on efficient non–refundable taxes with respect to the weighted total cost. A recent lower bound of $5/2$ on the price of anarchy of weighted load balancing games on identical machines [5] implies that the trivial tax function is not $(5/2 - \epsilon)$-pure-efficient for any $\epsilon > 0$. This lower bound can be modified so that resource removal cannot improve the price of anarchy either. We show that better non-refundable taxes do exist. Here, the corresponding tax function uses an optimal assignment. Unfortunately, even computing an approximate such assignment is hard [4]. We can use a PTAS from [4] to show a slightly worse constructive result when the number of machines is constant. We note that the lower bound in [5] uses a constant number of machines.

Theorem 6. *Any weighted load balancing game on identical machines admits $\left(1 + \sqrt{2}\right)$–mixed–efficient non–refundable taxes with respect to the weighted total cost.*

The proof of Theorem 6 uses the tax function

$$\delta_e(w) = \begin{cases} \sum_i x_{ie} w_i - w, & \text{if } \sum_i x_{ie} w_i \geq w; \\ 0, & \text{otherwise} \end{cases}$$

for a load balancing game Π with latency function of the form $f(w) = x + b$, where x denotes a pure assignment for Π that minimizes the weighted total latency.

In order to compute efficient non-refundable taxes with respect to the total cost, we use solutions to the quadratic program (QP1) with the objective function

$$g_3(x) = \sum_e \left(\alpha_e \left(\sum_i x_{ie} \right) \left(\sum_i x_{ie} w_i \right) + b_e \sum_i x_{ie} \right)$$

Ideally, we would like to use optimal pure assignments, i.e., an optimal integral solution x^* of (QP1) with the objective function g_3. However, even approximate semi-pure assignments can be used to obtain efficient non-refundable taxes.

Lemma 5. *Consider a weighted congestion game Π and let x be a semi–pure assignment with $g_3(x) \leq \rho \cdot g_3(x^*)$. Then, the function*

$$\delta_e(w) = \begin{cases} \alpha_e \left(2 \sum_i x_{ie} - 1\right) w, & \text{if } 2 \sum_i x_{ie} \geq 1 \\ 0, & \text{otherwise} \end{cases}$$

is a 4ρ–mixed–efficient non–refundable tax for Π with respect to the total cost.

Hence, by applying Lemma 5 for $\rho = 1$ we obtain the following existential result.

Corollary 1. *Any weighted congestion game admits 4–mixed–efficient non–refundable taxes with respect to the total cost.*

Next, we show how to compute efficient semi–pure assignments to obtain a slightly worse constructive result. We first solve the quadratic program (QP1) with the convex quadratic objective function

$$g_4(x) = \sum_e \left(\alpha_e \left(\frac{1}{2} + \sum_i x_{ie} \right) \left(\sum_i x_{ie} w_i \right) + b_e \sum_i x_{ie} \right)$$

Then, we obtain a half–integral solution $\hat{x}$ by applying randomized rounding to the solution x as follows. For each i, we use a die with one face for each $p \in \mathcal{P}_i$ such that $x_{ip} > 0$ and a probability of x_{ip} associated with the face corresponding to p. We cast the die twice and let p_1 and p_2 be the actions corresponding to the outcomes. If $p_1 = p_2$, we set $\hat{x}_{ip_1} = 1$, while if $p_1 \neq p_2$, we set $\hat{x}_{ip_1} = \hat{x}_{ip_2} = \frac{1}{2}$; we also set $\hat{x}_{ip} = 0$ for each $p \in \mathcal{P}_i \setminus \{p_1, p_2\}$. We also set $\hat{x}_{ie} = \sum_{p \in \mathcal{P}_i} \hat{x}_{ip}$.

Lemma 6. $E\left[g_3(\hat{x})\right] \leq \frac{3}{2} g_3(x^*)$

By using standard probabilistic arguments, we can guarantee that $g_3(\hat{x}) \leq \left(\frac{3}{2} + \epsilon\right) g_3(x^*)$ for any $\epsilon > 0$ by executing the randomized rounding procedure polynomially many times. Hence, Lemma 5 yields the following.

Theorem 7. *There exists a polynomial time algorithm for computing $(6 + \epsilon)$-mixed-efficient non-refundable taxes with respect to the total cost in weighted congestion games.*

5 Open Problems

Our work reveals several interesting open questions. Tightening the bounds for pure-efficient refundable taxes is a challenging task. In particular, extending the results of Theorem 1 and determining the subclass of unweighted congestion games that admit pure-optimal taxes is one of them. The candidate class is

that of the unweighted symmetric congestion games which include network congestion games with a single source and a single destination. The existence of efficient non-trivial universal tax functions for weighted congestion games is also open. We conjecture that such taxes do not exist. For non-refundable taxes, the question whether efficient non-trivial taxes for congestion games with respect to the weighted total cost exist is still open. Special cases as simple as unweighted symmetric load balancing are interesting as well. Here, besides the trivial upper bound, we have a preliminary statement that better than 27/23-pure-efficient non-refundable taxes do not exist. We point out that symmetry has not helped so far, since all our lower bounds are in a sense symmetric constructions. The impact of symmetry of games to the existence of efficient taxes needs further investigation. Complexity issues are also very interesting, i.e., given a congestion game Π, how easy is to compute a ρ-mixed/pure-efficient (non)-refundable tax for this particular game? Our results can be thought of as approximation algorithms for this optimization problem. Although we have made no attempt to formally prove this statement, we strongly believe that this problem is computationally hard for some constant $\rho > 1$. Another open problem is to prove bounds on the cost of taxes that force at least one nearly-optimal assignment to become an equilibrium. This is related to the study of the price of stability [5, 7]. Also, having players with different sensitivities to taxes as in the model of [8, 10, 15] is another interesting extension of our model. Finally, it is worth investigating taxes for congestion games with more general (e.g., polynomial) latency functions.

References

1. B. Awerbuch, Y. Azar, and A. Epstein. The price of routing unsplittable flow. In *Proc. of the 37th Annual ACM Symposium on Theory of Computing (STOC '05)*, pp. 57-66, 2005.
2. Y. Azar and A. Epstein. Convex programming for scheduling unrelated parallel machines. In *Proc. of the 37th Annual ACM Symposium on Theory of Computing (STOC '05)*, pp. 331-337, 2005.
3. Y. Azar and A. Epstein. The hardness of network design for unsplittable flow with selfish users. In *Proc. of the 3rd Workshop on Approximation and Online Algorithms (WAOA '05)*, pp. 41-54, 2005.
4. Y. Azar, L. Epstein, Y. Richter, and G. Woeginger. All-norm approximation algorithms. *Journal of Algorithms*, 52(2): 120-133, 2004.
5. I. Caragiannis, M. Flammini, C. Kaklamanis, P. Kanellopoulos, and L. Moscardelli. Tight bounds for selfish and greedy load balancing. In *Proc. of the 33rd International Colloquium on Automata, Languages and Programming (ICALP '06)*, LNCS 4051, Springer, Part I, pp. 311-322, 2006.
6. G. Christodoulou and E. Koutsoupias. The price of anarchy of finite congestion games. In *Proc. of the 37th Annual ACM Symposium on Theory of Computing (STOC '05)*, pp. 67-73, 2005.
7. G. Christodoulou and E. Koutsoupias. On the price of anarchy and stability of correlated equilibria of linear congestion games. In *Proc. of the 13th Annual European Symposium on Algorithms (ESA '05)*, LNCS 3669, Springer, pp. 59-70, 2005.

8. R. Cole, Y. Dodis, and T. Roughgarden. Pricing network edges for heterogeneous selfish users. In *Proc. of the 35th Annual ACM Symposium on Theory of Computing (STOC '03)*, pp. 521-530, 2003.
9. A. Czumaj and B. Vöcking. Tight bounds for worst-case equilibria. In *Proc. of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '02)*, pp. 413-420, 2002.
10. L. Fleischer, K. Jain, and M. Mahdian. Tolls for heterogeneous selfish users in multicommodity networks and generalized congestion games. In *Proc. of the 45th Annual IEEE Symposium on Foundations of Computer Science (FOCS '04)*, pp. 277-285, 2004.
11. D. Fotakis, S. Kontogiannis, E. Koutsoupias, M. Mavronicolas and P. Spirakis. The structure and complexity of Nash equilibria for a selfish routing game. In *Proc. of the 29th International Colloquium on Automata, Languages and Programming (ICALP '02)*, LNCS 2380, Springer, pp. 123-134, 2002.
12. D. Fotakis, S. Kontogiannis, and P. Spirakis. Selfish unsplittable flows. *Theoretical Computer Science*, 340(3): 514-538, 2005.
13. M. Gairing, T. Lücking, M. Mavronicolas and B. Monien. Computing Nash equilibria for scheduling on restricted parallel links. In *Proc. of the 36th Annual ACM Symposium on Theory of Computing (STOC '04)*, pp. 613-622, 2004.
14. M. Gairing, B. Monien, and K. Tiemann. Routing (un-) splittable flow in games with player-specific latency functions. In *Proc. of the 33rd International Colloquium on Automata, Languages and Programming (ICALP '06)*, LNCS, Springer, 2006, to appear.
15. G. Karakostas, and S. Kolliopoulos. Edge pricing of multicommodity networks for heterogeneous selfish users. In *Proc. of the 45th Annual IEEE Symposium on Foundations of Computer Science (FOCS '04)*, pp. 268-276, 2004.
16. E. Koutsoupias, M. Mavronicolas and P. Spirakis. Approximate equilibria and ball fusion. *Theory of Computing Systems*, 36(6): 683-693, 2003.
17. E. Koutsoupias and C. Papadimitriou. Worst-case equilibria. In *Proc. of the 16th International Symposium on Theoretical Aspects of Computer Science (STACS '99)*, LNCS 1563, Springer, pp. 404-413, 1999.
18. T. Lücking, M. Mavronicolas, B. Monien, and M. Rode. A new model for selfish routing. In *Proc. of the 21st International Symposium on Theoretical Aspects of Computer Science (STACS '04)*, LNCS 2996, Springer, pp. 547-558, 2004.
19. M. Mavronicolas and P. Spirakis. The price of selfish routing. In *Proc. of the 33rd Annual ACM Symposium on Theory of Computing (STOC '01)*, pp. 510-519, 2001.
20. I. Milchtaich. Congestion games with player-specific payoff functions. *Games and Economic Behavior*, 13: 111-124, 1996.
21. C. Papadimitriou. Algorithms, games and the internet. In *Proc. of the 33rd Annual ACM Symposium on Theory of Computing (STOC '01)*, pp. 749-753, 2001.
22. R. Rosenthal. A class of games possessing pure-strategy Nash equilibria. *International Journal of Game Theory*, 2: 65-67, 1973.
23. T. Roughgarden and E. Tardos. How bad is selfish routing? *Journal of the ACM*, 49(2): 236-259, 2002.
24. S. Suri, C. Tóth and Y. Zhou. Selfish load balancing and atomic congestion games. In *Proc. of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '04)*, pp. 188-195, 2004.

Spanners with Slack*

T.-H. Hubert Chan**, Michael Dinitz***, and Anupam Gupta

Carnegie Mellon University

Abstract. Given a metric (V, d), a *spanner* is a sparse graph whose shortest-path metric approximates the distance d to within a small multiplicative distortion. In this paper, we study the problem of *spanners with slack*: e.g., can we find sparse spanners where we are allowed to incur an arbitrarily large distortion on a small constant fraction of the distances, but are then required to incur only a constant (independent of n) distortion on the remaining distances? We answer this question in the affirmative, thus complementing similar recent results on embeddings with slack into ℓ_p spaces. For instance, we show that if we ignore an ϵ fraction of the distances, we can get spanners with $O(n)$ edges and $O(\log \frac{1}{\epsilon})$ distortion for the remaining distances.

We also show how to obtain sparse and low-weight spanners with slack from existing constructions of conventional spanners, and these techniques allow us to also obtain the best known results for distance oracles and distance labelings with slack. This paper complements similar results obtained in recent research on slack embeddings into normed metric spaces.

1 Introduction

The study of metric embeddings has been a central pursuit in algorithms research in the past decade: an embedding is a map from a metric space into a "simpler" metric space so that distances between points are changed by at most a small factor. More formally, given a *target class* $\mathcal{C}$ of metrics, an *embedding* of a finite metric space $M = (V, d)$ into the class $\mathcal{C}$ is a new metric space $M' = (V, d')$ such that $M' \in \mathcal{C}$. Most of the work on embeddings has used *distortion* as the fundamental measure of quality; the distortion of an embedding is the worst multiplicative factor by which distances are increased by the embedding[1]. Given

* Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213. This research was partly supported by the NSF CAREER award CCF-0448095, and by an Alfred P. Sloan Fellowship.

** Supported partly by a fellowship from the Croucher Foundation.

*** Supported partly by an NSF Graduate Research Fellowship and an ARCS Scholarship.

[1] Formally, for an embedding $\varphi : (V, d) \to (V, d')$, the *distortion* is the smallest D so that $\exists\, \alpha, \beta \geq 1$ with $\alpha \cdot \beta \leq D$ such that $\frac{1}{\alpha}\, d(x, y) \leq d'(\varphi(x), \varphi(y)) \leq \beta\, d(x, y)$ for all pairs $x, y \in V \times V$. Note that this definition of distortion is no longer invariant under arbitrary scaling, since $\alpha, \beta \geq 1$; however, this is merely for notational convenience, and all our results can be cast in the usual definitions of distortion.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 196–207, 2006.

the metric $M = (V, d)$ and the class $\mathcal{C}$, one natural goal is to find an embedding $\varphi((V, d)) = (V, d') \in \mathcal{C}$ such that the distortion of the map φ is minimized. Note that this notion of embedding is slightly non-standard but very natural, as it not only captures embeddings of metric spaces into geometric spaces (e.g., when $\mathcal{C}$ is the class of all Euclidean metrics, or the class of all ℓ_1 metrics), but also concepts such as *sparse spanners*, where the class $\mathcal{C}$ is the class of metrics generated by sparse graphs. In the rest of the paper, we will talk about metric embeddings in this general sense, thus including within its purview the results on spanners as well as those on embeddings into normed spaces, or for that matter, into distributions over tree metrics [5, 15]. (As an aside, let us note that the concept of distortion is often called "stretch" in the spanners literature, and we use the two terms interchangeably.)

In the theoretical community, the popularity of the notion of distortion/stretch has been driven by its applicability to approximation algorithms: if the embedding $\varphi : (V, d) \to (V, d')$ has a distortion of D, then the cost of solutions to some optimization problems on (V, d) and on (V, d') can only differ by some function of D; this idea has led to numerous approximation algorithms [20]. Seminal results in embeddings include the $O(\log n)$ distortion embeddings of arbitrary metrics into ℓ_p spaces [9], the fact that any metric admits an $O(\log n)$ stretch spanner with $O(n)$ edges [3], and that any metric can be embedded into a distribution of trees with distortion $O(\log n)$ [15]. (All the above three results are known to be tight.)

In parallel to this theoretical work, more applied communities have had much recent interest in embeddings (and more generally, but also somewhat vaguely, on problems of finding "simpler representations" of distance spaces). One example is the networking community, where there is much interest in taking the point-to-point latencies between nodes in a network, treating it as a metric space $M = (V, d)$ satisfying the triangle inequality,[2] and finding some simpler representation $M' = (V, d')$ of this resulting metric so that distances between nodes can be quickly and accurately computed in this "simpler" metric M'. (E.g., one may want to assign each node a short *label* so that the distance between two nodes can be inferred approximately by merely looking at their labels.)

Despite this similarity of interest, many of the theoretical results mentioned above have not been used widely in these applications; the logarithmic guarantees on the distortion are often deemed unacceptable. Indeed, the notion of distortion turns out to be a demanding and inflexible objective function, and the empirical works are often happy with guarantees of the following form: they allow some small fraction of the distances to be distorted by *arbitrary* amounts, but then seek very strong guarantees on the distortion incurred by the remaining large fraction of the distances. E.g., in the networking application above, we would be happy if *most* inter-node distances were correct and only a small fraction of distances would be estimated poorly. (This corresponds to some notion of being "good on average"; we will revisit this idea in Section 4.)

[2] While the triangle inequality can be violated by network latencies, empirical evidence suggests that these violations are small and/or infrequent enough to make metric methods a useful approach.

To remedy the situation, Kleinberg et. al. [21] defined the notion of *embeddings with slack*: in addition to the metric $M = (V, d)$ and the class $\mathcal{C}$ in the initial formulation above, we are also given a *slack parameter* ϵ. We now want to find a map $\varphi(M) = (V, d') \in \mathcal{C}$ whose distortion is bounded by some quantity $D(\epsilon)$ on all but an ϵ fraction of the pairs of points in $V \times V$. Note that we allow the distortion on the remaining ϵn^2 pairs of points to be arbitrarily large. The line of work starting with their paper, and furthered by Abraham et. al. [2] and [1] showed that very strong results were indeed possible: in fact, when allowed constant slack, one could get constant-distortion constant-dimensional embeddings!

Given these results for embeddings into normed spaces, it is natural to ask whether one can obtain similar results for spanners (and related constructs such as distance oracles and distance labelings). This paper studies spanners with slack, and gives strong guarantees that answer the question in the affirmative and complement the above results for embeddings into normed metric spaces.

1.1 Our Results

In this paper, we look at results on finding *spanners* when we are allowed to incur an arbitrary amount of distortion on an ϵ fraction of the distances. We say that $H = (V, E_H)$ is an *ϵ-slack spanner* of a metric (V, d) with distortion D if for each vertex $v \in V$, the furthest $(1-\epsilon)n$ vertices w from v satisfy $d(v, w) \leq d_H(v, w) \leq D\, d(v, w)$, i.e., the graph H maintains distances from each vertex to all but the closest ϵn vertices. Our first result is a general transformation procedure to convert standard constructions of spanners into spanners with slack.

Theorem 1 (General Conversion Theorem). *Suppose any metric admits an $\alpha(n)$-distortion spanner with $T(n)$ edges. Then given any metric (V, d) and any ϵ, we can find an ϵ-slack spanner H_ϵ for it with $O(\alpha(\frac{1}{\epsilon}))$-distortion and $n + T(\frac{1}{\epsilon})$ edges.*

Using this, it immediately follows that there are constant-slack spanners with linear number of edges and *constant* distortion!

Moreover, if we were given a graph $G = (V, E)$ whose shortest path metric is (V, d), we show in Section 3.1 how to extend the above theorem to output a *subgraph* of G that has slightly more edges. We can also extend Theorem 1 in another direction and find a spanner that has a small weight in addition to a small number of edges (see Section 3.2).

Note that in the above results, we are given an ϵ, and then output a spanner H_ϵ. We can do better, and output a *single* graph H such that it is an ϵ-slack spanner for *all ϵ simultaneously*. Two general conversion procedures are given in Section 4, which can be used to prove corollaries like:

Theorem 2 ("One Spanner for All Epsilons"). *Given any metric of size n, we can find a graph H with $O(n)$ edges that is an $O(\log \frac{1}{\epsilon})$-distortion ϵ-slack spanner for each ϵ. Moreover, if the metric is generated by a graph G, then H can be made a subgraph of G.*

The spanner H from the previous theorem preserves distances well on average. We consider two natural notions of average, which are defined in Section 2.

Theorem 3 ("Good on Average"). *The spanner H from Theorem 2 has $O(1)$ average distortion and $O(1)$ distortion-of-the-average, and moreover has $O(\log n)$ distortion in the worst case.*

We then turn our attention to the questions of constructing distance labelings and distance oracles, which are useful for resource-location applications mentioned in the introduction. Detailed results that appear in Section 5 include results like:

Theorem 4 (Labelings and Oracles). *Given any integer k, every metric admits ϵ-slack distance oracles where the query time and stretch are $O(k)$, and the space requirement is $O(n + k(\frac{1}{\epsilon})^{1+1/k})$ words. Moreover, there are ϵ-slack distance labeling schemes that uses $O((\frac{1}{\epsilon})^{1/k} \log^{1-1/k} \frac{1}{\epsilon})$ space and suffer distortion $O(k)$.*

1.2 Previous Work

This work builds on a large body of work on spanners dating back to the late 1980's [24, 3, 11, 4, 27] and still going strong [14, 8, 7, 6, 28]; see, e.g., [23] for many of the results. Spanners were initially studied for applications in network synchronization, but since then they have found myriad uses in network design and routing, as well as in many places where it is advisable to compactly store a graph without changing the distances much, such as in speeding up shortest path computations. Apart from the literature on finding spanners of general graphs, there has also been a large body of work on Euclidean spanners (see, e.g., [11, 4]), as well as work on spanners for doubling metrics [10, 19].

The study of distance labelings of graphs [22, 16, 25] requires assigning "short" labels to vertices so that the distance between two vertices can be inferred from their labels alone, without any additional information about the graph. It is known that if one wants to infer distances exactly, then one may require as many as n bits for each vertex; however, one can do with far less space if one just wants to estimate the distances approximately. A closely related concept is that of a *distance oracle*, which is a data structure that can be used to estimate distances between nodes using small space and fast query time. Exact distance oracles require one to store lots of information (e.g., the entire distance matrix) or large query time (to run a shortest-path computation), but fast and compact distance oracles for general graphs were given by Thorup and Zwick [27]; work on special classes of graphs appears in [26, 12, 18, 17].

As mentioned in the introduction, the work on embeddings with slack was initiated by Kleinberg et. al. [21], and many of the subsequent results were improved in Abraham et. al. [2] and [1]. Our techniques and results complement those in the two aforementioned papers. The notion of slack distance oracles has been studied in [1] and we extend some of the results marginally in Section 5.1.

Our work is closely related to the work on distance preservers [13] by Coppersmith and Elkin. A distance preserver is a subgraph that maintains distances *exactly* for a pre-specified subset of pairs of nodes, as opposed to our case in which the only guarantee is that a large fraction of pairs of nodes have their distances well-approximated. On the other hand, the work of Elkin and Peleg [14] shows that there exist sparse spanners such that for large distances that are at least logarithmically large, the multiplicative stretch can be close to 1. This agrees with our results in the sense that large distances can be better maintained than small distances.

2 Preliminaries and Notation

All metric spaces we consider in this paper are finite and the graphs we consider are undirected. Let (V, d) be a metric space, where $n = |V|$. The *ball* $B(x, r) = \{y \mid d(x, y) \leq r\}$ is the set of points at distance at most r from x. For $0 < \epsilon < 1$, $R(x, \epsilon)$ is the minimum distance r such that $|B(x, r)| \geq \epsilon n$. The point y is *ϵ-far* away from point x if $d(x, y) \geq R(x, \epsilon)$. Observe that all spanners H we consider are *non-contracting*; i.e., for any $x, y \in V$, $d(x, y) \leq d_H(x, y)$.

Definition 1 ((Uniform) Slack Spanner). *Given a metric (V, d) and $0 < \epsilon < 1$, a weighted graph $H = (V, E)$ with each edge $(u, v) \in E$ having weight $d(u, v)$ is an* α-spanner with ϵ-uniform slack *if for all $x, y \in V$ such that y is ϵ-far away from x, $d_H(x, y) \leq \alpha \cdot d(x, y)$. In general, α can be a function of ϵ and $|V|$. If the metric (V, d) is induced by some weighted graph G, we say that H is a* subgraph spanner *if H is a subgraph of G.*

In other words, an ϵ-uniform slack spanner is one such that for each point x, apart from the ϵn points closest to x, the distances from x to the rest of the points are well approximated. We call this concept "*uniform* slack" to be consistent with previous notation; all references to "ϵ-slack" in this paper mean "ϵ-uniform slack".[3]

Definition 2 (Gracefully degrading spanner). *A weighted graph H is an* $\alpha(\frac{1}{\epsilon})$-gracefully degrading spanner *for the metric (V, d) if for each $0 < \epsilon < 1$, H is an $\alpha(\frac{1}{\epsilon})$-spanner with ϵ-slack. The notion of subgraph spanner also applies analogously.*

We also consider two incomparable notions of "average" distortion; both have been considered previously in the literature, and we will construct spanners that are simultaneously good with respect to both these notions.

Definition 3 (Average Distortion). *The* average distortion *of a spanner H for a metric space (V, d) is* $\frac{1}{\binom{n}{2}} \sum_{\{x,y\} \in \binom{V}{2}} \frac{d_H(x,y)}{d(x,y)}$.

[3] For the record, there *is* a non-uniform notion of slack; see [2, Defn. 1.1] for details. Also, readers of [1] should note that ϵ-uniform slack embeddings are called "coarsely $(1 - \epsilon)$ partial embeddings" in that paper.

Definition 4 (Distortion of Averages). *The* distortion of averages *of a spanner H for a metric space (V, d) is* $\sum_{\{x,y\}\in\binom{V}{2}} d_H(x,y) / \sum_{\{x,y\}\in\binom{V}{2}} d(x,y)$.

Most of our algorithms make use of a small sample of points from the metric space V such that each point is "close" to some sample point:

Definition 5 (Density Net). *Given a metric space (V, d) with $n = |V|$, and $0 < \epsilon < 1$, an* ϵ-density net *is a set $N \subseteq V$ such that **(1)** for all $x \in V$, there exists $y \in N$ such that $d(x, y) \leq 2R(x, \epsilon)$, and **(2)** $|N| \leq \frac{1}{\epsilon}$.*

We will often refer to the nodes in N as *centers*. Note that the difference between an ϵ-net and an ϵ-density net is in the notion of "closeness"—here the allowed distance from x to its closest center depends on the density of points around x.

Lemma 1. *Given a metric space (V, d) and $0 < \epsilon < 1$, an ϵ-density net N can be found in polynomial time.*

Proof. For each point $x \in V$, let B_x denote the ball $B(x, R(x, \epsilon))$. We order the vertices in a list L by nondecreasing value of $R(\cdot, \epsilon)$, breaking ties arbitrarily, and initialize the set N to be empty. We remove the first vertex v from list L. If there exists $u \in N$ such that B_v intersects B_u, then we just discard v; otherwise, we add v to N and remove all vertices in the ball B_v from the list L. We repeat this process until the list L becomes empty and return N as our ϵ-density net.

We next show that the subset N returned satisfies the two properties given in Definition 5. Consider any point $x \in V$. We show that there is a point $y \in N$ within distance $2R(x, \epsilon)$ of x. If x is included in N, this is trivially true. Otherwise, either x was at some point the first vertex in list L and get discarded, or x was in some ball B_v and removed from list L. In the former case, there is some point $u \in N$ such that B_u intersects B_x. Since u appears before x in the initial list, $R(u, \epsilon) \leq R(x, \epsilon)$ and hence the distance between x and the density-net point u is $d(u, x) \leq R(u, \epsilon) + R(x, \epsilon) \leq 2R(x, \epsilon)$. In the latter case, as v appear before x in the initial list, we also have $R(v, \epsilon) \leq R(x, \epsilon)$ and so $d(x, v) \leq R(v, \epsilon) \leq R(x, \epsilon) \leq 2R(x, \epsilon)$. To show that $|N| \leq \frac{1}{\epsilon}$, observe that the intersection of B_x and B_y is empty for any two distinct points $x, y \in N$. Since for each $x \in N$, the ball B_x contains at least ϵn points, we conclude that $|N| \leq \frac{1}{\epsilon}$.

3 Slack Spanners

In this section, we give a general transformation technique to convert $\alpha(n)$-spanners with $T(n)$ edges into ϵ-slack spanners with distortion $(5 + 6\alpha(\frac{1}{\epsilon}))$ and $n + T(\frac{1}{\epsilon})$ edges. Our construction is very simple:

> **Construction.** We first construct an ϵ-density net N as given in Lemma 1. Since $|N| \leq \frac{1}{\epsilon}$, we can construct an $\alpha(\frac{1}{\epsilon})$-spanner $\widehat{H}$ for the set of centers N. Then, for each point $x \in X \setminus N$, we add an edge between x and its closest point in N to $\widehat{H}$; this gives us a spanner H for (V, d).

Theorem 5. *The spanner H has $n + T(\frac{1}{\epsilon})$ edges, and is a $(5 + 6\alpha(\frac{1}{\epsilon}))$-spanner with ϵ-uniform slack.*

Proof. First we bound the size of H. Since N has at most $\frac{1}{\epsilon}$ points, the spanner $\widehat{H}$ has at most $T(\frac{1}{\epsilon})$ edges. Moreover, for each point $x \in V \backslash N$, one extra edge is added. Hence, H has at most $n + T(\frac{1}{\epsilon})$ edges.

Next, we bound the stretch of H. Consider two points u and v such that v is ϵ-far away from u, i.e., $d(u, v) \geq R(u, \epsilon)$. Let u' be a closest point in N to which u is connected to in H (or set $u' = u$ if u is in N), and define v' similarly with respect to v. By the properties of the density net, the distance $d(u, u') \leq 2R(u, \epsilon) \leq 2d(u, v)$ and $d(v, v') \leq d(v, u') \leq d(v, u) + d(u, u') \leq 3d(u, v)$. Also, $d(u', v') \leq d(u', u) + d(u, v) + d(v, v') \leq 6d(u, v)$. This implies that

$$\begin{aligned} d_H(u, v) &\leq d(u, u') + d_H(u', v') + d(v', v) \leq 5d(u, v) + d_H(u', v') \\ &\leq 5d(u, v) + \alpha(\tfrac{1}{\epsilon})d(u', v') \leq 5d(u, v) + \alpha(\tfrac{1}{\epsilon})(6d(u, v)). \end{aligned}$$

As an example of how we apply Theorem 5, let us recall a well-known result about spanners for general metrics, from which we derive Corollary 1.

Theorem 6 (Spanners for general metrics [24, 3]). *For any metric of size n, there exists a $(2k - 1)$-spanner with $O(n^{1+1/k})$ edges.*

Corollary 1 (Uniform slack spanners for general metrics). *For any metric, for any $0 < \epsilon < 1$, for any integer $k > 0$, there exists a $(12k - 1)$-spanner with ϵ-uniform slack of size $n + O((\frac{1}{\epsilon})^{1+1/k})$.*

3.1 Subgraph Spanners

Note that if the metric (V, d) was generated by a graph $G = (V, E)$, our previous construction may result in a spanner that is not a subgraph of the original graph G. We now give an alternative construction to obtain a subgraph spanner. Let us first recall a theorem by Coppersmith and Elkin [13] on subgraphs that preserve distances exactly for a given set P of pairs of vertices in a weighted graph $G = (V, E)$.

Theorem 7 ([13]). *Given a weighted graph $G = (V, E)$ and a set P of pairs of vertices, there exists a subgraph H of G with $O(n + \sqrt{n} \cdot |P|)$ edges such that for any $\{u, v\} \in P$, $d_H(u, v) = d_G(u, v)$.*

Construction of the Subgraph Spanner. As before, let N be an ϵ-density net, which we know has at most $\frac{1}{\epsilon}$ elements. We construct an $\alpha(\frac{1}{\epsilon})$-spanner H' of size $T(\frac{1}{\epsilon})$ on N, which we convert to a subgraph in the following manner. We take P to be the set of distinct pairs $\{u, v\}$ that are edges in H' to be the subgraph of G that preserves distances for all pairs in P in the manner as stated in Theorem 7. Finally, points in V are connected to N by shortest path trees rooted at the points in N, using edges in the given graph G.

Theorem 8 shows that the resulting subgraph spanner H contains a small number of edges and has small stretch. Applying the theorem to Theorem 6 gives a nice corollary.

Theorem 8. *The subgraph H is a $(5+6\alpha(\frac{1}{\epsilon}))$-spanner with ϵ-uniform slack and has $O(n + \sqrt{n} \cdot T(\frac{1}{\epsilon}))$ edges.*

Corollary 2 (Subgraph uniform slack spanners for general metrics). *For any metric, for any $0 < \epsilon < 1$, for any integer $k > 0$, there exists a subgraph $(12k-1)$-spanner with ϵ-uniform slack of size $O(n + \sqrt{n} \cdot (\frac{1}{\epsilon})^{1+1/k})$.*

3.2 Low Weight Spanners

In some cases, we would like spanners which are not only sparse, but whose weight is also comparable to the weight of an MST on the metric (V, d). Due to lack of space, we merely mention a representative result here, and omit the details.

Proposition 1 (Low weight uniform slack spanner). *For any metric of size n, there exists an $O(\log \frac{1}{\epsilon})$-spanner with ϵ-uniform slack of size $O(n + \frac{1}{\epsilon})$ and weight $O(\log^2 \frac{1}{\epsilon})$ times that of an MST.*

4 Gracefully Degrading Spanners and Notions of Average Distortion

In this section, we give general procedures to convert ordinary spanners into gracefully degrading spanners. Suppose we know how to construct ordinary $\alpha(n)$-spanners of size $T(n)$ for finite metrics of size n. Observe that typically, $\alpha(\cdot)$ is a sublinear function, such as $O(\log n)$. In particular, we assume that there exists $C, c > 1$ such that $\alpha(n) \leq C\alpha(n^{1/c})$.

> **Construction.** Take $\epsilon_0 = n^{-1/c}$ (think of c as 2 or 4) and construct a 1-spanner H_0 for some ϵ_0-density net N_0 that has $O(n)$ edges. We also construct an $\alpha(n)$-spanner $\widehat{H}$ for the entire metric V. The gracefully degrading spanner consists of the union of $\widehat{H}$ and H_0, together with edges that connect each point in V to its closest point in N_0.

This simple construction gives us the following theorem on gracefully degrading spanners. Using similar techniques as in Section 3.1, we can obtain subgraph spanners as well. Applying Theorem 9 to Theorem 6 with $k = O(\log n)$, we obtain the result as promised in the introduction.

Theorem 9. *Suppose there exists an $\alpha(n)$-spanner of size $T(n)$ for any metric of size n, where $\alpha(\cdot)$ is a non-decreasing function such that there exists exists $C > 1$ such that $\alpha(n) \leq C\alpha(n^{1/2})$. Then, for any finite metric (V, d) of size n, there exists an $C\alpha(\frac{1}{\epsilon})$-gracefully degrading spanner of size at most $T(n) + O(n)$.*

If we have the stronger assumption that $\alpha(n) \leq C\alpha(n^{1/4})$, then the gracefully degrading spanner can be made to be a subgraph of the weighted graph that induces the metric (V, d).

Corollary 3 (Gracefully degrading spanner for general metrics). *Any metric of size n has a $O(\log \frac{1}{\epsilon})$-gracefully degrading spanner H of size $O(n)$. If the metric is induced by some weighted graph G, then H can be made to be a subgraph of G.*

Since the greedy construction given in [24, 3] gives an $O(\log n)$-spanner with size $O(n)$ for any metric, we can show that any metric has a spanner that has $O(1)$ "average distortion" for both notions of average distortion given in Definitions 3 and 4 in the following theorem.

Theorem 10 ("Average Distortion"). *For any metric (V, d), there exists a spanner H with size $O(n)$ that has both constant average distortion and constant distortion of the average, and moreover has $O(\log n)$ stretch in the worst case. If the metric (V, d) is induced by some graph G, then H can be made to be a subgraph of G.*

5 Distance Oracles and Labelings

The techniques that we have developed for slack spanners also turn out to be useful for developing slack distance oracles and distance labelings. Distance oracles and labelings have been widely studied, and distance labelings were in fact one of the original motivations for the study of slack embeddings by Kleinberg et. al. [21]. Slack distance oracles and (implicitly) labelings were considered by Abraham et. al. [1], who gave both slack and gracefully degrading constructions. We give simple constructions with slightly better bounds for distance oracles, and give the first uniform slack labelings that do not use an embedding into ℓ_p, allowing us to bypass a lower bound from Abraham et al. [2].

5.1 Distance Oracles

Thorup and Zwick [27] studied the problem of creating distance oracles for metric spaces. A distance oracle is a small data structure which allows fast queries for approximate distances. They give an oracle that, for any integer $k \geq 1$, takes $O(kn^{1+1/k})$ space, has $O(k)$ query time, and has stretch of $2k-1$. Slack distance oracles were first studied by Abraham et. al. [1], whose results we improve on for both uniform slack and gracefully degrading distance oracles. We first give a general transformational theorem. By using this transformation on the distance oracle of Thorup and Zwick [27, Theorem 3.1], we get a uniform slack distance oracle with the best known guarantee.

Theorem 11. *Suppose that there exists some distance oracle with $\alpha(n)$ stretch and $O(q(n))$ query time that uses $O(f(n))$ space. Then there exists a distance oracle with ϵ-uniform slack, $5 + 6\alpha(\frac{1}{\epsilon})$ stretch, and $O(q(\frac{1}{\epsilon}))$ query time that uses $O(n + f(\frac{1}{\epsilon}))$ space.*

Corollary 4 (Uniform slack distance oracle). *For every integer $k \geq 1$, there is a distance oracle with ϵ-uniform slack, $O(k)$ query time, and $12k-1$ stretch that uses $O(n + k(\frac{1}{\epsilon})^{1+1/k})$ space.*

The uniform slack distance oracle in the full version of Abraham et. al. [1] has stretch of only $6k-1$ and $O(k)$ query time, but uses $O(n \log n \log \frac{1}{\epsilon} + k \log n (\frac{1}{\epsilon} \log \frac{1}{\epsilon})^{1+1/k})$ space.

Gracefully Degrading Distance Oracles. The construction of gracefully degrading spanners in Section 4 can be easily modified to yield gracefully degrading distance oracles. Namely, we use two levels of distance oracles instead of two levels of spanners, where the oracle on the density net is exact. Combining this transformation theorem with the distance oracles of Thorup and Zwick [27] and the average case analysis of Theorem 10, we get the following:

Corollary 5 (Gracefully degrading distance oracle). *For any integer k with $1 \leq k \leq O(\log n)$, there is a distance oracle with worst case stretch of $2k-1$ and $O(k)$ query time that uses $O(kn^{1+1/k})$ space such that the average distortion and the distortion of average is $O(1)$.*

The gracefully degrading distance oracle of Abraham et. al. [1, Theorem 14] gives the same query time, worst case stretch, average distortion, and distortion of average. However, their construction modifies the standard Thorup and Zwick [27] construction by sampling the first level with probability $3n^{-1/k} \ln n$, and thus they use $O(n^{1+1/k} \log n)$ space. This is more than we use if $k = o(\log n)$ and the same if $k = \Theta(\log n)$.

5.2 Distance Labels

A distance labeling is an assignment of labels to the vertices so that the approximate distance between any two vertices can be computed simply by looking at the two labels. The goals are to minimize the stretch, the size of the label, and the time needed to compute the distance given the two labels. We give the first uniform slack distance labeling that uses space independent of n. Note that any embedding of a metric into ℓ_p gives a distance labeling where the size of a label is the dimension of the embedding. Embeddings of this form were considered by Abraham et al. [2], who proved that the dimension must depend on $\log n$. Thus any distance labeling that uses a slack embedding into ℓ_p must use space that depends on $\log n$, whereas our labeling is independent of n.

Theorem 12. *Let (V, d) be a metric space with n points. Suppose that there exists a distance labeling where each label has size $O(f(n))$ and for any two points u, v it is possible to compute, in $O(q(n))$ time, an approximation to the distance between u and v with a stretch of at most $\alpha(n)$. Then there exists a distance labeling with ϵ-uniform slack such that every label has size $O(f(\frac{1}{\epsilon}))$, and computing distances up to a stretch of $5 + 6\alpha(\frac{1}{\epsilon})$ can be done in $O(q(\frac{1}{\epsilon}))$ time.*

We get the following corollary by simply applying Theorem 12 to the distance labeling of Thorup and Zwick [27, Theorem 3.4]. Note that the size of the labels is independent of n.

Corollary 6 (Uniform slack distance labeling). *Let (V, d) be a metric space on n points. Let $0 < \epsilon < 1$, and let k be an integer with $1 \leq k \leq \log \frac{1}{\epsilon}$. Then it is possible to assign each point a label that uses $O((\frac{1}{\epsilon})^{1/k} \log^{1-1/k} \frac{1}{\epsilon})$ space such that, given the labels of vertices u, v where v is ϵ-far from u, the distance $d(u, v)$ can be computed up to a stretch of $12k - 1$ in $O(k)$ time.*

We can also get gracefully degrading labelings. These labelings will be larger than the embeddings into ℓ_p given by Abraham et. al. [1, Theorem 10], but will have faster time complexity. We get the following result by combining a standard transformation theorem (which we omit) with the labelings of Thorup and Zwick [27] and Theorem 10.

Corollary 7 (Gracefully degrading distance labeling). *For any integer k with $1 \leq k \leq O(\log n)$, there is a distance labeling of any n point metric such that each label has size at most $O(n^{1/k} \log^{1-1/k} n)$, and given two labels it is possible to compute the distance between the two points up to a worst case stretch of $2k - 1$ in $O(k)$ time. Furthermore, the average distortion and the distortion of average are $O(1)$.*

References

1. I. Abraham, Y. Bartal, and O. Neiman. Advances in metric embedding theory. In *Proceedings of the 38th Annual ACM Symposium on Theory of Computing*, 2006.
2. Ittai Abraham, Yair Bartal, T-H. Hubert Chan, Kedar Dhamdhere Dhamdhere, Anupam Gupta, Jon Kleinberg, Ofer Neiman, and Aleksandrs Slivkins. Metric embeddings with relaxed guarantees. In *FOCS '05: Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science*, pages 83–100, Washington, DC, USA, 2005. IEEE Computer Society.
3. Ingo Althöfer, Gautam Das, David Dobkin, Deborah Joseph, and José Soares. On sparse spanners of weighted graphs. *Discrete Comput. Geom.*, 9(1), 1993.
4. Sunil Arya, Gautam Das, David M. Mount, Jeffrey S. Salowe, and Michiel H. M. Smid. Euclidean spanners: short, thin, and lanky. *STOC*, 1995.
5. Yair Bartal. Probabilistic approximations of metric spaces and its algorithmic applications. In *Proceedings of the37thAnnual IEEE Symposium on Foundations of Computer Science*, 1996.
6. Surender Baswana, Telikepalli Kavitha, Kurt Mehlhorn, and Seth Pettie. New constructions of (α, β)-spanners and purely additive spanners. In *SODA*, pages 672–681, 2005.
7. Surender Baswana and Sandeep Sen. Approximate distance oracles for unweighted graphs in $\tilde{o}(n^2)$ time. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 271–280, 2004.
8. Béla Bollobás, Don Coppersmith, and Michael Elkin. Sparse distance preservers and additive spanners (extended abstract). In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (Baltimore, MD, 2003)*, pages 414–423, New York, 2003. ACM.

9. Jean Bourgain. On Lipschitz embeddings of finite metric spaces in Hilbert space. *Israel Journal of Mathematics*, 52(1-2), 1985.
10. Hubert T-H. Chan, Anupam Gupta, Bruce M. Maggs, and Shuheng Zhou. On hierarchical routing in DOubling metrics. In *Proceedings of the16thAnnual ACM-SIAM Symposium on Discrete Algorithms*, 2005.
11. Barun Chandra, Gautam Das, Giri Narasimhan, and José Soares. New sparseness results on graph spanners. *Internat. J. Comput. Geom. Appl.*, 5(1-2), 1995. Eighth Annual ACM Symposium on Computational Geometry (Berlin, 1992).
12. S. Chaudhuri and C. D. Zaroliagis. Shortest paths in digraphs of small treewidth. I. Sequential algorithms. *Algorithmica*, 27(3-4):212–226, 2000. Treewidth.
13. Don Coppersmith and Michael Elkin. Sparse source-wise and pair-wise distance preservers. In *Proceedings of the16thAnnual ACM-SIAM Symposium on Discrete Algorithms*, 2005.
14. Michael Elkin and David Peleg. $(1+\epsilon, \beta)$-spanner constructions for general graphs. *SIAM J. Comput.*, 33(3):608–631 (electronic), 2004.
15. Jittat Fakcharoenphol, Satish Rao, and Kunal Talwar. A tight bound on approximating arbitrary metrics by tree metrics. In *Proceedings of theAnnual ACM Symposium on Theory of Computing*, 2003.
16. Cyril Gavoille, David Peleg, Stephane Perennes, and Ran Raz. Distance labeling in graphs. In *Proceedings of the12thAnnual ACM-SIAM Symposium on Discrete Algorithms*, 2001.
17. Joachim Gudmundsson, Christos Levcopoulos, Giri Narasimhan, and Michiel H. M. Smid. Approximate distance oracles for geometric graphs. In *SODA*, pages 828–837, 2002.
18. Anupam Gupta, Amit Kumar, and Rajeev Rastogi. Traveling with a Pez dispenser (or, routing issues in MPLS). *SIAM J. Comput.*, 34(2):453–474 (electronic), 2004/05.
19. Sariel Har-Peled and Manor Mendel. Fast construction of nets in low dimensional metrics, and their applications. *Symposium on Computational Geometry*, pages 150–158, 2005.
20. Piotr Indyk. Algorithmic aspects of geometric embeddings. In *Proceedings of the42ndAnnual IEEE Symposium on Foundations of Computer Science*, 2001.
21. Jon M. Kleinberg, Aleksandrs Slivkins, and Tom Wexler. Triangulation and embedding using small sets of beacons. In *Proceedings of the45thAnnual IEEE Symposium on Foundations of Computer Science*, 2004.
22. David Peleg. Proximity-preserving labeling schemes and thei applications. In *25th Workshop on Graph-Theoretic Concepts in Computer Science*, Lecture Notes in Computer Science 1665, 1999.
23. David Peleg. *Distributed computing.* Society for Industrial and Applied Mathematics (SIAM), 2000. A locality-sensitive approach.
24. David Peleg and Alejandro A. Schäffer. Graph spanners. *J. Graph Theory*, 13(1), 1989.
25. Kunal Talwar. Bypassing the embedding: Algorithms for low-dimensional metrics. In *Proceedings of the36thAnnual ACM Symposium on Theory of Computing*, 2004.
26. Mikkel Thorup. Compact oracles for reachability and approximate distances in planar digraphs. In *Proceedings of the42ndAnnual IEEE Symposium on Foundations of Computer Science*, 2001.
27. Mikkel Thorup and Uri Zwick. Approximate distance oracles. In *Proceedings of the33ndAnnual ACM Symposium on Theory of Computing*, 2001.
28. Mikkel Thorup and Uri Zwick. Spanners and emulators with sublinear distance errors. In *SODA*, pages 802–809, 2006.

Compressed Indexes for Approximate String Matching

Ho-Leung Chan[1], Tak-Wah Lam[1,⋆], Wing-Kin Sung[2], Siu-Lung Tam[1], and Swee-Seong Wong[2]

[1] Department of Computer Science, University of Hong Kong
{hlchan, twlam, sltam}@cs.hku.hk
[2] Department of Computer Science, National University of Singapore
{ksung, wongss}@comp.nus.edu.sg

Abstract. We revisit the problem of indexing a string $S[1..n]$ to support searching all substrings in S that match a given pattern $P[1..m]$ with at most k errors. Previous solutions either require an index of size exponential in k or need $\Omega(m^k)$ time for searching. Motivated by the indexing of DNA sequences, we investigate space efficient indexes that occupy only $O(n)$ space. For $k = 1$, we give an index to support matching in $O(m+occ+\log n \log\log n)$ time. The previously best solution achieving this time complexity requires an index of size $O(n \log n)$. This new index can be used to improve existing indexes for $k \geq 2$ errors. Among others, it can support matching with $k = 2$ errors in $O(m \log n \log\log n + occ)$ time.

1 Introduction

Given a string $S[1..n]$ over a constant-size alphabet Σ and an integer $k \geq 0$, we want to build an index for S, such that for any subsequent query pattern $P[1..m]$, we can report efficiently all substrings in S that match P with at most k errors. The primary concern is how to achieve efficient pattern matching given limited space for indexing. We consider two kinds of errors: In the Hamming distance case, an error is a character substitution; in the edit distance case, an error can be a character substitution, insertion or deletion.

For exact string matching (i.e., $k = 0$), simple and efficient solutions have been known since the 1970s. Suffix trees [20,14] use $O(n)$ space[1] and achieve the optimal matching time, i.e. $O(m + occ)$, where occ is the number occurrences of P in S. Suffix arrays [13], also using $O(n)$ space but with a smaller constant, give an $O(m+occ+\log n)$ matching time. Recently, two compressed solutions, namely, CSA [8] and FM-index [7], have been proposed; they require only $O(n)$-bit space and support matching in $O(m + occ \log^{\epsilon} n)$ time, for any constant $\epsilon > 0$.

Approximate matching is a more challenging problem even if only one error is allowed. The simplest solution is to search the suffix tree of S for every 1-error

⋆ This research was supported by Hong Kong RGC Grant HKU 7140/06E.

[1] Unless otherwise stated, the space complexity is measured in terms of the number of words, where a word can store $O(\log n)$ bits.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 208–219, 2006.

Table 1. A summary of results. Results given in this paper are marked with †.

Space	$k = 1$		$k = 2$	
$O(n \log^2 n)$ words	$O(m \log n \log\log n + occ)$	[1]	$O(m + \log^2 n \log\log n + occ)$	[5]
$O(n \log n)$ words	$O(m \log\log n + occ)$	[2]	$O(m + \log^2 n \log\log n + occ)$	†
	$O(m + \log n \log\log n + occ)$	[5]		
$O(n)$ words	$O(\min\{n, m^2\} + occ)$	[4]	$O(\min\{n, m^3\} + occ)$	[4]
	$O(m \log n + occ)$	[10]	$O(m^2 \log n + occ)$	[10]
	$O(m \log\log n + occ)$	[11]	$O(m^2 \log\log n + occ)$	[11]
	$O(m + \log^3 n \log\log n + occ)$	[6]	$O(m + \log^6 n \log\log n + occ)$	[6]
	$O(m + \log n \log\log n + occ)$	†	$O(m \log n \log\log n + occ)$	†

modification of the query pattern, this requires $O(m^2 + occ)$ time[2] [4]. The first non-trivial improvement was due to Amir et al. [1], who showed that the matching time can be improved to $O(m \log n \log\log n + occ)$ using an index occupying $O(n \log^2 n)$ space. Later Buchshaum et al. [2] further improved the matching time to $O(m \log\log n + occ)$, as well as reducing the index space to $O(n \log n)$. Huynh et al. [10] and Lam et al. [11] further compressed the index to $O(n)$ space, while achieving the time complexity of the indexes reported in [1] and [2], respectively. It has been an open problem whether a time complexity linear in m and occ can be achieved. Recently, Cole et al. [5] resolved in the affirmative with an $O(n \log n)$-space index that supports one-error matching in $O(m + \log n \log\log n + occ)$ time. And more recently, Chan et al. [6] found that Cole et al.'s index admits a time-space tradeoff, i.e., the space can be reduced to $O(n)$ space, yet the time complexity increases to $O(m + \log^3 n \log\log n + occ)$. In this paper, we give new techniques for compressing Cole et al.'s index to $O(n)$ space, while retaining the same time complexity.

To cater for $k = O(1)$ errors, one can perform a brute-force search on a one-error index (i.e., repeatedly modify the pattern at different $k - 1$ positions and search for one-error matches); the matching becomes very slow, involving a factor of m^k in the time complexity. A breakthrough result was given by Cole et al. [5], who devised a recursive solution to build an index that occupies $O(n \log^k n)$ space and supports k-error matching in $O(m + \log^k n \log\log n + occ)$ time for Hamming distance. The term occ is replaced with $occ \cdot 3^k$ for edit distance. Our new 1-error index is essentially a compressed version of the Cole et al.'s 1-error index and can replace it as the base case in their recursive solution. This gives an $O(n \log^{k-1} n)$-space index for k-error matching with the same time complexity.

For indexing long sequences like DNA (which contains a few million to a few billion characters), it is not desirable to have an index whose space complexity grows exponentially as k increases. Like the case of 1-error, the k-error index of Cole et al.'s also admits a time-space tradeoff; in particular, Chan et al. [6] showed that the tree cross product technique by Buchshaum et al. [2] can be

[2] Unless otherwise stated, all matching time mentioned applies to both Hamming and edit distance.

used to trade time for space in the k-error index by Cole et al., the space can be reduced to $O(n)$ while the time for k-error matching increases to $O(m + \log^{k(k+1)} n \log\log n + occ)$. Note that this result is of theoretical interest only as the time complexity is far from practical. For $k = 2$, the time complexity already involves a term $\log^6 n \log\log n$, which is likely to be much bigger than m in most applications. In this paper, we devise a more practical solution for 2-error matching. Specifically, we show that our new $O(n)$-space index for 1-error matching can readily support 2-error matching in $O(m \log n \log\log n + occ)$ time. Furthermore, this index can also handle $k \geq 3$ errors using a brute force manner, and the matching time is $O(m^{k-1} \log n \log\log n + occ)$.

In this paper, we assume the alphabet size $|\Sigma|$ is a constant and hence does not affect the asymptotic analysis. If $|\Sigma|$ is huge or unbounded, we remark that our data structures takes $O(n)$ space (i.e., does not involve $|\Sigma|$); the 1-error matching time is $O(m + |\Sigma| \log n \log\log n + occ)$, the 2-error matching time is $O(|\Sigma|^2 m \log n \log\log n + occ)$, and the k-error matching time, $k > 2$, is $O(|\Sigma|^k m^{k-1} \log n \log\log n + occ)$.

On the technical side, our result is based on a new technique to replace the tree-like data structure of Cole et al. [5] with simple arrays of integers, which are basically some kind of lexicographical information about a suffix tree. We show how approximate string matching can be done by simple range queries over these arrays, instead of the more complicated tree traversals as in [5]. Furthermore, we show how to compress these arrays by storing the lexicographical information imprecisely. This simple approximation can save space and can be verified efficiently. Using the known results on concise representation of increasing sequences and range searching, we reduce the space requirement of Cole et al. by a factor of $O(\log n)$, without increasing the matching time.

We extend our data structure for 1-error matching to support a lazy preprocessing of the input pattern P, which takes $O(m)$ time. Then, for any P' formed by modifying P at one position, we can find the 1-error matches of P' in S in $O(\log n \log\log n + occ')$ time, where occ' is the number of 1-error matches for P'. There are $O(m)$ possible P', so all the 2-error matches of P can be found in $O(m \log n \log\log n + occ)$ time.

Due to the page limit, we omitted the details about our results on k-error matching, which will be given in the full paper. We remark that our paper concerns only worst-case performance. The literature also contains several interesting results on average-case performance, see, e.g., [17, 12, 3].

2 Preliminaries

Let $S[1..n]$ be a string over a constant-size alphabet Σ. The *suffix tree* T of S is a compact trie comprising all suffixes of S. Throughout this paper, we assume that the suffixes are ordered from left to right in increasing lexicographical order. The *suffix array* $SA[1..n]$ is an array of integers such that $SA[i] = j$ if $S[j..n]$ is the lexicographically i-th suffix of S. Note that the *inverse suffix array* $SA^{-1}[1..n]$ satisfies that $SA^{-1}[j]$ gives the lexicographical order of the suffix $S[j..n]$. We

always store T, $SA[1..n]$ and $SA^{-1}[1..n]$, which take $O(n)$ words, or equivalently, $O(n \log n)$ bits.

2.1 Centroid Path Decomposition

For a suffix tree T, the centroid path decomposition [5] of T is defined as follows. For every internal node u, let v be the child of u with the most number of leaves (ties broken arbitrarily). The edge uv is called a *core edge*. Edges other than core edges are called *side edges*. A *centroid path* $\mathcal{C}$ is a maximal path connecting consecutive core edges. The *root* of $\mathcal{C}$, denoted $r(\mathcal{C})$, is the top-most node on $\mathcal{C}$. The *level* of $\mathcal{C}$ is the number of side edges on the path from the root of T to $r(\mathcal{C})$. We denote $\Delta(T)$ the set of all centroid paths in T.

Denote T_u the subtree of T rooted at a node u and $|T_u|$ be the number of leaves in T_u. Let $T_{\mathcal{C}}$ be $T_{r(\mathcal{C})}$. A leaf (i.e., a suffix) x of T *belongs to* a centroid path $\mathcal{C}$ if x is in $T_{\mathcal{C}}$. A node u *hangs from* $\mathcal{C}$ if its parent edge is a side edge connecting to a node on $\mathcal{C}$, and T_u is called a *side tree* of $\mathcal{C}$. For any node u hanging from $\mathcal{C}$, we note that $|T_u| \leq |T_{\mathcal{C}}|/2$. We highlight some properties of the decomposition.

Fact 1. **(i)** *For any leaf x in T, the path from root of T to x has at most* $\log n$ *side edges, and x belongs to at most* $\log n$ *centroid paths.* **(ii)** $\Sigma_{\mathcal{C}\in\Delta(T)}|T_{\mathcal{C}}| \leq n \log n$. **(iii)** *For any two centroid paths $\mathcal{C}_1$ and $\mathcal{C}_2$ of the same level, $T_{\mathcal{C}_1}$ and $T_{\mathcal{C}_2}$ are disjoint, i.e., they do not have common leaves.*

2.2 The Side-Tree Rank of a Leaf

Consider a centroid path C. The leaves in $T_{\mathcal{C}}$ are partitioned among the side trees, and our compressed index needs the association between the leaves and the side trees. To save space, we rank the side trees hanging from $\mathcal{C}$ in descending order of their size (i.e., the number of leaves), and we store, for each leaf x, the rank of the side tree containing x, which is denoted *st-rank*$_{\mathcal{C}}(x)$. To store such side-tree ranks for all centroid paths, we need $\sum_{\mathcal{C}\in\Delta(T)}\sum_{x\in T_{\mathcal{C}}}\lceil \log \textit{st-rank}_{\mathcal{C}}(x)\rceil$ bits, which is naively at most $n \log^2 n$ bits (because $\Sigma_{\mathcal{C}\in\Delta(T)}|T_{\mathcal{C}}| \leq n \log n$ and *st-rank*$_{\mathcal{C}}(x) \leq n$). Our compressed index actually takes advantage of a better upper bound.

Lemma 1. **(i)** *Let x be a leaf in T, belonging to centroid paths $\mathcal{C}_1, \mathcal{C}_2, \cdots, \mathcal{C}_\alpha$. Then,* $\sum_{1\leq i\leq\alpha} \log \textit{st-rank}_{\mathcal{C}_i}(x) \leq \log n$. **(ii)** $\sum_{\mathcal{C}\in\Delta(T)}\sum_{x\in T_{\mathcal{C}}}\lceil \log \textit{st-rank}_{\mathcal{C}}(x)\rceil \leq 2n \log n$.

Proof. Let x be any leaf in T. **(i)** We assume that the $\alpha \geq 1$ centroid paths to which x belongs are labeled in such a way that $r(\mathcal{C}_{i+1})$ hangs from $\mathcal{C}_i$, for $i = 1, \ldots, \alpha - 1$. Let $r_i = \textit{st-rank}_{\mathcal{C}_i}(x)$. We note that $|T_{\mathcal{C}_i}| \geq |T_{\mathcal{C}_{i+1}}| \times r_i$, because the r_i-th largest side tree has at most $\frac{1}{r_i}$ of all leaves belonging to $\mathcal{C}_i$. Thus, we have $\sum_{i=1}^{\alpha} \log r_i \leq \sum_{i=1}^{\alpha-1} \log(\frac{|T_{\mathcal{C}_i}|}{|T_{\mathcal{C}_{i+1}}|}) + \log r_\alpha \leq \log \frac{|T_{\mathcal{C}_1}|}{|T_{\mathcal{C}_\alpha}|} + \log |T_{\mathcal{C}_\alpha}| = \log |T_{\mathcal{C}_1}| \leq \log n$. **(ii)** It follows directly from (i). □

2.3 The y-Fast Trie

Let $A[1..\ell]$ be a sorted array of integers. Given any integer j, the predecessor query reports the smallest i such that $j < A[i]$. Willard [21] gave the y-fast trie data structure with the following performance.

Lemma 2 ([21]). *Let $A[1..\ell]$ be a sort array of integers in $[1, n]$, we can build a y-fast trie for A using $O(\ell \log n)$ bits to support the predecessor query in $O(\log \log n)$ time.*

2.4 The LCP Data Structure

Let T be the suffix tree for $S[1..n]$. Let T' be a trie for only a subset of suffixes in T. A *location* in T' is a node in T' or a point on an edge some characters below a node. Given a pattern $P[1..m]$, an integer $i \leq m$ and a location u in T', the query $\mathrm{LCP}(P, i, u)$ asks for the location at which the suffix $P[i..m]$ diverges from T', when $P[i..m]$ is aligned to T' starting from u. We are allowed to preprocess P in $O(m)$ time, and the concern is to efficiently answer subsequent LCP queries for different suffix $P[i..m]$ and location u.

Let ℓ be the number of leaves in T'. Cole et al. [5] proposed an $O(\ell \log^2 n)$-bit LCP data structure to answer each subsequent LCP query in $O(\log \log n)$ time. In this paper, we use a simple observation to reduce the space requirement to $O(\ell \log n)$ bits.

We first outline the LCP data structure in [5]. Essentially, [5] performs a centroid path decomposition on T'. For any $\mathcal{C} \in \Delta(T')$, let $T'_{\mathcal{C}}$ be the subtree of T' rooted at $r(\mathcal{C})$ and let $\ell_{\mathcal{C}}$ be the number of leaves in $T'_{\mathcal{C}}$. The path from $r(\mathcal{C})$ to a leaf in T' is a suffix of S. In [5], an array $A_{\mathcal{C}}[1..\ell_{\mathcal{C}}]$ is used to store the lexicographical order of each such suffix, among all suffixes of S. Note that $A_{\mathcal{C}}[1..\ell_{\mathcal{C}}]$ is strictly increasing. A y-fast trie [21] of size $O(\ell_{\mathcal{C}} \log n)$ bits is built to answer in $O(\log \log n)$ time the predecessor query. These A arrays and y-fast tries over all centroid paths in T' take totally $O(\ell \log^2 n)$-bit space. The remaining part of the LCP data structure takes only $O(\ell \log n)$-bit space. We use the following observation to reduce the space requirement.

Lemma 3. *For any centroid path $\mathcal{C}$ in T', let $h_{\mathcal{C}}$ be the total length of edge label from root of T' to $r(\mathcal{C})$.* **(i)** *For $i = 1, \ldots, \ell_{\mathcal{C}}$, $A_{\mathcal{C}}[i]$ can be computed in $O(1)$ time using $h_{\mathcal{C}}$ and the inverse suffix array of S.* **(ii)** *The predecessor query can be supported in $O(\log \log n)$ time using an $O(\ell_{\mathcal{C}})$-bit data structure.*

Proof. **(i)** Consider the i-th leaf in $T'_{\mathcal{C}}$ and let $S[j..n]$ be its leaf label, i.e., $S[j..n]$ is the suffix corresponding to the path from root of T' to this leaf. By definition, $A_{\mathcal{C}}[i]$ is the lexicographical order of $S[j + h_{\mathcal{C}}..n]$, so $A_{\mathcal{C}}[i] = SA^{-1}[j + h_{\mathcal{C}}]$.

(ii) Instead of building a y-fast trie on the complete $A_{\mathcal{C}}$ array, we only build a y-fast trie for $A_{\mathcal{C}}[\log n], A_{\mathcal{C}}[2 \log n], \ldots$ using $O(\ell_{\mathcal{C}})$ bits space. The predecessor query can be done by first querying y-fast trie, then performing a binary search in $A_{\mathcal{C}}$ within an interval of length $\log n$. It takes $O(\log \log n)$ time. □

Thus, for each centroid path $\mathcal{C}$, we store an integer $h_{\mathcal{C}}$ and an $O(\ell_{\mathcal{C}})$-bit predecessor data structure. It takes totally $O(\ell \log n)$ bits over all centroid paths in T'. Together with the remaining part of the LCP data structure of [5], we have the following lemma.

Lemma 4. *Let T' be a compact trie comprising ℓ suffixes of $S[1..n]$. We can build an $O(\ell \log n)$-bit LCP data structure for T'. Given any pattern $P[1..m]$, we can preprocess P in $O(m)$ time. Each subsequent LCP query can be answered in $O(\log \log n)$ time.*

3 An $O(n \log n)$-Bit Index for 1-Error Matching

This section explains how to compress the data structure of Cole et al. [5] in order to obtain an $O(n \log n)$-bit index for $S[1..n]$, such that for any pattern $P[1..m]$, it finds the 1-error matches of P in $O(m + occ + \log n \log \log n)$ time. We consider Hamming distance first. Extension to edit distance is given at the end of the section.

Let us first consider the suffix tree T of S. We perform a centroid path decomposition on T and let $\Delta(T)$ be the set of all centroid paths. For any centroid path $\mathcal{C}$, we define a set of *modified suffixes* as follows. Let s be a suffix in T passing through the root of $\mathcal{C}$, and diverging from $\mathcal{C}$ at a node u on $\mathcal{C}$. We create a modified suffix s' by modifying s at the first character after u, replacing it with the first character on the core edge out of u. We say that s *generates* s' *according to* $\mathcal{C}$. We assume the suffix corresponding to $\mathcal{C}$ itself is also a modified suffix generated according to $\mathcal{C}$.

To find the 1-error matches of P in S, the core of our algorithm is solving the following prefix matching problem.

Definition 1 (Prefix matching query for modified suffixes). Let T be the suffix tree of S and P be a pattern. For any centroid path $\mathcal{C}$ of T, let $\phi_{\mathcal{C}}$ be the set of modified suffixes generated according to $\mathcal{C}$ with P as a prefix. The *prefix matching query*, denoted *prefix*$(\mathcal{C})$, reports the set $\Phi_{\mathcal{C}}$ of suffixes in T that generate the modified suffixes in $\phi_{\mathcal{C}}$.

Lemma 5. *Let T be the suffix tree of $S[1..n]$. We can build an $O(n \log n)$-bit index for T. For any pattern $P[1..m]$, we can preprocess P in $O(m)$ time; then prefix$(\mathcal{C})$, for any centroid path $\mathcal{C}$ in T, can be answered in $O(\log \log n + |\Phi_{\mathcal{C}}| + e_{\mathcal{C}})$ time, where $e_{\mathcal{C}}$ is non-negative and the sum of $e_{\mathcal{C}}$ over all centroid paths in T is at most $2 \log n$.*

The following subsections are devoted to the proof of Lemma 5. Then by using the framework of Cole et al. [5], we can exploit our indexes for the prefix matching queries and LCP queries to obtain an $O(n \log n)$-bit index for the 1-error matching problem, and Theorem 1 follows. We leave the details to the full paper.

Theorem 1. *We can build an $O(n \log n)$-bit index for $S[1..n]$ that finds the 1-error matches of any $P[1..m]$ in $O(m + occ + \log n \log \log n)$ time, where occ is the number of matches found.*

3.1 The Prefix Matching Query for Modified Suffixes

Cole et al. [5] used the error-tree data structure to support the prefix matching query in $O(m)$ preprocessing time and $O(\log\log n + |\Phi_{\mathcal{C}}|)$ query time. Their solution takes $O(n\log^2 n)$-bits and requires sophisticated tree operations. In this paper, we uses interesting techniques to replace their tree-like data structure with simple arrays of integers.

A simple $O(n\log^2 n)$-bit solution. Let T be the suffix tree of S. Let U be the set of all the $O(n\log n)$ modified suffixes generated according to all the centroid paths. For each centroid path $\mathcal{C}$, we simply store two arrays of integers. Let $s'_1, s'_2, \ldots, s'_\ell$ be the modified suffixes generated according to $\mathcal{C}$, in increasing lexicographical order. We store (1) *lex-order*$_{\mathcal{C}}$, where *lex-order*$_{\mathcal{C}}[i]$ is the lexicographical order of s'_i among all modified suffixes in U. (2) *label*$_{\mathcal{C}}$, where *label*$_{\mathcal{C}}[i] = j$ if s'_i is generated by the suffix $S[j..n]$.

In addition, we store a compact trie M for U. Given any pattern P, we preprocess P by aligning P with M starting from the root. It determines the range $[d, e]$ such that all modified suffixes with lexicographical order in $[d, e]$ (w.r.t. U) have P as a prefix. Given any centroid path $\mathcal{C}$ of T, the prefix matching query is done by a range search query on *lex-order*$_{\mathcal{C}}$. For each i such that $d \le$ *lex-order*$_{\mathcal{C}}[i] \le e$, we report *label*$_{\mathcal{C}}[i]$. The range search on *lex-order*$_{\mathcal{C}}$ takes $O(\log\log n)$ time by storing a y-fast trie [21] on *lex-order*$_{\mathcal{C}}$. Thus, finding the $\Phi_{\mathcal{C}}$ takes $O(\log\log n + |\Phi_{\mathcal{C}}|)$ time. The total space requirement is $O(n\log^2 n)$ bits.

An $O(n\log n)$-bit solution. We exploit sophisticated techniques to reduce the space requirement of the above solution to $O(n\log n)$ bits.

1. *Sampling.* Instead of M, we store a compact trie containing only one in every $\log n$ leaves of M. With this approximation, answering the prefix matching query on a centroid path $\mathcal{C}$ requires extra verification on the suffixes before reporting it as $\Phi_{\mathcal{C}}$. Let $e_{\mathcal{C}}$ be the number of suffixes that require verification. We show that the sum of $e_{\mathcal{C}}$ is at most $2\log n$ over all centroid paths.
2. *Constant time verification.* Given the pattern P, a centroid path $\mathcal{C}$ and a suffix $s = S[j..n]$, we need to verify whether s generates a modified suffix s' according to $\mathcal{C}$ with P as a prefix. We show how to perform the verification in $O(1)$ time using the suffix tree, suffix array and the LCP data structure.
3. *Concise representation.* The *lex-order*$_{\mathcal{C}}$ and *label*$_{\mathcal{C}}$ arrays take totally $O(n\log^2 n)$ bits if stored directly. We replace their entries with integers of smaller values, by exploiting the properties of the centroid path decomposition. Then, we use variable size encoding to represent the arrays in $O(n\log n)$ bits.

Precisely, our $O(n\log n)$-bit solution stores a compact trie N comprising $O(n)$ modified suffixes in U, namely, the lexicographically $(\log n)$-th, $(2\log n)$-th, $(3\log n)$-th, ... modified suffixes. For a centroid path $\mathcal{C}$, let $s'_1, s'_2, \ldots, s'_\ell$ be the modified suffixes generated for $\mathcal{C}$. We store two length-ℓ arrays for $\mathcal{C}$.

- *lex-order*$_{\mathcal{C}}[1..\ell]$: *lex-order*$_{\mathcal{C}}[i]$ is the lexicographical order of s'_i among all the $O(n)$ modified suffixes in N.
- *label*$_{\mathcal{C}}[1..\ell]$: Define *label*$_{\mathcal{C}}[i] = j$ if s'_i is generated by $S[j..n]$.

A naive way to store the *rank* and *label* arrays still takes $O(n \log^2 n)$ bits. In Section 3.3, we give non-trivial techniques to compress them into $O(n \log n)$ bits. We first proceed to show how to use these two arrays to find $\Phi_{\mathcal{C}}$ efficiently.

3.2 Answering a Prefix Matching Query

Given a pattern P, we show how to preprocess P in $O(m)$ time such that for any centroid path $\mathcal{C}$, the prefix matching query *prefix*($\mathcal{C}$) can be answered in $O(\log \log n + |\Phi_{\mathcal{C}}| + e_{\mathcal{C}})$ time, and the sum of $e_{\mathcal{C}}$ over all centroid paths $\mathcal{C}$ is at most $2 \log n$.

Error-bounded candidate generation. We align P with N starting from the root in $O(m)$ time to find the range $[d, e]$ corresponding to leaves in N with P as a prefix. Then, for any centroid path $\mathcal{C}$, we can find $\Phi_{\mathcal{C}}$ as follows.

1. Find the maximal range $[p..q]$ such that $d - 1 \leq$ *lex-order*$_{\mathcal{C}}[p]$ $\leq$ *lex-order*$_{\mathcal{C}}[q] \leq e + 1$ by a *range_search* query on the *lex-order*$_{\mathcal{C}}$ array.
2. For each i in $[p..q]$, let $j =$ *label*$_{\mathcal{C}}[i]$. If *lex-order*$_{\mathcal{C}}[i]$ is not $d - 1$ or $e + 1$, report $S[j..n]$ in $\Phi_{\mathcal{C}}$; otherwise, call $S[j..n]$ a *candidate* and verify whether $S[j..n]$ is in $\Phi_{\mathcal{C}}$.

We want the *lex-order*$_{\mathcal{C}}$ array to support the operation *range_search*(d, e): given integers d and e, $d \leq e$, return p, q such that *lex-order*$_{\mathcal{C}}[p..q]$ is the largest interval satisfying $d - 1 \leq$ *lex-order*$_{\mathcal{C}}[p] \leq$ *lex-order*$_{\mathcal{C}}[q] \leq e + 1$. We can build a y-fast trie [21] on one per $\log n$ entries in *lex-order*$_{\mathcal{C}}$. Then *range_search* can be done in $O(\log \log n)$ time by a query to the y-fast trie and then a binary search in an interval of length $\log n$. It uses $O(n \log n)$ bits over all centroid paths.

Lemma 6. *For any centroid path $\mathcal{C}$, let $e_{\mathcal{C}}$ be the number of candidates generated for verification. The sum of $e_{\mathcal{C}}$ over all centroid paths is at most $2 \log n$.*

Proof. For any integer i, at most $\log n$ entries over all *lex-order* arrays equal i, and we verify a suffix only if its *lex-order* value is $d - 1$ or $e + 1$. □

Constant time verification. We need to verify whether a candidate is in $\Phi_{\mathcal{C}}$.

Lemma 7. *We can preprocess P in $O(m)$ time. Then, for any centroid path $\mathcal{C}$ and candidate $S[j..n]$, we can verify in $O(1)$ time whether $S[j..n]$ is in $\Phi_{\mathcal{C}}$, i.e., $S[j..n]$ generates a modified suffix according to $\mathcal{C}$ with P as a prefix.*

Proof. We preprocess P with the suffix tree T, which takes $O(m)$ time: For each suffix $P[r..m]$, we store the range $[d_r, e_r]$ such that all leaves with lexicographical order (w.r.t. T) in $[d_r, e_r]$ have $P[r..m]$ as a prefix.

To verify a suffix $S[j..n]$, let v be the node in T that $S[j..n]$ diverges from the path of P. v can be found in constant time using an $O(n \log n)$-bit LCA data structure [9] for T. Let $P[1..r]$ (or equivalently, $S[j..j + r - 1]$) be the path label from the root to v. (For each node v in T, we store the path length from the root to v so that $P[1..r]$ is known in $O(1)$ time.) $S[j..n]$ is in $\Phi_{\mathcal{C}}$ if (1) v is on $\mathcal{C}$, (2) the first character on the core edge out of v is $P[r+1]$, and (3) $S[j+r+1..n]$

has a prefix matching $P[r+2..m]$. The last condition can be checked in constant time by comparing $SA^{-1}[j+r+1]$ with the range $[d_{r+2}, e_{r+2}]$ obtained during the preprocessing. □

In conclusion, Lemmas 6 and 7 show that we can build $O(n \log n)$-bit data structures on top of *lex-order*$_{\mathcal{C}}$ and *label*$_{\mathcal{C}}$. Then, we can preprocess P in $O(m)$ time; for any centroid path $\mathcal{C}$, we can answer the prefix matching query in $O(\log \log n + |\Phi_{\mathcal{C}}| + e_{\mathcal{C}})$ time, where $e_{\mathcal{C}}$ is the number of verification performed and the sum of $e_{\mathcal{C}}$ over all centroid path is at most $2 \log n$.

3.3 Compressed Representation of the Lexicographical Information

We show how to store the *lex-order* and *label* arrays in $O(n \log n)$-bit space.

Compressing the *lex-order* **arrays.** For any centroid path $\mathcal{C}$, entries in *lex-order*$_{\mathcal{C}}$ are monotonic increasing, so efficient compression is possible by Lemma 8. Proof of Lemma 8 will be given in the full paper.

Lemma 8. *Let $c_1 \leq c_2 \leq \ldots \leq c_\ell$ be a sequence of positive integers. We can store the sequence in $O(\log c_1 + \ell \cdot \max\{\log(\frac{c_\ell - c_1}{\ell}), 1\})$ bits and support $O(1)$ retrieval time for each c_i.*

Lemma 9. *We can store the lex-order arrays of all centroid paths in $O(n \log n)$-bit space and support $O(1)$ retrieval time to each entry.*

Proof. For any centroid path $\mathcal{C}$, let $\ell_{\mathcal{C}}$ be the number of modified suffixes generated according to $\mathcal{C}$. Let $h_{\mathcal{C}} =$ *lex-order*$_{\mathcal{C}}[\ell_{\mathcal{C}}] -$ *lex-order*$_{\mathcal{C}}[1]$. Consider all $\mathcal{C} \in \Delta(T)$. By Lemma 8, the total space required for the *lex-order*$_{\mathcal{C}}$ arrays is $\sum O(\log$ *lex-order*$_{\mathcal{C}}[1] + \ell_{\mathcal{C}} \cdot \max\{\log(\frac{h_{\mathcal{C}}}{\ell_{\mathcal{C}}}), 1\}) \leq \sum O(\log$ *lex-order*$_{\mathcal{C}}[1] + \ell_{\mathcal{C}} \cdot \log(2 + \frac{h_{\mathcal{C}}}{\ell_{\mathcal{C}}})) = O(n \log n) + O(\log \prod (2 + \frac{h_{\mathcal{C}}}{\ell_{\mathcal{C}}})^{\ell_{\mathcal{C}}})$ bits. By the AM-GM inequality on $\sum \ell_{\mathcal{C}}$ numbers, we have $\prod \left(2 + \frac{h_{\mathcal{C}}}{\ell_{\mathcal{C}}}\right)^{\ell_{\mathcal{C}}} \leq \left(\frac{1}{\sum \ell_{\mathcal{C}}} \sum \ell_{\mathcal{C}} (2 + \frac{h_{\mathcal{C}}}{\ell_{\mathcal{C}}})\right)^{\sum \ell_{\mathcal{C}}} = \left(\frac{2 \sum \ell_{\mathcal{C}} + \sum h_{\mathcal{C}}}{\sum \ell_{\mathcal{C}}}\right)^{\sum \ell_{\mathcal{C}}}$. Note that $x^{1/x} \leq 2$ for $x \geq 2$. Let $x = \frac{2 \sum \ell_{\mathcal{C}} + \sum h_{\mathcal{C}}}{\sum \ell_{\mathcal{C}}}$, we have $\left(\frac{2 \sum \ell_{\mathcal{C}} + \sum h_{\mathcal{C}}}{\sum \ell_{\mathcal{C}}}\right)^{\sum \ell_{\mathcal{C}}} = x^{\frac{1}{x}(2 \sum \ell_{\mathcal{C}} + \sum h_{\mathcal{C}})} \leq 2^{(2 \sum \ell_{\mathcal{C}} + \sum h_{\mathcal{C}})}$. Thus, $\log \prod (2 + \frac{h_{\mathcal{C}}}{\ell_{\mathcal{C}}})^{\ell_{\mathcal{C}}} \leq 2 \sum \ell_{\mathcal{C}} + \sum h_{\mathcal{C}}$. Let L_j be the set of all centroid paths with level j, $j \leq \log n$. For any two centroid paths in L_j, their *lex-order* arrays are disjoint, so $\sum_{\mathcal{C} \in L_j} h_{\mathcal{C}} \leq n$. There are at most $\log n$ levels, so $\sum_{\mathcal{C} \in \Delta(T)} h_{\mathcal{C}} \leq n \log n$. □

Compressing the *label* **arrays.** Unlike *lex-order*, the *label* array is not an increasing sequence. To compress *label*, we simulate it by other "simpler" arrays. For a centroid path $\mathcal{C}$, let $s'_1, s'_2, \ldots, s'_\ell$ be the modified suffixes generated according to $\mathcal{C}$, in increasing lexicographical order. Assume that t side trees hang from $\mathcal{C}$. We store the following information.

- *st-rank*$_{\mathcal{C}}[1..\ell]$: Suppose s'_i is generated by the suffix s in T. Then *st-rank*$_{\mathcal{C}}[i]$ stores the side-tree-rank of s, i.e., the rank of the side tree containing s.
- *tree_pointer*$_{\mathcal{C}}[1..t]$: *tree_pointer*$_{\mathcal{C}}[j]$ points to the j-th largest side tree of $\mathcal{C}$ in T, ties are broken arbitrarily.

– $modified\_rank_{\mathcal{C},v}[1..|T_v|]$, for each side-tree T_v of $\mathcal{C}$ in T: $modified\_rank_{\mathcal{C},v}[j] = i$ if the j-th suffix in T_v generates s'_i.

To find $label_{\mathcal{C}}[i]$, note that $tree\_pointer_{\mathcal{C}}[st\text{-}rank_{\mathcal{C}}[i]]$ returns the side tree T_v hanging from $\mathcal{C}$ which contains the suffix that generates s'_i. We perform a $rank(i)$ query on $unmodifed\_rank_{\mathcal{C},v}$, where $rank(i)$ returns j if $modified\_rank_{\mathcal{C},v}[j] = i$. Thus, $label_{\mathcal{C}}[i]$ is the j-th suffix in T_v. Let w_v be the number suffixes on the left of v in T. Then $label_{\mathcal{C}}[i] = SA[w_v + j]$.

By Lemma 1, the $st\text{-}rank_{\mathcal{C}}$ arrays for all centroid paths $\mathcal{C}$ can be represented in $O(n \log n)$ bits using variable size encoding. We can build a *select* data structure [15] on the arrays, which uses $O(n \log n)$ bit, to support $O(1)$ time access to each entry. The *tree_pointer* arrays contain only n pointers in total and take $O(n \log n)$ bits over all centroid paths.

Lemma 10. *We can store $modified\_rank_{\mathcal{C},v}$ array to support the $rank(i)$ query in $O(1)$ time: given any integer i, return j if $modified\_rank_{\mathcal{C},v}[j] = i$; return null otherwise. Total space required over all $\mathcal{C} \in \Delta(T)$ and all side trees T_v hanging from $\mathcal{C}$ is $O(n \log n)$ bits.*

Proof. Consider any centroid path $\mathcal{C}$ and a side tree T_v hanging from $\mathcal{C}$. The sequence $modified\_rank_{\mathcal{C},v}$ is strictly increasing and ranges from 1 to $|T_{\mathcal{C}}|$, hence it can be stored in $O(|T_v| \log \frac{|T_{\mathcal{C}}|}{|T_v|})$ bits while supporting the *rank* query [18]. Let $f(T_{\mathcal{C}})$ denote the total space required to store the *modified_rank* arrays for all centroid paths with root in $T_{\mathcal{C}}$, including $\mathcal{C}$. Let $T_{v_1}, T_{v_2}, \ldots, T_{v_t}$ be side trees hanging from $\mathcal{C}$. Note that $f(T_{\mathcal{C}}) \leq \sum_{i=1}^{t} \left(O(|T_{v_i}| \log \frac{|T_{\mathcal{C}}|}{|T_{v_i}|}) + f(T_{v_i})\right)$. Resolving this recurrence, we have $f(T_{\mathcal{C}}) = O(|T_{\mathcal{C}}| \log |T_{\mathcal{C}}|)$ for any $\mathcal{C}$. Therefore, all *modified_rank* arrays in T can be stored in $O(n \log n)$ bits. □

In conclusion, Lemma 9 and 10 show that the *lex-order* and *label* arrays can be represented in $O(n \log n)$ bits and support $O(1)$ time retrieval. Together with the matching algorithm of Section 3.2, Lemma 5 follows.

Extending to edit distance. We handle each type of edit operations separately. Substitution is handled by the above data structure. To find substrings of S that matches P with one insertion (to the substrings), we generate another type of modified suffixes, which we called the *insertion suffixes*. Precisely, let $\mathcal{C}$ be a centroid path in the suffix tree T. Let s be a suffix in T passing through the root of $\mathcal{C}$, and diverging from $\mathcal{C}$ at a node u on $\mathcal{C}$. We create an insertion suffix s' by inserting a character c to s after u, where c is the first character on the core edge out of u. We say that s generates an insertion suffix s' according to $\mathcal{C}$. Given a pattern P, finding the 1-error matches can be reduced to a number of prefix matching queries on the insertion suffixes and LCP queries. By handing the prefix matching queries using the same techniques as shown, we find all 1-error matches for insertion in the $O(m + occ' + \log n \log \log n)$ time, where occ' is the number of matches found. Handling deletion is identical. The total space for the data structures is $O(n \log n)$ bits.

4 An $O(n \log n)$-Bit Index for 2-Error

Given a pattern $P[1..m]$, we can find its 2-error matches in S as follows: First modify P at each position $P[i]$, substituting it with a character $c \neq P[i]$. Denote the modified pattern as $P_{i,c}[1..m]$. Then, find all 1-error matches of $P_{i,c}$ with the error in $P_{i,c}[1..i-1]$. By trying all $i = 1, \ldots, m$ and each possible $c \in \Sigma$, each 2-error match of P is found exactly once.

To support the above operations, we store the $O(n \log n)$-bit index for 1-error matching, as well as some $O(n \log n)$-bit auxiliary data structures (to be defined). Then, to find all 2-error matches of a pattern $P[1..m]$, we perform the followings for every $i = 1, \ldots, m$ and for every $c \in \Sigma$.

1. Search $P_{i,c}$ in T to identify the centroid paths and side edges $P_{i,c}$ overlaps.
2. Search $P_{i,c}$ in the sampled 1-error tree N to identify an interval $[d, e]$ corresponding to modified suffixes in N with $P_{i,c}$ as a prefix.
3. Find the 1-error matches of $P_{i,c}$ where the error is in $P[1..i-1]$ and is on a side edge. This is done by performing an LCP query in T for each side edge $P_{i,c}$ overlaps.
4. Find the 1-error matches of $P_{i,c}$ where the error is in $P[1..i-1]$ and is on a centroid path. We follow the approach in Section 3.2, generating candidates and verifying them for correct matches.

By preprocessing P (but not $P_{i,c}$) in $O(m)$ time, we can perform Step 1 in $O(\log \log n + w)$ time, where $w \leq 2 \log n$ is the number of centroid paths and side edges $P_{i,c}$ overlaps. We can also build an $O(n \log n)$-bit LCP data structure for N so that Step 2 takes $O(\log \log n)$ time. Step 3 can be done in $O(\log n \log \log n + \#output)$ time.

The main challenge is Step 4. Generating candidates involves *range_search* queries on the $rank_{\mathcal{C}}$ arrays, and the candidates generated may include an unbounded number of modified suffixes having $P_{i,c}$ as a prefix but their modified position is not in $P_{i,c}[1..i-1]$. Thus, for each centroid path $\mathcal{C}$, we store a $modified_{\mathcal{C}}$ array storing the location of the modified character of each modified suffix generated according to $\mathcal{C}$. We then use a Bounded Value Range Query (BVRQ) data structure [16] to ensure generating candidates with the required error positions, plus at most $2 \log n$ counterfeits. Finally, we verify each candidate in $O(1)$ time. We can show that the $modified_{\mathcal{C}}$ arrays and the BVRQ data structures take totally $O(n \log n)$ bits and Step 4 takes $O(\log n \log \log n + \#output)$ time. So, we have the following lemma.

Lemma 11. *We can build an $O(n \log n)$-bit index for $S[1..n]$. Given a pattern $P[1..m]$, we can preprocess P in $O(m)$ time. For any modified pattern $P_{i,c}$, we can find all 1-error matches of $P_{i,c}$ with the error in $P_{i,c}[1..i-1]$ in $O(\log n \log \log n + occ_{i,c})$ time, where $occ_{i,c}$ is the number of matches found.*

By repeating the search for each $P_{i,c}$, $i \leq m$ and $c \in \Sigma$, we can find all 2-error matches of P in $O(m \log n \log \log n + occ)$ time, where occ is the number of 2-error matches.

References

1. A. Amir, D. Keselman, G. M. Landau, M. Lewenstein, N. Lewenstein, and M. Rodeh. Indexing and dictionary matching with one error. In *Proceedings of Workshop on Algorithms and Data Structures*, 1999, pages 181–192.
2. A. L. Buchsbaum, M. T. Goodrich, and J. R. Westbrook. Range searching over tree cross products. In *European Symposium on Algorithms*, 2000, pages 120–131.
3. E. Chavez and G. Navarro. A metric index for approximate string matching. In *Proceedings of Latin American Theoretical Informatics*, 2002, pages 181-195.
4. A. Cobbs. Fast approximate matching using suffix trees. In *Proceedings of Symposium on Combinatorial Pattern Matching*, 1995, pages 41–54.
5. R. Cole, L. A. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proceedings of Symposium on Theory of Computing*, 2004, pages 91–100.
6. H. L. Chan, T. W. Lam, W. K. Sung, S. L. Tam, and S. S. Wong. A Linear-Size Index for Approximate Pattern Matching. To appear in *CPM*, 2006.
7. P. Ferragina and G. Manzini. Opportunistic Data Structures with Applications. In *Symposium on Foundations of Computer Science*, 2000, pages 390–398.
8. R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. In *Proceedings of Symposium on Theory of Computing*, 2000, pages 397–406.
9. D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338-355, 1984.
10. T. N. D. Huynh, W. K. Hon, T. W. Lam, and W. K. Sung. Approximate string matching using compressed suffix arrays. In *Proceedings of Symposium on Combinatorial Pattern Matching*, 2004, pages 434–444.
11. T. W. Lam, W. K. Sung, S. S. Wong. Improved approximate string matching using compressed suffix data structures. In *Proceedings of International Symposium on Algorithms and Computation*, 2005.
12. M. G. Maaß and J. Nowak. Text indexing with errors. Technical Report TUM-10503, Fakultät für Informatik, TU München, Mar. 2005.
13. U. Manber and G. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
14. E. M. McCreight. A Space-economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262–272, 1976.
15. J. I. Munro. Tables. In *Proceedings of Conference on Foundations of Software Technology and Computer Science*, 1996, pages 37–42.
16. S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proceedings of 13th Symposium on Discrete Algorithms*, 2002, pages 657–666.
17. G. Navarro and R. Baeza-Yates. A Hybrid Indexing Method for Approximate String Matching. *J. Discrete Algorithms*, 1(1):205-209, 2000.
18. R. Raman, V. Raman, and S. S. Rao Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms*, 2002, pages 233–242.
19. K. Sadakane. Succinct representations of *lcp* information and improvements in the compressed suffix arrays. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms*, 2002, pages 225–232.
20. P. Weiner. Linear Pattern Matching Algorithms. In *Proceedings of Symposium on Switching and Automata Theory*, 1973, pages 1–11.
21. D. E. Willard. Log-Logarithmic worst-case range queries are possible in space $\Theta(n)$. *Information Processing Letters*, 17(2):81–84, 1983.

Traversing the Machining Graph

Danny Z. Chen[1,⋆], Rudolf Fleischer[2,⋆⋆], Jian Li[2],
Haitao Wang[2], and Hong Zhu[2,⋆⋆⋆]

[1] Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556, USA
dchen@cse.nd.edu

[2] Department of Computer Science and Engineering, Shanghai Key Laboratory of Intelligent Information Processing, Fudan University, Shanghai, China
{rudolf, lijan83, wanghaitao, hzhu}@fudan.edu.cn

Abstract. Zigzag pocket machining (or $2D$-milling) plays an important role in the manufacturing industry. The objective is to minimize the number of tool retractions in the zigzag machining path for a given *pocket* (i.e., a planar domain). We give an optimal linear time dynamic programming algorithm for simply connected pockets, and a linear plus $O(1)^{O(h)}$ time optimal algorithm for pockets with h holes. If the dual graph of the zigzag line segment partition of the given pocket is a partial k-tree of bounded degree or a k-outerplanar graph, for a fixed k, we solve the problem optimally in time $O(n \log n)$. Finally, we propose a polynomial time algorithm for finding a machining path for a general pocket with h holes using at most $OPT + \epsilon h$ retractions, where OPT is the smallest possible number of retractions and $\epsilon > 0$ is any constant.

1 Introduction

$2D$*-milling*, or *zigzag pocket machining (*ZPM*)*, is an important problem in the manufacturing industry [13, 21, 24]. Either a workpiece is translated under a spinning milling tool, or a cutter is moved across the workpiece. We model the workpiece as an arbitrary planar domain, called a *pocket*. The actual shape of the pocket is not really important for us, so we may just think of a polygon. Usually, the cutter (or the workpiece moving below the milling tool) can only cut while

⋆ The research of this author was supported in part by a grant from the Shanghai Key Laboratory of Intelligent Information Processing, Fudan University, Shanghai, China and by the US National Science Foundation under Grant CCF-0515203. This work was partially done while the author was visiting the Shanghai Key Laboratory of Intelligent Information Processing at Fudan University, China.

⋆⋆ The second and last authors gratefully acknowledge support from the National Natural Science Fund China (grant nos. 60496321, 60373021, and 60573025) and the Shanghai Science and Technology Development Fund (grant no. 03JC14014).

⋆⋆⋆ The order of the authors follows the international standard of alphabetic order of the last names. In China, where the first-authorship is a very important aspect of a publication, the order of the authors should be Jian Li, Haitao Wang, Danny Z. Chen, Rudolf Fleischer, and Hong Zhu.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 220–231, 2006.

moving along a fixed direction, for example, parallel to the x-axis (but it can cut moving in both directions along the line). When it reaches the boundary of the workpiece, it must either move along the boundary to another line parallel to the x-axis (usually the lines are assumed to be equally spaced with a *zigzag step-over distance* $\delta > 0$; again, this technical requirement is not important to us), creating a zigzag movement pattern, or it must jump to another part of the workpiece. A jump requires the cutter to be *retracted*. Since retractions are time-consuming (and may have other disadvantages due to technological problems), the goal is to find a machining path minimizing the number of retractions, under the additional constraint that the cutter must work on any part of the workpiece exactly once while it cannot traverse any part of the boundary more than once.

Considerable work has been done on ZPM, see [24] for an extensive survey of the current state-of-the-art. A few algorithms were given in [9, 16], but they did not attempt to minimize the number of tool retractions or to optimize any other criteria. Some heuristic methods were used to reduce the number of tool retractions for general pockets [12, 13]. For pockets with holes, ZPM was shown to be NP-hard by Arkin et al. [3] by a reduction from the *Planar 3-Satisfiability Problem* [17]. They also presented a linear time approximation algorithm with at most $5OPT + 6h$ tool retractions based on a graph model, called the *machining graph*, where OPT is the smallest possible number of retractions and h is the number of holes in the pocket. In the full paper, we will describe how to modify their algorithm to achieve at most $3OPT + 3h$ tool retractions in linear time. Tang et al. [22] studied a special case when the step-over distance is small with respect to the geometry of the pocket. However, no quantitative measure was given on how small the step-over distance needs to be in order for the optimal solution to hold (see [14]).

Tang and Joneja [23] presented a linear time approximation algorithm for ZPM with at most $OPT + h + N_r$ retractions, where N_r is the number of the so-called *reducible blocks* of the pocket. Although the coefficients of OPT and h are both one, the third parameter N_r can be quite large, depending on the shape of the given pocket, the step-over distance, the inclination of the reference line, etc. For example, N_r will usually grow with a larger step-over distance. Thus, the results in [3] and in [23] are not directly comparable. Some practical implications and applications of ZPM were discussed in [13, 22].

It is worth pointing out that although ZPM for pockets with holes is NP-hard [3], for the important case of simply connected pockets (i.e., without holes), only approximation algorithms were previously known [3, 23]. In fact, it was an open problem to decide whether the case of a simply connected pocket is NP-hard.

Other optimization criteria for ZPM have also been considered. For example, multitool retraction minimization was studied in [6]. The problem of minimizing the total length of the tool path was studied in [2]. Algorithms for determining a cutting direction in order to minimize the tool retraction length were given in [15, 19]. Some algorithms were designed to optimize the tool path length and the number of tool retractions [1, 20]. A survey of the pocketing requirements can be found in [11].

In this paper, we present significantly improved algorithms for ZPM. Our techniques are different from the previous ones [3, 23], and our algorithms are superior in theoretical performance, and may even be interesting for practical applications (e.g., for k-outerplanar dual graphs, with a small k).

(1) We give an optimal linear time dynamic programming algorithm for simply connected pockets, settling the open question of the complexity of this case.
(2) For pockets with holes, we introduce the concept of the *boundary graph* to remodel the problem, and generalize our dynamic programming approach to optimally solve in time $O(n \log n)$ the cases when the dual planar graph of the pocket is a partial k-tree of bounded degree or a k-outerplanar graph, for any fixed k.
(3) Combining the ingredients of (1) and (2) with the approximation scheme for planar graphs in [7], we develop a polynomial time approximation algorithm for finding a machining path with at most $OPT + \epsilon h$ retractions for a pocket with h holes, for any constant $\epsilon > 0$. This substantially improves the previous approximation solutions [3, 23], and in fact gives a best possible approximation if the output quality is measured in terms of the number of holes.
(4) We give an exact dynamic programming algorithm running in linear plus $O(1)^{O(h)}$ time for a pocket with h holes. This implies that, in particular, ZPM with a logarithmic number of holes is still solvable in polynomial time.

The rest of this paper is organized as follows. In Section 2, we review some definitions from [3] and state the problem formally. In Section 3, we describe an optimal linear time dynamic program for simply connected pockets. In Section 4, we first introduce the concept of a boundary graph and the MRPC problem, and then present exact polynomial time algorithms if the dual graph of a pocket with holes is a partial k-tree of bounded degree or a k-outerplanar graph. In Section 5, we present a "best possible" polynomial time approximation algorithm for the general problem.

2 Preliminaries

We mainly use the terminology from [3]. A *pocket* P is a compact connected planar domain bounded by a contour B. It is *simply connected* (e.g., a simple polygon) if it contains no holes, or *multiply connected* otherwise. For a pocket with h holes, B consists of $h + 1$ unconnected loops (the boundary of the outer face and the boundaries of the h holes). The edges of B can be straight line segments or any types of simple curves.

Consider an arbitrary set of non-crossing line segments (it could even be curves) partitioning P into a set of regions. In 2D-milling, the line segments would all be parallel and equally-spaced. Let N be the set of all endpoints of the line segments, and let n denote the size of N. On the node set N we define the undirected *machining graph* $M_P = (N, E)$, with two types of edges: (1) *compulsory* edges (*c-edges*) connecting the two endpoints of a line segment; (2)

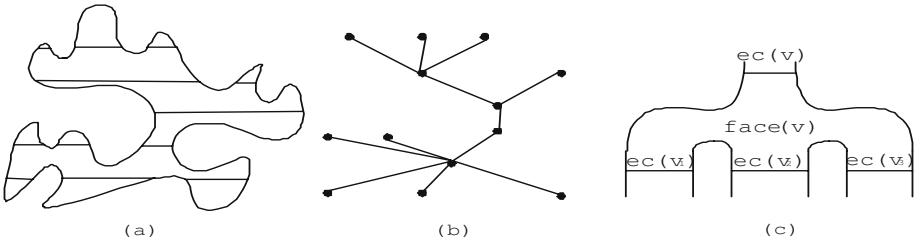


Fig. 1. (a) A simple pocket. (b) Its dual graph is a tree. (c) A face in a simple pocket.

non-compulsory edges (*nc-edges*) connecting two neighboring nodes on B, if they are not already connected by a c-edge. Note that every node in N is incident to exactly one c-edge and at most two nc-edges.

Given a machining graph M_P, a *tool traversing path* (or *machining path*) is a collection $\mathcal{P}$ of simple vertex-disjoint paths in M_P, called *no-retraction paths*, such that (1) every c-edge is traversed exactly once, and (2) every nc-edge is traversed at most once. The machining tool must follow all no-retraction paths of $\mathcal{P}$. When it reaches the end of a path, it *jumps* to an unprocessed no-retraction path. This operation is called a *retraction*. The number of retractions is one less than the number of no-retraction paths in $\mathcal{P}$. An *optimal* (or minimum) machining path $\mathcal{P}$ minimizes the number of retractions (or equivalently, the number of no-retraction paths in $\mathcal{P}$). If the pocket is multiply connected, then finding an optimal machining path is NP-hard [3].

The c-edges and nc-edges of a machining graph M_P induce a planar partition P_G of the pocket P. The faces F of P_G induce the dual graph $D_P = (F, E_d)$ of P_G (see Fig. 1(b)).

Throughout this paper, we denote by OPT the minimum number of retractions of all feasible machining paths for P, the number of holes in P by h, the number of nodes in M_P by n, and the number of edges in M_P by m. We also abuse the notation by calling D_P the dual graph of M_P.

3 An Optimal Algorithm for Simply Connected Pockets

In this section, we optimally solve `ZPM` for a simply connected pocket P in linear time based on dynamic programming. We first discuss some properties of an optimal traversal of M_P and then present our new algorithm. Observe that D_P is a tree if and only if P is a simply connected pocket. Figs. 1(a) and 1(b) show an example. For a vertex $v \in F$, let *face*(v) denote the corresponding face in P_G.

Lemma 1 [3]. *There exists an optimal machining path $\mathcal{P}$ such that*

1. *each no-retraction path in $\mathcal{P}$ starts and ends with a c-edge, and*
2. *no two nc-edges are traversed consecutively.* □

We will compute a machining path satisfying the conditions of Lemma 1. We treat $D_P = (F, E_d)$ as a tree T rooted at an arbitrarily chosen node *root* $\in F$ of

degree one. For a node $v \in F$, let T_v denote the subtree of T rooted at v, and $P_{G/v}$ the portion of the partition P_G corresponding to T_v. If $v \neq \mathit{root}$, then the boundary of $\mathit{face}(v)$ contains a c-edge, $ec(v)$, of M_P separating $\mathit{face}(v)$ from the face of v's parent in T. Let M_v denote the machining subgraph of $P_{G/v}$, where we require that $ec(v)$ is a c-edge of M_v. The two endpoints of $ec(v)$ are left_v and right_v, where the former one is the endpoint we reach first when we walk counterclockwise along B, starting somewhere in $\mathit{face}(\mathit{root})$.

We compute optimal machining paths for $P_{G/v}$ for all nodes v of T in a bottom-up manner, starting at the leaves. This will give us a linear time algorithm. Fig. 1(c) shows the situation for an internal node v with three children. In general, v may have k children $v_1, \ldots, v_k$, where we assume that $ec(v_1), \ldots, ec(v_k)$ appear counterclockwise along the boundary of $\mathit{face}(v)$. Let $\mathit{bridge}(v_i)$ denote the nc-edge connecting $ec(v_i)$ and $ec(v_{i+1})$ for $1 \leq i < k$. Note that we have already computed the machining paths for the subtrees rooted at $v_1, \ldots, v_k$, respectively. Now we must extend these paths to integrate the edge $ec(v)$.

There are a few cases. If left_{v_1} or right_{v_k} is the endpoint of a no-retraction path, we can extend this path to include $ec(v)$. If both these points are endpoints of two different paths, we can even connect both paths via $ec(v)$, thus saving one retraction. If both points are not path endpoints, $ec(v)$ will form a new path by itself. Since we do not know in advance which case can yield an optimal solution of M_P, we must provide for all cases, i.e., we use dynamic programming.

Let $M_{i,j} = \cup_{\ell=i}^{j} M_{v_\ell} \bigcup \cup_{\ell=i}^{j-1} \mathit{bridge}(v_\ell)$, for $1 \leq i \leq j \leq k$, be the connected portion of M_P formed by $M_{v_i}, \ldots, M_{v_j}$ and the bridges connecting the pieces. We characterize all feasible machining paths for $M_{i,j}$ into five classes. $P^0_{i,j}$ contains all machining paths such that no no-retraction path ends at left_{v_i} or right_{v_j}; $P^l_{i,j}$ contains all paths such that a no-retraction path ends at left_{v_i} but no such path ends at right_{v_j}; $P^r_{i,j}$ contains all paths such that a no-retraction path ends at right_{v_j} but no such path ends at left_{v_i}; finally, $P^{lr}_{i,j}$ contains all paths such that some no-retraction paths end at left_{v_i} and right_{v_j}; actually, we must divide the last class further into $P^{lr1}_{i,j}$, where the same no-retraction path connects both points, and $P^{lr2}_{i,j}$, where two different no-retraction paths end at the two points.

Let $h^x(v_i, v_j)$, for $x \in \{0, l, r, lr1, lr2\}$, be the minimum number of retractions among all machining paths in $P^x_{i,j}$. Then the optimal value for M_P is $\min\{h^0(\mathit{root}, \mathit{root}), h^l(\mathit{root}, \mathit{root}), h^r(\mathit{root}, \mathit{root}), h^{lr1}(\mathit{root}, \mathit{root}), h^{lr2}(\mathit{root}, \mathit{root})\}$.

Initially, for any leaf v of T, $h^{lr1}(v, v) = 0$ and $h^0(v, v) = h^l(v, v) = h^r(v, v) = h^{lr2}(v, v) = \texttt{NULL}$, where NULL means there is no machining path in this class. If a term NULL appears in an arithmetic expression, the expression has value NULL. It is straightforward to compute these values iteratively for an internal node v and all pairs of indices $i \leq j$ if the values for all children of v are known. For example, $h^r(v_i, v_{j+1}) = \min\{h^r(v_i, v_j) + h^{lr1}(v_{j+1}, v_{j+1})$ (we use $\mathit{bridge}(v_j)$), $h^r(v_i, v_j) + h^r(v_{j+1}, v_{j+1}) + 1$ (we cannot use $\mathit{bridge}(v_j)$), $h^0(v_i, v_j) + h^{lr1}(v_{j+1}, v_{j+1}) + 1$, $h^0(v_i, v_j) + h^r(v_{j+1}, v_{j+1}) + 1\}$.

The other recursive formulas are similar. The values $h^x(v, v)$ can be computed as $h^{lr1}(v, v) = 1 + \min_{x \in \{0,l,r,lr1,lr2\}}\{h^x(v_1, v_k)\}$, $h^{lr2}(v, v) = \texttt{NULL}$, $h^r(v, v) =$

$\min_{x\in\{l,lr1,lr2\}}\{h^x(v_1,v_k)\}$, $h^l(v,v)=\min_{x\in\{r,lr1,lr2\}}\{h^x(v_1,v_k)\}$, and $h^0(v,v) = h^{lr2}(v_1,v_k) - 1$.

Theorem 2. *The dynamic program computes an optimal machining path for simply connected pockets in linear time.* □

If P has $h > 0$ holes, the dual graph D_P is no longer a tree. However, we can identify $O(h)$ *pivot nodes* such that the removal of these nodes results in a forest of trees such that each tree is adjacent to a constant number of pivot nodes (for details see the full version of the paper). The trees of the forest can be handled similarly as in Section 3. Since each tree has only a constant size interface with the pivot nodes, we can test all possible choices for the c-edges of these nodes in $O(1)^{O(h)}$ time. This implies that the problem is still solvable in polynomial time for pockets with $h = O(\log(n + m))$ holes.

Theorem 3. *We can find an optimal machining path for a pocket with h holes in time* $O(n + m) + O(1)^{O(h)}$. □

4 The MRPC Problem

Note that even for a pocket with holes, its dual graph is a planar graph embedded in the plane. In this section, we investigate certain types of planar dual graphs. These special cases will be a key to our "best possible" approximation algorithm for the general case in Section 5.

Let the *boundary graph* $B_P = (N_B, E_B)$ of M_P be the graph obtained when we contract every c-edge into a single node, deleting self-loops and multiple edges. Note that B_P is a planar graph of maximum degree four. Its edges are exactly the nc-edges of M_P. Not every path in B_P is a feasible no-retraction path in M_P, because such a path should not use two consecutive nc-edges, i.e., on a path in B_P we cannot leave a node on an arbitrary other edge since some edge is prohibited. We call a path in B_P *valid* if and only if it corresponds to a feasible no-retraction path in M_P. We call pairs of edges that can lie consecutively on a valid path *consistent*. The *minimum restricted path cover problem (*MRPC*)* is thus the problem of finding a set of vertex-disjoint simple valid paths in B_P such that each node of B_P lies on exactly one path. Our goal is to minimize the number of such paths.

We assume that the reader is familiar with the definitions of k-trees and bounded tree-width (see, for example, [10]). We follow the terminology and the dynamic programming framework developed in [5]. We will now give an algorithm assuming that the boundary graph B_P has bounded tree-width and that a k-tree is given together with its reduction sequence. We will later relax these assumptions.

For simplicity, we use the following notation: For a vertex v, $K = K(v)$ is the set of neighbors of v when v becomes a k-leaf during the k-tree reduction process. Let $K' = K \cup \{v\}$ and $K^u = K' - \{u\}$, for any vertex $u \in K'$. $B(K)$ is, at all times, the set of vertices in the currently removed branches on K. R is the root clique.

We now extend the simple dynamic program from Section 3, where we worked bottom-up in a tree, to k-trees. We keep a state for each K. This state information for K in some sense represents the equivalence classes of the solutions (usually for a slight generalization of the original problem) on the subgraph induced by K and all its descendants. In Section 3 we distinguished between classes P^0, P^l, P^r, P^{lr1}, and P^{lr2}. In a k-tree, the interface between a node and its descendants is more complicated because a k-leaf is connected to a k-clique instead of a single parent node. Still, for fixed k, the number of possible equivalence classes is a (large) constant.

We use an *index set* to denote the state. The index set $C(K)$ for K is a set of solutions to the problem on the subgraph induced by $K \cup B(K)$. An index $c \in C(K)$ is an equivalence class of the solutions. The value of K with index c, $s(c, K)$, is the optimum value of the solutions represented by c, and we call this value the *state value*. They have a similar meaning as the functions h^0, h^l, h^r, h^{lr1}, and h^{lr2} in Section 3.

If a vertex v is removed, we must update the state value $s(c, K)$ for every $c \in C(K)$. The update is based on the state values for all K^u ($u \in K'$) and reflects each K^u's influence on the state value of K for a particular index $c \in C(K)$. This procedure is normally called *combining removed branches*, since the removal of v means that we need to combine v with the already removed branches on K.

Now we define the index set for our algorithm. Let a *ve-pair* be a pair (v, e), where v is an end vertex of an edge e in the boundary graph B_P. Let $asc(v)$ denote the edge e in the ve-pair (v, e). We say that two ve-pairs (v_1, e_1) and (v_2, e_2) are *disjoint* if $v_1 \neq v_2$. Intuitively, a no-retraction path can only enter and then again leave the interface K on a pair of disjoint ve-pairs.

The state of K is indexed by a set of triples (D, S, I), where D is a set of mutually disjoint (unordered) pairs of ve-pairs (a path enters and leaves K), S is a set of disjoint ve-pairs (a path ends in K), and I is a set of the 'touched' vertices (internal vertices of a path) in K that are disjoint from D and S. Let $D = \{((v_{i1}, e_{i1}), (v_{i2}, e_{i2})) \mid 1 \leq i \leq |D|\}$, and $S = \{(v_i, e_i) \mid 1 \leq i \leq |S|\}$. Further, let $V(D) = \{(v_{i1}, v_{i2}) \mid 1 \leq i \leq |D|\}, E(D) = \{(e_{i1}, e_{i2}) \mid 1 \leq i \leq |D|\}$, and $V(S) = \{v_i \mid 1 \leq i \leq |S|\}$, and $E(S) = \{e_i \mid 1 \leq i \leq |S|\}$.

A partial solution of index $c = (D, S, I) \in C(K)$ means a set of $|D|$ disjoint simple valid paths with both endpoints in K, a set of $|S|$ disjoint simple valid paths with only one endpoint in K, and some other simple valid paths with no endpoint in K in the subgraph induced by $K \cup B(K)$, such that no two consecutive internal vertices of a path are both in K and these paths should cover all vertices of $B(K)$. The state value $s((D, S, I), K)$ is a positive integer that is the minimum number of valid paths covering $K \cup B(K)$ under the restrictions of index (D, S, I), or is `NULL` if no such valid paths exist.

The state values $s(c, K)$ are initially `NULL` for all c and K. Upon the removal of v, we update $s((D, S, I), K)$, where (D, S, I) arises from a set of $k + 1$ triples $(D_u, S_u, I_u) \in C(K^u)$, one for every $u \in K'$, such that the following conditions (i)–(v) are satisfied. The case for removing the root needs a little more work and

will be described later separately. Intuitively, satisfying these conditions means that we can combine the partial solutions for different K^u's for $u \in K'$.

(i) $I_u \cap I_w = \emptyset$ if $u \neq w$, $(\cup_{p \in V(D_u)}\{p\}) \cap I_w = \emptyset$ for $u, w \in K'$, and $V(S_u) \cap I_w = \emptyset$ for $u, w \in K'$.
(ii) A vertex pair $d \in V(D)$ occurs at most once.
(iii) Consider the multi-set $M = \bigcup_{u \in K'}(\cup_{p \in V(D_u)}\{p\} \cup V(S_u))$ (multiplicity is counted). Every vertex $u \in K'$ appears at most twice in M.
(iv) For a vertex v that appears twice in M, consider the two ve-pairs (each contributing a v to M) (v, e_1) and (v, e_2); then e_1 and e_2 must be consistent.
(v) The graph $F = (K', \cup_{u \in K'} V(D_u))$ has no cycle, i.e., F is a set of paths and isolated vertices.

We define (D', S', I') as follows. Consider a path in F. Suppose its two endpoints are v_1 and v_2. If the multiplicities of v_1 and v_2 in M are both one, then $((v_1, asc(v_1)), (v_2, asc(v_2))) \in D'$. If only one of v_1 and v_2 has multiplicity one in M, say v_1, then $(v_1, asc(v_1)) \in S'$ and $v_2 \in I'$. Moreover, I' contains the union of the I_u's and the interior vertices of the paths of F.

Next we show how to compute (D, S, I) from (D', S', I') and update the state value $s((D, S, I), K)$. It is possible that we obtain more than one (D, S, I) from a single (D', S', I'), and we update the state value for every (D, S, I) obtained. Recall that $s((D, S, I), K)$ is initially $NULL$. Once we obtain a (D, S, I) from (D', S', I') and some preliminary state value for it, called a *p-value*, we replace the value of $s((D, S, I), K)$ by the p-value if it is smaller than the current value of $s((D, S, I), K)$. (D, S, I) is obtained from (D', S', I') in one of the following ways. Suppose we are currently removing v. Denote the p-value of (D, S, I) by $pv = \sum_{u \in K'} s((D^u, S^u, I^u), K) - \#$(elements with multiplicity two in M).

We distinguish four cases. (1) If $v \in I'$, then $I = I' - \{v\}$ and $D = D'$, $S = S'$. (2) If $v \in V(S')$, then $S = S' - \{(v, asc(v))\}$, $D = D'$, and $I = I'$. There is a vertex $v_1 \in K'$ that is adjacent to v and $v_1 \notin I'$. If $v_1 \in V(S')$ and $asc(v_1)$ and $asc(v)$ are consistent with (v, v_1), then $S = S' - \{(v, asc(v)), (v_1, asc(v_1))\}$, and $pv = pv - 1$. If $(v_1, v_2) \in V(D')$ and $asc(v_1)$ and $asc(v)$ are consistent with (v, v_1), then $D = D' - \{((v_1, asc(v_1)), (v_2, asc(v_2)))\}$, $S = S - \{(v, asc(v))\} \cup \{(v_2, asc(v_2))\}$, $I = I' \cup \{v_1\}$, and $pv = pv - 1$. If v_1 is neither in $V(S')$ nor in any pair of $V(D')$ and $asc(v)$ is consistent with (v, v_1), then $S = S' - \{(v, asc(v))\} \cup \{(v_1, (v, v_1))\}$. (3) If $(v, v_1) \in V(D')$ for some v_1, then $D = D' - \{((v, asc(v)), (v_1, asc(v_1)))\}$, $S = S' \cup \{(v_1, asc(v_1))\}$, and $I = I'$. There is a vertex $v_2 \in K'$ that is adjacent to v and $v_2 \notin I'$. If $v_2 \in V(S')$ and $asc(v_2)$ and $asc(v)$ are consistent with (v, v_2), then $D = D' - \{((v, asc(v)), (v_1, asc(v_1)))\}$, $S = S' \cup \{(v_1, asc(v_1))\}$, and $pv = pv - 1$. If $(v_2, v_3) \in V(D')$ and $asc(v_2)$ and $asc(v)$ are consistent with (v, v_2), then $D = D' - \{((v, asc(v)), (v_1, asc(v_1))), ((v_2, asc(v_2)), (v_3, asc(v_3)))\} \cup \{((v_1, asc(v_1)), (v_3, asc(v_3)))\}$, $I = I' \cup \{v_2\}$, and $pv = pv - 1$. If v_2 is neither in $V(S')$ nor in any pair of $V(D')$ and $asc(v)$ is consistent with (v, v_2), then $D = D' - \{((v, asc(v)), (v_1, asc(v_1)))\} \cup \{((v_1, asc(v_1)), (v_2, asc(v_2)))\}$. (4) If v is neither in I' nor in $V(S')$ nor in any pair of $V(D')$, then $D = D'$, $S = S'$,

$I = I'$, and $pv = pv + 1$. If v is adjacent to v_1 which is in $K' - I'$, then let $asc(v_1) = (v, v_1)$ and $I' = I' \cup \{v\}$. Add v_1 to the multi-set M, test whether conditions (i)–(v) are all satisfied, obtain (D', S', I'), and then obtain (D, S, I) and pv from (D', S', I') in the same way as above. If v is adjacent to v_1 and v_2 which are both in $K' - I'$, and (v, v_1) and (v, v_2) are consistent with each other, then let $asc(v_1) = (v, v_1)$, $asc(v_2) = (v, v_2)$, and $I' = I' \cup \{v\}$. Add v_1 and v_2 to the multi-set M and augment F with the edge (v_1, v_2). Further, test whether conditions (i)–(v) are all satisfied and obtain (D', S', I') and then (D, S, I) and pv from (D', S', I') in the same way as above.

This finishes the discussion of the non-root case of the `MRPC` algorithm. For the root clique R we do the same as in the non-root case, but we also perform some additional steps. For an obtained index (D, S, I), we try all possible combinations of the unused edges in the subgraph induced by R. We check the validity (i.e., the consistency of all pairs of adjacent edges) of each path, and decide the final value of pv. Note that an isolated vertex in R should be treated as one no-retraction path in the final solution.

Up to now we assumed that B_P is a k-tree. Unfortunately, deciding whether a graph has a tree-decomposition with tree-width at most k is NP-complete [4]. However, for a graph $G = (V, E)$ of (unknown) tree-width k, it is possible to compute a tree-decomposition of G of tree-width at most $3k + 2$ in $O((|V| + |E|) \log(|V| + |E|))$ time [18]. Thus, we can find a tree-decomposition of the boundary graph B_P of at most three times the optimal tree width in time $O(n \log n)$. Then we can compute a reduction sequence of the k-tree for B_P and apply the dynamic program to obtain an optimal solution of `ZPM` in linear time.

Theorem 4. *If the boundary graph B_P of D_P has bounded tree-width, then* `MRPC` *can be solved optimally in time $O(n \log n)$.* □

Lemma 5. *If D_P is a partial k-tree of maximum degree d, then B_P is a partial $(kd + d - 1)$-tree.*

Proof. Suppose $D_P = (F, E_D)$ has a tree-decomposition $(X, T(I, F))$ with $\max_{i \in I} |X_i| \leq k + 1$. We construct a tree-decomposition $(Y, T_B(I_B, F_B))$ of B_P. T and T_{B_P} have the same topology. For a node $i \in I$ representing $X_i \subseteq F$, the corresponding node $Y_i \subseteq E_D$ in I_B is defined as $Y_i = \{e \in E_D \mid e$ has an end vertex in $X_i\}$. Then T_{B_P} is a tree-decomposition of B_P and $\max_{i \in I_B} |Y_i| \leq (k + 1)d$. □

Theorem 6. *If D_P has bounded tree-width and bounded degree, then we can compute an optimal machining path in time $O(n \log n)$.* □

Intuitively, a *k-outerplanar graph* is a planar graph such that nothing remains after we peel away its outer vertices k times. It is known that a k-outerplanar graph has tree-width at most $3k - 1$ [8]. It is easy to see that B_P is $(2k)$-outerplanar if D_P is k-outerplanar.

Theorem 7. *If D_P is a k-outerplanar graph, then we can compute an optimal machining path in time $O(n \log n)$.* □

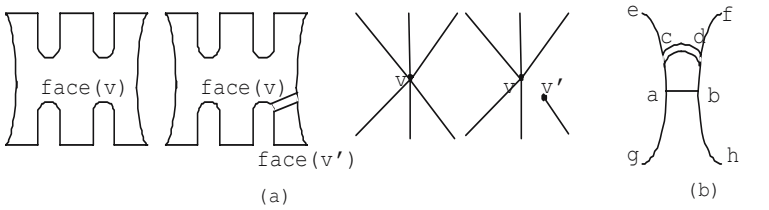


Fig. 2. (a) An edge-cutting operation. (b) Illustrating the proof of Lemma 8.

5 An Approximation Algorithm for Pockets with Holes

This section gives a "best possible" approximation algorithm for the zigzag pocket machining problem for a pocket with h holes. On the dual graph $D_P = (F, E_d)$, we define an *edge-cutting* operation on an edge (v, v_1) at v as follows: Eliminate edge (v, v_1), create a new node v', and add the edge (v', v_1). The corresponding operation on the original machining graph is to divide *face*(v) into two parts. The boundary of one part will contain the c-edge between *face*(v) and *face*(v_1). We denote this new face as *face*(v'). See Fig. 2(a) for an illustration. Note that it may happen that the pocket gets divided into two unconnected regions. Nevertheless, we denote the pocket (or both pockets together) after the cut as $P/\{v, v_1\}$ and its minimum number of retractions for a machining path as $OPT/\{v, v_1\}$.

Lemma 8. *If we cut an edge (v, v_1) in D_P at v, then (1) $OPT/\{v, v_1\} \leq OPT+1$, and (2) from an optimal solution for $P/\{v, v_1\}$ we can obtain a solution for P with at most $OPT/\{v, v_1\} + 1$ retractions.* □

Our approximation algorithm uses a similar approach as the one in [7]. Let k be a fixed integer. The term "face" below refers to the embedding of D_P. We define the *j-th layer* L_j as the set of vertices that are removed in the j-th round of removal if we repeatedly remove the outer vertices of the remaining portion D_j of D_P (initially, $D_1 = D_P$). Let $D(L_j)$ denote the subgraph of D_j induced by the vertices of L_j with the same embedding as D_P, $O(L_j)$ be the outer face of $D(L_j)$, and $I(L_j)$ be the union of the inner faces of $D(L_j)$. Let h_j be the number of faces in $\mathbb{R}^2 - O(L_j) - I(L_{j+1})$ (the portion between layer j and layer $j+1$). Since $\bigcup_{j\geq 1}(\mathbb{R}^2 - O(L_j) - I(L_{j+1})) = \mathbb{R}^2 - O(L_1)$, we have $\sum_j h_j = h$.

Lemma 9. *We can disconnect the boundary of $I(L_j)$ from the boundary of $I(L_{j+1})$ by breaking at most $h_j + h_{j+1}$ edges.*

Proof. W.l.o.g., we assume that $I(L_j)$ is connected. Denote the outer boundary vertices of $I(L_j)$ by $B = \{b_1, \ldots, b_l\}$, in counterclockwise order. We proceed in two phases. In the first phase, consider a vertex v on the outer boundary of $I(L_j)$. If no such vertex is adjacent to some vertex in L_{j+1}, then the lemma holds. Suppose v is adjacent to $v_1 \in L_{j+1}$. If the edge (v, v_1) is shared by two different faces, we cut (v, v_1) at v. Let the newly created vertex be $v' \in L_{i+1}$. This

operation merges these two faces into one face. We repeat the above operation until it cannot be applied anymore, and let the resulting graph be G_1. Consider a connected component C of $D(L_{j+1})$. If C is connected to B, then there must be an edge (b_x, v_C) such that $b_x \in B$ and $v_C \in C$. Cutting (b_x, v_C) at b_x will disconnect B and C in G_1 because there cannot be a second edge connecting B to C (otherwise, each of the two edges would be adjacent to two different faces).

In the second phase, we perform the following operation on G_1. If B is connected to a connected component C of $D(L_{j+1})$ (C contains some nonempty inner faces) by an edge (b_x, v_C) with $b_x \in B$ and $v_C \in C$, then we cut (b_x, v_C) at b_x. We repeat this operation until it cannot be applied anymore on G_1. This disconnect the boundaries of $I(L_j)$ and $I(L_{j+1})$. In the first phase at most h_j edges are cut, and in the second phase at most h_{j+1} edges (at most one edge for each connected component C of $D(L_{j+1})$ with a nonempty inner face). □

Let q, $0 \leq q < k$, be a constant integer to be chosen later. We disconnect the boundary of $I(L_q)$ from the boundary of $I(L_{q+1})$, the boundary of $I(L_{k+q})$ from the boundary of $I(L_{k+q+1})$, the boundary of $I(L_{2k+q})$ from the boundary of $I(L_{2k+q+1})$, ..., by cutting in total at most $\sum_i (h_{ik+q} + h_{ik+q+1})$ edges, by Lemma 9. Choose q to be the integer that minimizes $H_q = \sum_i (h_{ik+q} + h_{ik+q+1})$. Since $\sum_{i=0}^{k-1} H_i = \sum_{i=1}^{k-1} \sum_j (h_{jk+i} + h_{jk+i+1}) \leq 2 \sum_i h_i = 2h$, we have $H_q \leq \frac{2h}{k}$. Moreover, it is easy to see that the resulting graph is a series of $(k+1)$-outerplanar graphs. Using the techniques developed at the end of Section 4, we compute in polynomial time an optimal solution for each of these $(k+1)$-outerplanar graphs. The sum of the retractions for all $(k+1)$-outerplanar graphs is at most $OPT + H_q \leq OPT + \frac{2h}{k}$ by Lemma 8(1). By Lemma 8(2) we know how to convert these solutions into a final solution with at most $OPT + \frac{4h}{k}$ retractions. Choosing a constant integer value of $k = \lceil \frac{4}{\epsilon} \rceil$, we obtain the following result.

Theorem 10. *We can compute in polynomial time a machining path with at most $OPT + \epsilon h$ retractions, for any constant $\epsilon > 0$.* □

References

1. E. M. Arkin, M. A. Bender, E. D. Demaine, S. P. Fekete, J. S. B. Mitchell, and S. Sethia. Optimal Covering Tours with Turn Costs. *SIAM Journal on Computing*, 35(3):531–566, 2005.
2. E. M. Arkin, S. P. Fekete, and J. S. B. Mitchell. Approximation Algorithms for Lawn Mowing and Milling. *Computational Geometry: Theory and Applications*, 17:25–50, 2000.
3. E. M. Arkin, M. Held, and C. L. Smith. Optimization Problems Related to Zigzag Pocket Machining. *Algorithmica*, 26(2):197–236, 2000.
4. S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of Finding Embeddings in a k-tree. *SIAM Journal on Algebraic Discrete Methods*, 8:277–284, 1987.
5. S. Arnborg and A. Proskurowski. Linear Time Algorithms for NP-hard Problems Restricted for Partial k-trees. *Discrete Applied Mathematics*, 23:11–24, 1989.
6. S. Arya, S.-W. Cheng, and D. M. Mount. Approximation Algorithm for Multiple-tool Milling. *International Journal of Computational Geometry and Applications*, 11(3):339–372, 2001.

7. B. Baker. Approximation Algorithm for NP-complete Problems on Planar Graphs. *Journal of the ACM*, 41(1):153–180, 1994.
8. H. L. Bodlaender. Some Classes of Graphs with Bounded Tree-width. *Bulletin of the EATCS*, 36:116–126, 1988.
9. L. K. Bruckner. Geometric Algorithms for 2.5D Roughing Process of Sculptured Surfaces. *Proc. Joint Anglo-Hungarian Seminar on Computer-Aided Geometric Design*, Budapest, Hungary, Oct. 1982.
10. R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer-Verlag, New York, 1999.
11. M. K. Guyder. Automating the Optimization of 2.5 Axis Milling. *Proc. Computer Applications in Production and Engineering*, North-Holland, Oct. 1989.
12. M. Held. A Geometry-Based Investigation of the Tool Path Generation for Zigzag Pocket Machining. *Visual Computers*, 7(5-6):296–308, 1991.
13. M. Held. *On the Computational Geometry of Pocket Machining*. Lecture Notes in Computer Science, Vol. 500, Berlin, Springer-Verlag, 1991.
14. M. Held and E. M. Arkin. Letter to Editor: An Algorithm for Reducing Tool Retractions in Zigzag Pocket Machining. *Computer-Aided Design*, 32(10):917–919, 2000.
15. B. K. Kim, J. Y. Park, H. C. Lee, and D. S. Kim. Determination of Cutting Direction for Minimization of Tool Retraction Length in Zigzag Pocket Machining. *Proc. International Conf. on Computational Science and Its Applications*, Lecture Notes in Computer Science, Vol. 2669, pp. 680–689, 2003.
16. T. R. Kramer. Pocket Milling with Tool Engagement Detection. *J. Manufacturing Systems*, 11(2):114–123, 1992.
17. D. Lichtenstein. Planar Satisfiability and Its Uses. *SIAM Journal on Computing*, 11:329–343, 1982.
18. B. Reed. Finding Approximate Separators and Computing Tree-width Quickly. *Proc. 24th Annual ACM Symp. on Theory of Computing*, pp. 221–228, 1992.
19. J. Schwerdt, M. Smid, and R. Janardan. Computing an optimal hatching direction in layered manufacturing. *International Journal of Computer Mathematics*, 79:1067–1081, 2002.
20. Y. S. Suh and K. Lee. Neural Network Modeling for Tool Path Planing of the Rough Cut in Complex Pocket Milling. *J. Manufacturing Systems*, 15(5):273–284, 1990.
21. K. Tang. Geometric Optimization Algorithms in Manufacturing. *Computer-Aided Design and Applications*, 2(6):747–758, 2005.
22. K. Tang, S.-Y. Chou, and L.-L. Chen. An Algorithm for Reducing Tool Retractions in Zigzag Pocket Machining. *Computer-Aided Design*, 30(2):123–129, 1998.
23. K. Tang and A. Joneja. Traversing the Machining Graph of a Pocket. *Computer-Aided Design*, 35(11):1023–1040, 2003.
24. Z.-Y. Yao and S. K. Gupta. Cutter Path Generation for 2.5D Milling by Combining Multiple Different Cutter Path Patterns. *Int. J. Prod. Res.*, 42(11):2141–2161, 2001.

Efficient Computation of Nash Equilibria for Very Sparse Win-Lose Bimatrix Games

Bruno Codenotti[1], Mauro Leoncini[2], and Giovanni Resta[1]

[1] IIT-CNR, Via Moruzzi 1, Pisa (Italy)
{b.codenotti, g.resta}@iit.cnr.it
[2] Dipartimento di Ingegneria dell'Informazione
Università di Modena e Reggio Emilia, Modena (Italy)
leoncini@unimo.it

Abstract. It is known that finding a Nash equilibrium for win-lose bimatrix games, i.e., two-player games where the players' payoffs are zero and one, is complete for the class $PPAD$.

We describe a linear time algorithm which computes a Nash equilibrium for win-lose bimatrix games where the number of winning positions per strategy of each of the players is at most two.

The algorithm acts on the directed graph that represents the zero-one pattern of the payoff matrices describing the game. It is based upon the efficient detection of certain subgraphs which enable us to determine the support of a Nash equilibrium.

1 Introduction

In 1951 Nash proved that any n-player game has an equilibrium in the mixed strategies [10]. The proof was based on a fixed point argument, and left open the associated computational question of finding such an equilibrium. In 1964, Lemke and Howson introduced an algorithm for the computation of a Nash equilibrium in 2-player games. In the worst case, this algorithm has an exponential running time [12]. It provides us with another, different from Nash's, proof of the existence of an equilibrium.

In 1994 Papadimitriou introduced a complexity class, $PPAD$, which captures a wealth of equilibrium problems, e.g., the market equilibrium problem as well as Nash equilibria for n-player games [11]. Problems complete for this class include a (suitably defined) computational version of the Brouwer Fixed Point Theorem.

In 2005 a flurry of results appeared, where first the $PPAD$-completeness of 4-player games [6], then of 3-player games [2, 7], and finally of 2-player games [3] were proven. In particular, the latter hardness result by Chen and Deng came as a sort of surprise, since the 2-player case was conjectured to be computationally tractable. Combined with a result by Abbott, Kane, and Valiant [1], it also implies the $PPAD$-completeness of win-lose bimatrix games, i.e., 2-player games where the players' payoffs are zero and one.

Therefore it seems unlikely that polynomial time algorithms exist for win-lose bimatrix games. This fact makes it worthwhile to analyze restricted versions of

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 232–243, 2006.

the problem which might be endowed with computationally useful structural properties.

In this paper, we consider some restricted win-lose games, where the zero-one matrices describing the payoffs of the players have at most two ones per row and column. Following [5], we cast the problem of computing an equilibrium for win-lose games in terms of finding a *good assignment* in a directed graph - see Section 2 below for proper definitions.

The restriction on the zero-one pattern induces very sparse directed graphs. We show how to efficiently detect suitable subgraphs of these sparse graphs, which lead to the discovery of the support of a Nash equilibrium, and to the actual determination of the equilibrium strategies.

This paper is organized as follows. In Section 2 we introduce the game theoretic notions to be used in the paper. After defining the concept of Nash equilibrium for 2-player games in normal form, we show how the computation of a Nash equilibrium for a win-lose 2-player game is equivalent to the computation of a *good assignment* in a directed graph. In Section 3 we informally describe our algorithm, with the help of illustrations and examples. In Section 4 we formally state our main results. Finally in Section 5 we present some conclusions and open questions.

2 Background

We consider 2-player games in *normal form*. These games are described by a pair (A, B) of matrices, whose entries are the *payoffs* of the two players, called row and column player. $A = (a_{ij})$ is the payoff matrix of the row player, and $B = (b_{ij})$ is the payoff matrix of the column player.

The rows (resp. columns) of A and B are indexed by the row (resp. column) player's *pure strategies*.

The entry a_{ij} is the payoff to the row player, when she plays her i-th pure strategy and the opponent plays his j-th pure strategy. Similarly, b_{ij} is the payoff to the column player, when he plays his j-th pure strategy and the opponent plays her i-th pure strategy.

A *mixed strategy* is a probability distribution over the set of pure strategies which indicates how likely it is that each pure strategy is played. More precisely, in a mixed strategy a player associates to her i-th pure strategy a quantity p_i between 0 and 1, such that $\sum_i p_i = 1$, where the sum ranges over all pure strategies.

Let us consider the game (A, B), where A and B are $m \times n$ matrices. In such a game the row player has m pure strategies, while the column player has n pure strategies. Let x (resp. y) be a mixed strategy of the row (resp. column) player. Strategy x is the m-tuple $x = (x_1, x_2, \ldots, x_m)$, where $x_i \geq 0$, and $\sum_{i=1}^{m} x_i = 1$. Similarly, $y = (y_1, y_2, \ldots, y_n)$, where $y_i \geq 0$, and $\sum_{i=1}^{n} y_i = 1$.

When the pair of mixed strategies x and y is played, the entry a_{ij} contributes to the expected payoff of the row player with weight $x_i y_j$. The expected payoff of the row player can be evaluated by adding up all the entries of A weighted

by the corresponding entries of x and y, i.e., $\sum_{ij} x_i y_j a_{ij}$. This can be rewritten as $\sum_i x_i \sum_j a_{ij} y_j$, which can be expressed in matrix terms as[1] $x^T Ay$. Similarly, the expected payoff of the column player is $x^T By$.

Definition 1. *[Nash Equilibrium] A pair of mixed strategies (x, y) is in Nash equilibrium if $x^T Ay \geq x'^T Ay$, for all stochastic m-vectors x', and $x^T By \geq x^T By'$, for all stochastic n-vectors y'.*

We say that x (resp. y) is a Nash equilibrium strategy *for the row (resp. column) player.*

The set of indices such that $x_i > 0$ (resp. $y_i > 0$) is called the support *of the Nash equilibrium strategy x (resp. y).*

The following Lemma shows that it is possible to restrict our attention to bimatrix games where one of the players' payoff matrix is the identity.

Lemma 1 ([5]). *Let A, B be two $m \times n$ matrices with nonnegative entries, where A (resp., B) has at least one nonzero entry in each row (resp., column). Let $C = \begin{pmatrix} 0 & B \\ A^T & 0 \end{pmatrix}$ and let I be the $(m+n) \times (m+n)$ identity matrix. The Nash equilibria of the game (A, B) are in one-to-one correspondence with the Nash equilibrium strategies of the row player in the game (I, C).*

We now consider games of the form (I, C), where C is a square matrix with zero-one entries.

To avoid trivial pure strategy Nash equilibria, we assume that the matrices I and C do not have entries equal to 1 in the same position, i.e., we assume that the entries on the main diagonal of C are all zero.

Following [5], we now define the notion of *good assignment* in a directed graph G, and then show (Lemma 2 below) that it is equivalent to the notion of Nash equilibrium for the game (I, A), where A is the adjacency matrix of G.

Definition 2. *[Good Assignment]*

Let $G = (V, E)$ be a directed graph. Let x be an assignment of nonnegative weights to the vertices of G. We can assume that x is normalized, i. e. $\sum_i x_i = 1$. The income *$i_x(v)$ of a vertex v is the sum of weights of vertices u which point to v, i. e. $i_x(v) = \sum_{u:(u,v)\in E} x_u$. A vertex v is* happy *if it has highest income (i. e. $i_x(v) \geq i_x(u)$ for all $u \in V$). A vertex v is* working *if it has nonzero weight (i. e. $x(v) > 0$). An assignment x is* good *if all the working vertices are happy.*

Lemma 2 ([5]). *Let (I, A) be a bimatrix game, where A is a zero-one matrix with zero entries along the main diagonal. Let $G[A]$ be the digraph with adjacency matrix A. The Nash equilibrium strategies of the row player in (I, A) are in one-to-one correspondence with the good assignments in $G[A]$.*

The proof of Lemma 2 is quite simple. For the sake of self-containment, we now sketch the idea. Consider a good assignment x for $G[A]$. It has the property that

[1] We use the notation x^T to denote the transpose of vector x.

the entries of the row vector $x^T A$ are maximal for indices j such that $x_j > 0$. If needed, we can scale the entries of x so that $\sum_i x_i = 1$. Let k be the size of the support of x, i.e., k is the number of positive entries of x. Consider the vector y such that $y_j = \frac{1}{k}$, if $x_j > 0$, and $y_j = 0$, if $x_j = 0$. It is easy to check that the pair (x, y) is a Nash equilibrium for the game (I, A).

Based on Lemmas 1 and 2, we can focus on the computation of a good assignment.

3 An Informal Description of the Algorithm

In this section we provide an informal description of the algorithm for the computation of a good assignment. In the next section, we will then give a more formal presentation, with proper definitions, the pseudocode, the proof of correctness, and the analysis of the running time.

First of all, notice that we can restrict ourselves to finding a good assignment in a strongly connected digraph. In fact, if the graph is not connected, we can find a good assignment in one of its connected components and assign zero weight to all vertices in the other components. If the graph is connected, but not strongly connected, then its vertices can be partitioned into two subsets V' and V'' such that the graph induced by the vertices in V'' is strongly connected, and the arcs joining vertices from V' and vertices from V'' are all directed towards the vertices in V''. Therefore a good assignment for the original graph can be obtained via an extension of a good assignment for V'', obtained by setting to zero the weights of the vertices in V'.

The reduction outlined above can be computed by standard algorithms in time linear in the number of vertices plus the number of edges of the graph.

The algorithm is composed of two main phases. The first one partitions the vertices of the input graph into two subsets, where the vertices of one subset can be assigned weight zero. The input to the second phase of the algorithm is the subgraph induced by the second subset of vertices. The output of this phase is a further smaller set of vertices for which it is easy to assign nonzero weights that form the support of the good assignment for the original graph.

Our approach thus consists of the decomposition of a strongly connected digraph G in two components, the first one with a structure that admits an easy to compute good assignment with positive weights, and the other one composed by vertices whose weights can be set to zero.

This idea is illustrated by the example shown in Fig.1 (left). The vertices belonging to the subgraph G' (a cycle) are assigned weight 1, while all the other vertices are assigned weight 0. All vertices with weight 1 have income 1 and all the vertices with weight 0 have income *at most* 1.

A major obstacle in the search for such a decomposition is the potential presence of two arcs exiting from G' and pointing to the same vertex (w in Fig.1(right)) not in G'. Indeed, if we add the arc (u, w) to G, then, unless we change the weights of the nodes in G', the node w will have income 2, which prevents the current assignment from being a good assignment.

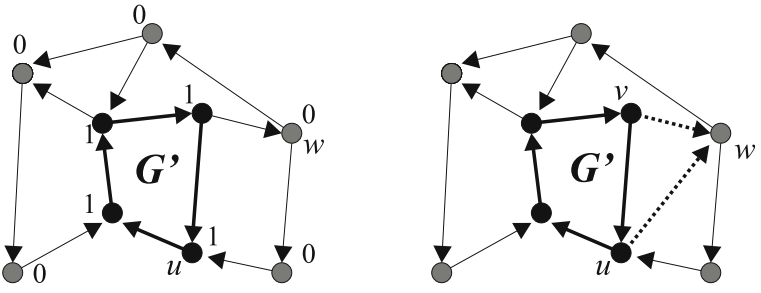


Fig. 1. On the left a valid solution, obtained by equating to one the weights of the vertices belonging to the subgraph G'. On the right, the addition of the edge (u, w) creates a "bump", disrupting the previous solution.

We will call a *"bump"* a configuration given by two arcs exiting a subgraph G' and pointing to the same vertex, not belonging to G'.

The goal of the first phase of our algorithm will be that of finding a subgraph G' of G with two desirable properties:

- G' can be represented as a directed cycle C, containing all the vertices of G', plus a number of *chords*, i.e., arcs of G' not in C.
- G' has no bumps in G, i.e., there are no vertices $u, v \in G'$ and $w \in G \backslash G'$ such that both edges (u, w) and (u, w) exist in G.

Note that, contrary to what was the case in the example of Fig.1, a subgraph G' satisfying the two properties above does not yet allow us to determine a trivial assignment of weights, because of the presence of chords. In the second phase of the algorithm we will take care of the chords, by finding a suitable subgraph of G' with additional properties.

We now show how to detect such a subgraph G'. Given a graph G, in order to obtain a cycle $C \subseteq G$ we can start from an arbitrary vertex u in G, and then just follow one of its outgoing arcs (since G is strongly connected, there is at least one such arc). Repeating this procedure we will eventually reach an already visited vertex, thus obtaining a cycle.

However there is no guarantee that such a cycle C will be bump-free, i.e., it might contain two vertices u and v, and a third vertex $w \in G \backslash C$, where the edges $u \rightarrow w \leftarrow v$ belong to G. The easiest way to make sure that such a bump does not occur during the construction of the cycle C is to follow the arc $u \rightarrow w$ if we are visiting u, and the arc $v \rightarrow w$ if we are visiting v. Either way w will be included in the final cycle C if any of u and w is included in C, thus preventing the adjacency configuration $u \rightarrow w \leftarrow v$ to be a bump for C.

However, we can end up getting a configuration like the one shown in Fig. 2, where, if we first follow the arc $v \rightarrow x$ (to avoid the potential bump with apex x), and later (to avoid the potential bump with apex y') we follow $u' \rightarrow y'$, then, if in the construction of the cycle we reach vertex z, then we are presented with

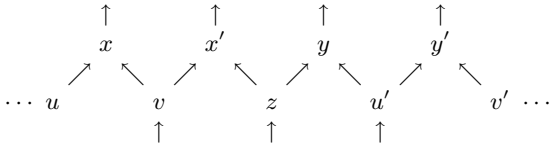


Fig. 2. A configuration preventing the detection of a bump-free cycle

a dilemma with no solution: if we follow $z \to x'$ (resp. $z \to y$) then we generate a cycle with the bump $\{z, u', y\}$ (resp. $\{z, v, x'\}$) (see Fig. 3).

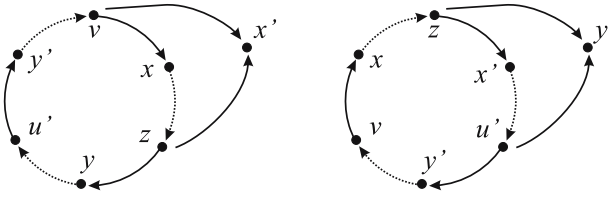


Fig. 3. The construction of the cycle introduces either the bump on the left or the one on the right

To overcome this problem, we use a labeling of the arcs of G, based on G's *alternating decomposition*. This labeling will force us to follow a *bump-safe* path during the construction of the cycle C.

An alternating decomposition of G is a partition of the edges of G in alternating paths, of the form $u_1 \leftarrow u_2 \rightarrow u_3 \leftarrow u_4 \cdots$ or $u_1 \rightarrow u_2 \leftarrow u_3 \rightarrow u_4 \cdots$, and alternating cycles, which are closed alternating paths with an even number of edges, like $u_1 \leftarrow u_2 \rightarrow u_3 \leftarrow u_4 \rightarrow u_1$. We restrict ourself to *maximal* decompositions, in which no path can be further extended.

It is very easy to see that, given the limitation on the in-degree and out-degree of G, such a maximal decomposition is unique and can be obtained in a straightforward manner. Figure 4 presents an example of decomposition. Note that the alternating paths do not need to be simple: a vertex can appear twice in the same path. Figure 5 lists the alternating paths for the graph of Figure 4.

Once we have obtained an alternating decomposition, it is quite intuitive how to avoid the obstacle illustrated in Figs. 2 and 3. For each alternating path or cycle, we consider only the vertices with two outgoing edges. For instance, in the first alternating path of Figure 5, we consider vertices $\{1, 3, 5\}$. We then choose a node, say 3, and mark arbitrarily one of its outgoing edges, say $3 \rightarrow 4$. We proceed by marking every other edge in the alternating path, so that each node can be reached by one unmarked edge. An example of such labeling is illustrated in Fig.6. The effect of the labeling is that, if there is a vertex which is the apex of

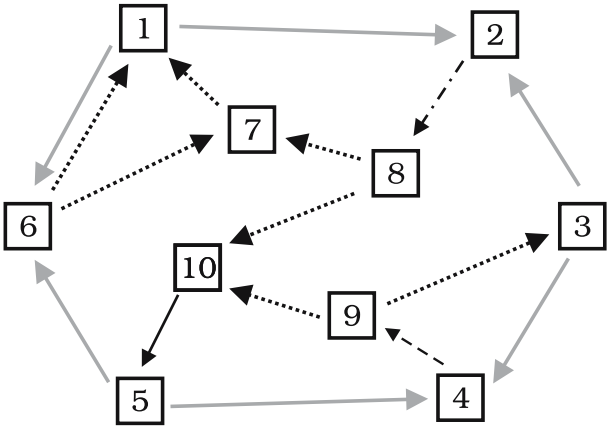


Fig. 4. A digraph G with in- and out-degree at most 2, and its five (arc disjoint) alternating paths whose union includes all the arcs of G

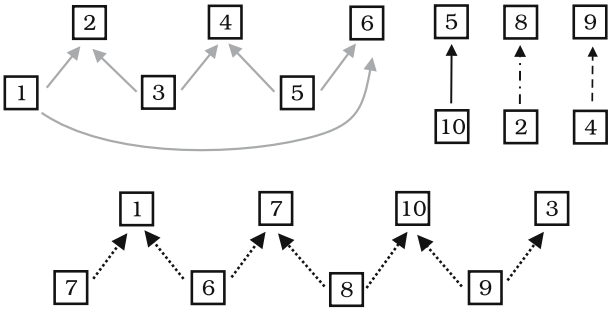


Fig. 5. The alternating decomposition of the graph G of Fig. 4

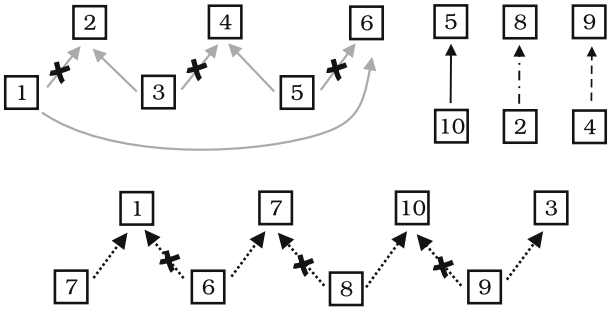


Fig. 6. The labeling process, for the graph in Figure 4, based on the alternating decomposition

a potential bump, say vertex 4, then we can reach it only through an unmarked edge.

We can now return to our cycle construction. We start from an arbitrary vertex u, and we construct a cycle by following unmarked edges until we reach a node we have already visited.

For example, in the case of Figs. 4 and 6, starting from vertex 1, we obtain the cycle $C = \{1, 6, 7\}$, while starting from vertex 8, we obtain $C = \{8, 10, 5, 4, 9, 3, 2\}$.

This procedure always obtains a cycle, since the labeling guarantees that at least an outgoing arc for every vertex is unmarked. It is also easy to see that the cycle is bump-free. Indeed, suppose that the cycle C obtained by the procedure above has a bump $u \to w \leftarrow v$, with $u, v \in C$ and $w \in G \backslash C$. Since w does not belong to C, while u and v do, then both u and v must have two outgoing edges, say $x \leftarrow u \to w \leftarrow v \to y$, and during the construction of C we must have followed edges $u \to x$ and $v \to y$. But this is incompatible with our preliminary alternating labeling since $x \leftarrow u \to w \leftarrow v \to y$ is a fragment of an alternating path or cycle.

The subgraph G' to be fed to the second phase of the algorithm will be the subgraph of G induced by the vertices of the bump-free cycle C. It is easy to see that the fact that G' is bump-free implies that we can assign weight 0 to the nodes in $G \backslash G'$ and deal with the weight assignment for G', independently of the rest of the graph.

In the second phase we will find a subgraph G'' of G' such that

- G'' is induced by a cycle;
- G'' is bump-free with respect to G';
- the maximal alternating decomposition of G'' only consists of alternating cycles and alternating paths of length 1, i.e., single arcs.

These properties imply a "coupling" in G'' between out-degree 2 vertices and in-degree 2 vertices, as shown in Figure 7. What happens is that every in-degree 2 vertex receives its income only from out-degree 2 vertices. It is easy to see that a good assignment can thus be obtained giving weight 1/2 to vertices with out-degree 2, weight 1 to the other vertices in G'' and weight 0 to all the vertices in $G \backslash G''$.

In order to identify a subgraph G'' with these properties, we use the following strategy.

The algorithm, which acts on the current bump-free cycle C and its chords, is divided into a number of stages. At each stage, the algorithm either stops (if the current cycle C induces a subgraph G'' with the desired properties), or returns a shorter cycle C'.

At every stage the algorithm computes an alternating decomposition of the graph induced by the current cycle C. If C does not have chords, or if all the chords belong to alternating cycles as it is the case in Fig. 7, then C and its chords form a subgraph G'' which satisfies our requirements, and the algorithm ends.

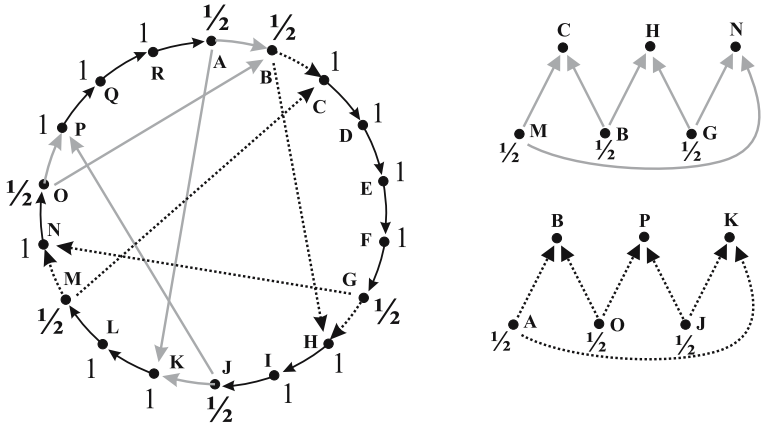


Fig. 7. An example of subgraph obtained after phase 2 and the corresponding weight assignment

Otherwise, we consider any chord (i, j) which is part of an open alternating path (and thus which does not belong to one of the alternating cycles), as in Fig. 8 (top), and process such chord with the goal of shrinking the current cycle without introducing new bumps. If the edge (h, k), where k immediately follows i along the current cycle, is not present, then we can get a shorter cycle C' by following the path from j to i along C and then closing the path along edge (i, j). The resulting new cycle C' will be bump-free in G since the only potential bump for C' has apex k.

On the other hand, if the edge (h, k) is present, as in the example illustrated in Fig. 8 (bottom), we proceed as follows. If the edge (u, v) (see Fig. 8 – bottom) is not present, then we can again return the cycle C' as shown in the Figure, since v was the only potential bump for C'.

If (u, v) exists, then we repeat the procedure above. Since we are moving along an alternating open path, then we will necessarily reach a chord which can be used to shortcut the cycle without creating a bump.

Since at every stage we either stop or shrink the current bump-free cycle, the algorithm will always terminate in a finite number of steps, returning a subgraphs with the desired properties. Recalling the example of Fig. 7 and the related discussion, we can now assign positive weights to the vertices of G'', and obtain a good assignment.

4 Pseudocode and Correctness

For reasons of space, we omit from this extended abstract the formal definitions, the pseudocode of the algorithm sketched in Section 3, the proof of its correctness, and the analysis of its running time. The reader can find all the details in [4].

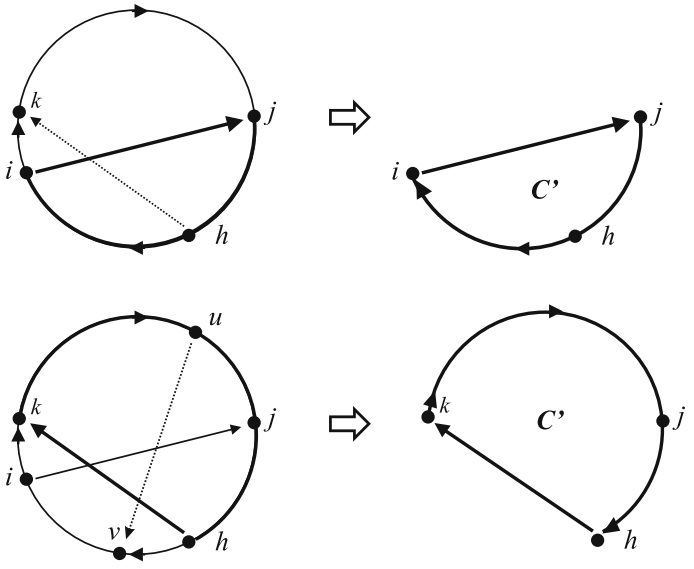


Fig. 8. Phase 2: an open alternating path induces a reduction of the current cycle

Here we only state the main results.

Definition 3. *[Alternating path] Let $G = (V, A)$ be a directed graph. We say that a sequence of vertices $\langle u_1, u_2, \ldots, u_k \rangle$ defines an* alternating path *if and only if*

$$A \supseteq \{(u_1, u_2), (u_3, u_2), (u_3, u_4), (u_5, u_4), \ldots\}$$

or

$$A \supseteq \{(u_2, u_1), (u_2, u_3), (u_4, u_3), (u_4, u_5), \ldots\}.$$

We say that an alternating path is *maximal* if it cannot be extended.

The paths through a vertex v can be easily computed starting from v and extending the current path in both directions until no further extension is possible.

Proposition 1. *Let $G = (V, A)$ be a strongly connected digraph with in- and out-degree at most 2. Then A can be be uniquely described as the disjoint union A_G of the maximal alternating paths through the vertices of G.*

The partition A_G of Proposition 1 is called the *alternating decomposition* of G and can be easily computed in linear time. Starting from the empty set, we repeatedly pick a node $v \in G$ not yet in A_G, and add to A_G the (at most) two maximal paths through v until no further node remains. Figure 5 shows an example of an alternating decomposition.

Proposition 2. *Every strongly connected digraph $G = (V, A)$ with in- and out-degree at most 2 contains a bump free cycle C.*

The results of this section can be summarized as follows.

Theorem 1. *Let G be a digraph on n vertices, with in- and out-degree at most two. A good assignment to G can be computed in $O(n)$ time. Moreover, the weights can be chosen from the set $\{0, 1/2, 1\}$.*

Building upon the Lemmas of Section 2, Theorem 1 immediately leads to Corollary 1 below.

Corollary 1. *Let A and B be $m \times n$ zero-one matrices with at most two nonzero entries per row and column. A Nash equilibrium for the bimatrix game (A, B) can be computed in $O(n+m)$ time. Moreover, the entries of the Nash equilibrium strategies can be chosen from the set $\{0, 1/2, 1\}$.*

5 Conclusions

The problem of computing a Nash equilibrium for 2-player win-lose games is complete for the class $PPAD$, and thus unlikely to be solvable in polynomial time. The core of the computational difficulty is in finding the support of a Nash equilibrium. In fact, once the support is known, the problem simplifies to linear programming.

In this paper we have dealt with a restriction under which the determination of the support translates into an interesting problem on certain very sparse directed graphs. By taking advantage of the structural properties that arise from the sparseness, we have been able to devise an efficient algorithm which determines the support, and then the actual weights.

Future work includes further exploring and possibly extending the frontier of tractability of the problem, e.g., by mitigating the sparsity assumptions.

References

1. T. Abbott, D. Kane, P. Valiant, On the Complexity of Two-Player Win-Lose Games. Proc. 46th Annual IEEE Symposium on Foundations of Computer Science, pp. 113-122 (2005).
2. X. Chen, X. Deng, 3-NASH is PPAD-Complete, ECCC TR05-134 (2005).
3. X. Chen, X. Deng, Settling the Complexity of 2-Player Nash-Equilibrium, ECCC TR05-140 (2005).
4. B. Codenotti, M. Leoncini, G. Resta, Efficient Computation of Nash Equilibria for Very Sparse Win-Lose Games, ECCC Technical Report TR06-012. Available at http://eccc.hpi-web.de/eccc-reports/2006/TR06-012/index.html.
5. B. Codenotti, D. Stefankovic, On the computational complexity of Nash equilibria for (0,1)-bimatrix games. Information Processing Letters, 94(3), pp.145-150 (2005).
6. C. Daskalakis, P. Goldberg, C. Papadimitriou, The complexity of computing a Nash equilibrium, ECCC TR05-115 (2005).
7. C. Daskalakis, C. Papadimitriou, Three-Player Games Are Hard, ECCC TR05-139 (2005).

8. P. W. Goldberg, C. Papadimitriou, Reducibility Among Equilibrium Problems, ECCC TR05-090 (2005).
9. C.E. Lemke and J.T. Howson, Equilibrium points in bimatrix games, Journal of the Society for Industrial and Applied Mathematics 12, pp. 413- 423 (1964).
10. J. Nash, Non-Cooperative Games, Annals of Mathematics 54(2), pp. 286-295 (1951).
11. C. 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).
12. R. Savani and B. von Stengel, Exponentially Many Steps for Finding a Nash Equilibrium in a Bimatrix Game. Proc. 45th Annual IEEE Symposium on Foundations of Computer Science, pp. 258-267 (2004).

Distributed Almost Exact Approximations for Minor-Closed Families

Andrzej Czygrinow[1] and Michał Hańćkowiak[2]

[1] Department of Mathematics and Statistics
Arizona State University
Tempe, AZ 85287-1804, USA
andrzej@math.la.asu.edu
[2] Faculty of Mathematics and Computer Science
Adam Mickiewicz University
Poznań, Poland
mhanckow@amu.edu.pl

Abstract. We give efficient deterministic distributed algorithms which given a graph G from a proper minor-closed family $\mathcal{C}$ find an approximation of a minimum dominating set in G and a minimum connected dominating set in G. The algorithms are deterministic and run in a polylogarithmic number of rounds. The approximation accomplished differs from an optimal by a multiplicative factor of $(1 + o(1))$.

1 Introduction

The most fundamental challenge in theory of distributed algorithms is to determine how the local structure of a network impacts its global properties. This leads to a completely different computational paradigm than the sequential model or the massively parallel PRAM model. Not surprisingly, many problems which admit efficient sequential protocols, such as the maximum matching problem or the maximal independent set problem elude efficient distributed solutions. In this paper, we will study distributed approximations for two classical graph-theoretic problems assuming the underlying graph belongs to a proper minor-closed family. We will consider the distributed model which was introduced by Linial in [L92]. In this model, the network is represented by an undirected graph with vertices corresponding to processors, and edges corresponding to communication links between processors. The network is synchronized and computations proceed in discrete rounds. In a single round a vertex can send and receive messages from its neighbors, and can perform some local computations. Neither the amount of local computations nor the lengths of messages is restricted in any way. Importantly, we will also assume that nodes in the network have unique identifiers which are positive integers from $\{1, \ldots, n\}$ where $n = poly(|G|)$ is globally known and $|G|$ is the order of the graph.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 244–255, 2006.

1.1 Results

Although different possible measures of efficiency of a distributed algorithm can be considered, traditionally a deterministic distributed algorithm is called *efficient* in the model if it runs in a poly-logarithmic (in the order of the graph) number of rounds. Only very few classical graph-theoretic problems are known to admit an efficient deterministic distributed algorithm. For example, even the maximal independent set problem, for which an efficient deterministic PRAM algorithm exists [L86], still has an unknown distributed complexity. In this paper, we shall focus on distributed approximation algorithms for two classical problems, the minimum dominating set problem and the minimum connected dominating set problem. Let β be a graph-theoretic function to be optimized and let β^* denote its optimal value. An *almost exact approximation* for the optimization problem is a distributed approximation algorithm which given a positive integer k, finds in a graph G in a poly-logarithmic number of rounds a solution with value of at least $(1 - O(1/\ln^k |G|))\beta^*(G)$, where $|G|$ is the order of G. For example, Kuhn et. al. in [KMNW05b] give almost-exact approximations for the maximum independent set and minimum dominating set problems in unit-disk graphs.

In this paper we will give efficient distributed approximation algorithms for the minimum dominating set problem and the minimum connected dominating set problem for graphs which are from a proper minor-closed family. Let $G = (V, E)$ be a graph. Graph H is called a minor of G if for some subgraph G' of G, there is a partition of $V(G')$ into $V_1, \ldots, V_l$, such that the graph $\bar{H}$, with vertex set $\{1, \ldots, l\}$ and edges between i and j whenever there is an edge in G' with one endpoint in V_i, another in V_j, is isomorphic to H. It is well-known (see [D97]) that H is a minor of G if and only if it can be obtained from a subgraph of G be a series of edge contractions. An infinite family of graphs $\mathcal{C}$ is called minor-closed when for every graph $G \in \mathcal{C}$ any minor of G is also in $\mathcal{C}$. A family $\mathcal{C}$ is called proper if there exists a graph which is not in $\mathcal{C}$, i.e. $\mathcal{C}$ is not the family of all graphs. Certainly, the most important example of a proper minor-closed family is the class of planar graphs. For $\mathcal{C}$, let $\rho_{\mathcal{C}}$ be the infimum of the edge density of graphs from $\mathcal{C}$. Complexity of algorithms depends on $\rho_{\mathcal{C}}$ and we will often use the fact that if $\mathcal{C}$ is proper then $\rho_{\mathcal{C}}$ is finite (see [NM05]).

Distributed approximation algorithms for planar graphs were studied in [CH04] and [CHS06]. In [CH04], almost exact approximation are obtained for the maximum-weight independent set problem provided the underlying graph is planar. In [CHS06], an almost exact approximation for the maximum matching problem is given in planar graphs and an almost exact approximation for the minimum dominating set problem is given in planar graphs that do not contain $K_{2,\ln|G|}$ as a subgraph. In this paper we will not only get rid of the annoying additional assumption on planar graphs from [CHS06] but also we will show how to solve the problems in any minor closed family $\mathcal{C}$. Finally, we will prove that the minimum connected dominating set problem can be approached in a very similar way.

A *dominating set* in a graph G is a subset D of vertices such that for every vertex $v \notin D$ a neighbor u of v belongs to D. By $\gamma(G)$ we will denote the cardi-

nality of a smallest dominating set in G. A dominating set D is called a *connected dominating set* in G if in addition, the subgraph of G induced by D is connected. We will denote by $\gamma_c(G)$ the cardinality of the smallest connected dominating set in a connected graph G. For the minimum dominating set problem, we will prove that there is a distributed algorithm which given positive integer q finds in graph $G \in \mathcal{C}$ a dominating set $\bar{D}$ such that $|\bar{D}| \leq \left(1 + O\left(\frac{1}{\ln^q |G|}\right)\right)\gamma(G)$. The algorithm runs in $O(\ln\ln|G| \ln^* |G| \ln^{1+r} |G|)$ rounds where $r = 6(q+1)\rho_{\mathcal{C}} \ln 3$. (Theorem 1.) For the minimum connected dominating set problem we will show that there is a distributed algorithm which finds in a connected graph $G \in \mathcal{C}$ a connected dominating set $\bar{D}$ such that $|\bar{D}| \leq \left(1 + O\left(\frac{1}{\ln^q |G|}\right)\right)\gamma_c(G)$. The algorithm runs in $O(\ln\ln|G| \ln^* |G| \ln^{1+r} |G|)$ rounds with $r = 6(q+1)\rho_{\mathcal{C}} \ln 3$. (Theorem 2.)

1.2 Related Work

We briefly indicate how our contribution compares with other results refering to Elkin's survey [E04], for a more comprehensive overview. First let us mention that efficient distributed algorithms that find an exact solution do not exist for the minimum dominating set problem even when restricted to cycles [L92]. In addition, recently, Kuhn et. al. in [KMW04] showed that the number of rounds required to achieve a poly-logarithmic approximation ratio for minimum dominating set is at least $\Omega(\sqrt{\log|G|/\log\log|G|})$ or $\Omega(\log\Delta/\log\log\Delta)$, where Δ denotes the maximum degree of graph G.

On a more positive note, Kutten and Peleg [KP95] gave an efficient distributed algorithm which finds a dominating set of size at most $|G|/2$ in an arbitrary graph G. Not surprisingly, if randomization is allowed, then fast approximations can be obtained. In particular, a nice algorithm from [KW03] gives a randomized $O(k\Delta^{2/k}\log\Delta)$-approximation in a constant time using an LP relaxation. As in the case of the minimum dominating set problem, efficient randomized algorithms for the connected dominating set are known [DPRS03].

It is also worth mentioning that our algorithms share many similarities with almost-exact approximations for the above problems in unit-disk graphs from [KMNW05b] and particularly [CH06]. Specifically, algorithms for unit-disk graphs and graphs from a minor closed families are both attacked by first finding a cluster graph and then perform computations locally. Clustering from [CH04] (as well as [KMNW05b]) exploits the bounded-growth property of unit-disk graphs and is based on the ruling-set technique from [AGLP89]. The clustering in this paper, generalizes the clustering procedures from [CH04] and [CHS06] and relies on properties of minor-monotone families.

1.3 Notation and Organization

We will use the standard graph-theoretic notation and terminology. In particular, following the convention from [D97], for a graph G, $|G|$ will denote the number of vertices in G and $||G||$ the number of edges. In the rest of the paper we will first give an auxiliary distributed $O(\ln|G|)$-approximation for the minimum

dominating set problem in a graph $G \in \mathcal{C}$ (Section 2). Section 2 also contains a generalization of the clustering from [CH04] to minor-closed families $\mathcal{C}$. In Section 3, we give our approximation algorithms and give a specification to the important case when $\mathcal{C}$ is the class of planar graphs.

2 Tools

Let $\mathcal{C}$ be a proper minor-closed family of graphs. In this section, we will describe two auxiliary algorithms. The first procedure finds a $O(\ln |G|)$-approximation of the minimum dominating set in a graph $G \in \mathcal{C}$. This is a very simple distributed greedy algorithm which will be used as an initial procedure that yields an auxiliary graph which is further clustered by the main algorithm. The second procedure is a modification of the clustering algorithm from [CH04]. This is our main tool for finding a clustering of a graph from $\mathcal{C}$.

2.1 Distributed $O(\ln |G|)$-Approximation

For a proper minor-closed family $\mathcal{C}$ let $\rho_{\mathcal{C}}$ be the edge density of $\mathcal{C}$, i.e. $\rho_{\mathcal{C}}$ is the infimum of ρ such that for every graph $G \in \mathcal{C}$, $||G|| \leq \rho|G|$. Then $\rho_{\mathcal{C}}$ is finite as long as $\mathcal{C}$ is proper (see [NM05]) and if G is nontrivial (i.e. contains a nonempty graph) then $\rho_{\mathcal{C}} \geq 0.5$. Let $G \in \mathcal{C}$ and suppose that V_1, V_2 is a partition of V. Let

$$deg_i(v) = |N(v) \cap V_i|, \Delta_i = \max_v deg_i(v)$$

where $N(v)$ is a set of neighbours of v. In addition for $S \subset V$ let $N_i(S)$ denote the set of vertices in V_i which have a neighbor in S.

Lemma 1. *Let $\mathcal{C}$ be a proper nontrivial minor closed family. Let G be a nonempty graph from $\mathcal{C}$, let V_1, V_2 be a partition of $V(G)$ and let $B = \{v|deg_1(v) \geq \Delta_1/2\}$. If $\Delta_1 \geq 4\rho_{\mathcal{C}}$ and D is a subset of V which dominates all vertices from V_1 then*

$$|D| \geq \frac{|B|}{6\rho_{\mathcal{C}}(2\rho_{\mathcal{C}} + 1)}.$$

We will consider the following greedy algorithm.

GreedyDS
Input: Graph $G = (V, E)$ from $\mathcal{C}$.
Output: Dominating set D^* in G.

(1) $D^* := \emptyset$, $V_1 := V$, $V_2 := \emptyset$.
(2) for $i := 0$ to $\lfloor \lg |G| \rfloor - \lceil \lg 4\rho_{\mathcal{C}} \rceil - 1$ do
  (a) Let $B := \{v|deg_1(v) \geq |G|/2^{i+1}\}$.
  (b) If $v \in V_1$ and $N(v) \cap B \neq \emptyset$ then move v from V_1 to V_2.
  (c) $D^* := D^* \cup B$. Delete all vertices in B and all edges incident to B from G.
(3) Let $D^* := D^* \cup V_1$. Return D^*.

We shall first make a few preliminary observations about GreedyDS. Let $G^{(i)}$ be the graph in the ith iteration of the for loop. Similarly let $V_k^{(i)}$ ($B^{(i)}$), be the set V_k (B) in the ith iteration and let $\Delta_1^{(i)} = \Delta_1(G^{(i)})$. We first observe the following easy lemma.

Lemma 2. *Let $\mathcal{C}$ be a proper nontrivial minor closed family and let $G \in \mathcal{C}$.*

- *We have $\Delta_1^{(i)} \leq |G|/2^i$.*
- *If $B^{(i)} \neq \emptyset$ then $\Delta_1^{(i)} \geq 4\rho_{\mathcal{C}}$.*

We can now prove the main property of GreedyDS.

Lemma 3. *Let $\mathcal{C}$ be a proper nontrivial minor closed family. Let D be a dominating set in graph $G = (V, E)$ from $\mathcal{C}$. Then*

$$|B^{(i)}| \leq 6\rho_{\mathcal{C}}(2\rho_{\mathcal{C}} + 1)|D|.$$

In addition, if V_1^* denotes the set of vertices in V_1 in the step (3) of GreedyDS *then*

$$|V_1^*| \leq (4\rho_{\mathcal{C}} + 2)|D|.$$

Proof. Let $B^{(<i)} := B^{(0)} \cup \cdots \cup B^{(i-1)}$. Vertices from $V_1^{(i)}$ cannot be dominated by vertices from $B^{(<i)}$ as all neighbors of $B^{(<i)}$ in G are contained in $B^{(<i)} \cup V_2^{(i)}$. Consequently $D \cap (V_1^{(i)} \cup V_2^{(i)})$ is a set which dominates $V_1^{(i)}$ in $G^{(i)}$. By Lemma 2, if $B^{(i)} \neq \emptyset$ then $\Delta_1(G^{(i)}) \geq 4\rho_{\mathcal{C}}$ and we have $B^{(i)} \subseteq \{v | deg_1(v) \geq \Delta_1^{(i)}/2\}$. As $G^{(i)}$ is a subgraph of G, Lemma 1 implies that

$$|B^{(i)}| \leq 6\rho_{\mathcal{C}}(2\rho_{\mathcal{C}} + 1)|D \cap (V_1^{(i)} \cup V_2^{(i)})| \leq 6\rho_{\mathcal{C}}(2\rho_{\mathcal{C}} + 1)|D|.$$

To prove the second part, note that after the iterations from step (2), the maximum degree $\Delta_1 \leq 4\rho_{\mathcal{C}} + 2$. As a result, to dominate all vertices from V_1^* at least $|V_1^*|/(4\rho_{\mathcal{C}} + 2)$ vertices are needed.

Lemma 4. *Let $\mathcal{C}$ be a minor closed family with $\rho_{\mathcal{C}} > 0$ and let $G \in \mathcal{C}$.* GreedyDS *finds a dominating set D^* with*

$$|D^*| = O(\ln |G| \gamma(G)),$$

where $\gamma(G)$ is the size of the minimum dominating set in G.

Proof. D^* is a dominating set as in step (3) all of the remaining vertices from V_1 are added to D^*. There are less than $\lg |G| = \Theta(\ln |G|)$ iteration of step (2) and so, by Lemma 3, $|D^*| = O(\ln |G| \gamma(G))$.

2.2 Clustering Algorithm

We will modify the clustering method from [CHS06] (see also [CH04]) which was applied there to planar graphs. The basic idea of the method is to find

appropriate subgraphs of a graph and contract them. The process is repeated $O(\ln \ln |G|)$ times and the vertices of the graph obtained from contractions in all of the previous iterations give clusters of G. To find appropriate subgraphs it is neccessary to consider weights on edges. We shall start with the following basic observation.

Lemma 5. *Let $\mathcal{C}$ be a proper minor closed family. Let $G = (V, E)$ be a graph from $\mathcal{C}$ and let $A = \{v | deg(v) \leq 3\rho_{\mathcal{C}}\}$. Then*

$$|A| \geq |G|/3.$$

As mentioned before, we will assumme that vertices have unique identifiers which are positive integers. For $v \in V(G)$ the identifier of v will be denoted by $ID(v)$. Note that if $ID(v) \leq n$ for every vertex from $V(G)$ then $|G| \leq n$.

DECOMPOSITION
Input: $G \in \mathcal{C}$, number n such that $ID(v) \leq n$ for $v \in V(G)$.
Output: Partition $V_1, \ldots, V_{\log_k n}$ of G with $k = O(1)$.

1. Let $U := V(G)$, $i := 1$ and $k := (9\rho_{\mathcal{C}} + 3)/(9\rho_{\mathcal{C}} + 2)$.
2. Iterate $\log_k n + 1$ times:
   (a) Let A be the set of vertices in $G[U]$ of degree at most $3\rho_{\mathcal{C}}$.
   (b) Use the Cole-Vishkin algorithm from [CV86] to find a maximal independent set I in the subgraph of $G[U]$ induced by A.
   (c) $V_i := I$, $i := i + 1$, $U := U \setminus I$.

Lemma 6. *[CH04] Let $G = (V, E)$ be a graph from $\mathcal{C}$ such that the identifiers of V are in $\{1, \ldots, n\}$. Then the procedure* DECOMPOSITION *finds a partition $V_1, \ldots, V_{\log_k n}$ of $V(G)$ such that each V_i is an independent set and for every $v \in V_i$, $deg(v, \bigcup_{j>i} V_j) \leq 3\rho_{\mathcal{C}}$. The algorithm runs in $O(\ln^* n \ln n)$ rounds.*

We will now describe our clustering algorithm. This is essentially the algorithm from [CH04] which is here adopted to minor-closed families. Main idea of the algorithm is to find appropriate subgraphs of G and contract the subgraphs so that the number of contracted edges is a constant fraction of $||G||$. The process is iterated $O(\ln \ln n)$ times where $|G| \leq n$. We will identify graphs with their edge sets and if ω is a weight function defined on the edge set of graph H then for $F \subseteq E(H)$, $\omega(F) := \sum_{e \in F} \omega(e)$. In addition, $N(w)$ will denote the set of neighbors of vertex w.

CLUSTERING
Input: Graph $G = (V, E) \in \mathcal{C}$, number n such that $ID(v) \leq n$ for every $v \in V$, positive integer c.
Output: Partition of V.

1. $H := G$ and let $\omega(e) := 1$ for every $e \in H$. Let $l := 6c\rho_{\mathcal{C}} \ln \ln n$.
2. Iterate l times:

(a) Call DECOMPOSITION to find a partition $W_1, \ldots, W_K$ of H with $K = O(\ln n)$. Set $W_{K+1} := \emptyset$ and let $Z_i := \bigcup_{j>i} W_j$.
(b) For every vertex w:
(c) If i is such that $w \in W_i$ and $N(w) \cap Z_i \neq \emptyset$ then:
  – Let $u(w)$ be a vertex in $N(w) \cap Z_i$ such that

$$\omega(\{w, u(w)\}) := \max_{v \in N(w) \cap Z_i} \omega(\{w, v\}).$$

  – Add $\{w, u(w)\}$ to the auxiliary graph F.
(d) Each connected component of F is a tree of diameter $O(K) = O(\ln n)$. For each tree T in F, in parallel, find a set of disjoint stars $S_1 \ldots S_k$ in T such that $\omega(S_1 \cup \cdots \cup S_k) \geq \omega(T - (S_1 \cup \cdots \cup S_k))$.
(e) Modify H as follows:
  – Contract each star S_i to a new vertex $x(S_i)$.
  – For every vertex $x(S_i)$ and $y \in V(H) \cap N(S_i)$ set the weight of

$$\omega(\{x(S_i), y\}) := \sum_{u \in V(S_i) \cap N(y)} \omega(\{u, y\})$$

and set $V(H) := \bigcup\{x(S_i)\} \cup (V(H) - \bigcup V(S_i))$.

3. If $V(H) = \{v_1, ..., v_L\}$ then for each v_i let V_i be the set of vertices of G contracted to v_i in all of the above iterations. Return $V_1, \ldots, V_L$.

Note that graph H obtained in each iteration of CLUSTERING belongs to $\mathcal{C}$ and so its edge density is at most $\rho_{\mathcal{C}}$. We can summarize the performance of CLUSTERING as follows.

Lemma 7. *Let $V_1, \ldots, V_L$ be the clusters in G obtained from* CLUSTERING. *Then*

1. *For every i, $G[V_i]$ is a subgraph of diameter $O(\ln^d n)$, where*

$$d = 6c\rho_{\mathcal{C}} \ln 3.$$

2. *The number of edges connecting different clusters is $O(||G||/\ln^c n)$.*
3. CLUSTERING *runs in $O(\ln \ln n \ln^* n \ln^{1+d} n)$ rounds.*

3 Domination Problems

Let $\mathcal{C}$ be a proper minor-closed family of graphs. In this section, we will give almost exact approximations for the minimum dominating set problem and for the connected minimum dominating set problem in graphs G such that $G \in \mathcal{C}$. Instead of finding a clustering directly in graph G, it is very convenient to work in an auxiliary graph that arises from the $O(\ln n)$-approximation of the dominating set and perform the clustering in this graph. By virtue of the minor-closed property of $\mathcal{C}$, the auxiliary graph will also be a member of $\mathcal{C}$.

3.1 Minimum Dominating Set

We will start with an almost exact approximation for the minimum dominating set problem.

Definition 1. *Let $G = (V, E)$ be a graph and let $D = \{v_1, \ldots, v_l\} \subseteq V$ be a dominating set in G. Then let $\mathbf{A}(D, G)$ be the graph $(\mathbf{V}, \mathbf{E})$ obtained as follows.*

- *Partition $V = V_1 \cup V_2 \cup \cdots \cup V_l$ so that (1) $v_i \in V_i$ and (2) for every $v \in V \setminus D$, $v \in V_i$ if $\{v, v_i\} \in E$ and if $\{v, v_j\} \in E$ for $j \neq i$ then $ID(v_j) > ID(v_i)$.*
- *$\mathbf{V} := \{V_1, \ldots, V_l\}$ (contract V_i to a vertex) and $\{V_i, V_j\} \in \mathbf{E}$ $(i \neq j)$ if there is an edge in G between a vertex from V_i and a vertex from V_j.*

In addition, we will call v_i the *center* of V_i. Let $\mathbf{U}$ be a subset of $\mathbf{V}$ then $\mathbf{U}$ corresponds in a natural way to subset U of $V(G)$ by

$$U := \bigcup_{W \in \mathbf{U}} W.$$

If $\mathbf{U}_1, \mathbf{U}_2, \ldots, \mathbf{U}_l$ is a partition of $\mathbf{V}$ then the corresponding sequence $U_1, U_2, \ldots, U_l$, with $U_i := \bigcup_{W \in \mathbf{U}_i} W$, is a partition of $V(G)$. We will then say that $U_1, U_2, \ldots, U_l$ *arises* from $\mathbf{U}_1, \mathbf{U}_2, \cdots, \mathbf{U}_l$. Finally, for a graph $H = (X, F)$ if $Y \subseteq X$ then $bd(Y)$ will denote the set of all vertices in Y which have a neighbor in $X \setminus Y$.

For a dominating set D obtained by GreedyDS in G let $\mathbf{A} := \mathbf{A}(D, G)$. From Lemma 4

$$|\mathbf{A}| = O(\gamma(G) \ln |G|). \tag{1}$$

Since $\mathbf{A}$ is obtained from G be contracting V_i's, $\mathbf{A} \in \mathcal{C}$. In addition, identifiers of vertices from $\mathbf{A}$ are bounded from above by $n = poly(|G|)$.

ApproxDS
Input: Graph $G = (V, E)$ from $\mathcal{C}$ with $ID(v) \leq n$ for any $v \in V(G)$, a positive integer q.
Output: Dominating set $\bar{D}$ in G.

1. Call GreedyDS to find a dominating set D and consider $\mathbf{A} = (\mathbf{V}, \mathbf{E})$.
2. Call Clustering with $c = 1 + q$ in $\mathbf{A}$.
3. Let $\mathbf{U}_1, \ldots, \mathbf{U}_l$ be a partition of $\mathbf{V}$ and let $U_1, \ldots, U_l$ be a partition of $V(G)$ that arises from $\mathbf{U}_1, \ldots, \mathbf{U}_l$.
4. In each U_i in parallel:
   (a) Find locally in U_i a set $D_i \subseteq U_i$ of the smallest size such that D_i dominates $U_i \setminus bd(U_i)$ in G.
   (b) Let C_i be the set of centers of vertices from $bd(\mathbf{U}_i)$.
   (c) Let $\bar{D}_i := D_i \cup C_i$.
5. Return $\bar{D} := \bigcup_{i=1}^{l} \bar{D}_i$.

Theorem 1. *Let $\mathcal{C}$ be a proper minor-closed family of graphs. Algorithm* ApproxDS *finds in a graph $G \in \mathcal{C}$ a dominating set $\bar{D}$ such that*

$$|\bar{D}| \leq \left(1 + O\left(\frac{1}{\ln^q |G|}\right)\right)\gamma(G)$$

in $O(\ln\ln|G| \ln^* |G| \ln^{1+r} |G|)$ rounds where $r = 6(q+1)\rho_{\mathcal{C}} \ln 3$.

Proof. We will first note that set $\bar{D}$ returned in step five of ApproxDS is a dominating set. Indeed let $v \in V(G)$ and suppose that $v \notin \bar{D}$. If for some i, $v \in U_i \setminus bd(U_i)$ then v is dominated by a vertex from D_i. Otherwise for some i, $v \in bd(U_i)$ and so, by definition of $\mathbf{A}$, $v \in W$ for some $W \in bd(\mathbf{U}_i)$. Consequently, v is dominated by the center of W and the center is in C_i. To establish the bound for $|\bar{D}|$ let us first recall that $\mathbf{A} \in \mathcal{C}$ and so $||\mathbf{A}|| = O(|\mathbf{A}|)$. In addition the graph induced by border vertices of $\mathbf{A}$, i.e. $\mathbf{A}\left[\bigcup_i bd(\mathbf{U}_i)\right]$, has denisty $\rho_{\mathcal{C}}$ and so $|\bigcup_i bd(\mathbf{U}_i)| = O\left(||\mathbf{A}\left[\bigcup_i bd(\mathbf{U}_i)\right]||\right)$ and by Lemma 7 part 2 applied with $n = |G|$, $|\bigcup_i bd(\mathbf{U}_i)| = O(||\mathbf{A}||/\ln^{q+1}|G|)$ as $|\mathbf{A}| \leq |G|$. Consequently, as $||\mathbf{A}|| = O(|\mathbf{A}|)$ and (1) holds, $|\bigcup_i bd(\mathbf{U}_i)| = O(\gamma(G)/\ln^q |G|)$. Since $bd(\mathbf{U}_i)$ are pairwise disjoint and $|bd(\mathbf{U}_i)| = |C_i|$, we have

$$\sum_{i=1}^{l} |C_i| = O(\gamma(G)/\ln^q |G|). \tag{2}$$

Let D^* be a dominating set in G with $|D^*| = \gamma(G)$. Then $|D^* \cap U_i| \geq |D_i|$ as every vertex in $U_i \setminus bd(U_i)$ must be dominated by a vertex from $D^* \cap U_i$. Consequently $\gamma(G) = |D^*| = \sum_{i=1}^{l} |D^* \cap U_i| \geq \sum_{i=1}^{l} |D_i|$ and so $|\bar{D}| \leq \sum_{i=1}^{l} |D_i| + \sum_{i=1}^{l} |C_i| \leq \gamma(G) + \sum_{i=1}^{l} |C_i|$ which in view of (2) gives $|\bar{D}| = \left(1 + O\left(\frac{1}{\ln^q |G|}\right)\right)\gamma(G)$. To estimate the running time, note that Clustring runs in $O(\ln\ln|G| \ln^* |G| \ln^{1+r} |G|)$ rounds in $\mathbf{A}$ and every vertex in $\mathbf{A}$ has diameter of at most two in G. In addition for every i, $\mathbf{A}[\mathbf{U}_i]$ has diameter $O(\ln^r |G|)$ by Lemma 7 part 1 and so the diameter of each $G[U_i]$ is also $O(\ln^r |G|)$. Therefore, finding D_i and C_i can be done in $O(\ln^r |G|)$ rounds and the time complexity of ApproxDS is $O(\ln\ln|G| \ln^* |G| \ln^{1+r} |G|)$.

3.2 Minimum Connected Dominating Set

An algorithm for the minimum connected dominating set problem is very similar. In fact the first three steps are identical and only a very small change must be made in steps four and five. First note that the auxiliary graph $\mathbf{A}$ satisfies

$$|\mathbf{A}| = O(\gamma_c(G) \ln |G|) \tag{3}$$

where $\gamma_c(G)$ is the size if the smallest connected dominating set as $\gamma(G) \leq \gamma_c(G) \leq 3\gamma(G)$ in any connected graph G.

ApproxCDS
Input: A connected graph $G = (V, E) \in \mathcal{C}$, a positive integer q.
Output: A connected dominating set $\bar{D}$ in G.

1. Call GreedyDS to find a dominating set D and consider $\mathbf{A} = (\mathbf{V}, \mathbf{E})$.
2. Call Clustering with $c = 1 + q$ in $\mathbf{A}$.
3. Let $\mathbf{U}_1, \ldots, \mathbf{U}_l$ be a partition of $\mathbf{V}$ obtained in step 2. Let $U_1, \ldots, U_l$ be a partition of $V(G)$ that arises from $\mathbf{U}_1, \ldots, \mathbf{U}_l$.
4. In each U_i in parallel:
   (a) Let C_i be the set of centers of vertices from $bd(\mathbf{U}_i)$.
   (b) Find locally in U_i a set $D_i \subseteq U_i$ of the smallest size such that D_i dominates U_i in G, $G[D_i]$ is a connected subgraph of G, and $C_i \subseteq D_i$.
   (c) For every cluster U_j such that there is an edge in $\mathbf{A}$ between $\mathbf{U}_i$ and $\mathbf{U}_j$ find the shortest path P_{ij} between a vertex from D_i and a vertex from D_j and let $P_i := \bigcup V(P_{ij})$ where the union is taken over all of these paths.
   (d) Let $\bar{D}_i := D_i \cup P_i$.
5. Return $\bar{D} := \bigcup_{i=1}^{l} \bar{D}_i$.

The argument is slightly different than the one given for Theorem 1 as this time the main part of the argument is to show that that G contains a connected dominating set D' such that $|D'| \leq (1 + O(1/\ln^q |G|))\gamma_c(G)$, $G[D' \cap U_i]$ is a connected subgraph, $D' \cap U_i$ dominates U_i, and $C_i \subseteq D' \cap U_i$.

Lemma 8. *Let $G \in \mathcal{C}$ be a connected graph. Then G contains a connected dominating set D' such that $|D'| \leq (1+O(1/\ln^q |G|))\gamma_c(G)$ and for every $i = 1, \ldots, l$*

1. *$G[D' \cap U_i]$ is a connected subgraph of G,*
2. *$D' \cap U_i$ dominates U_i,*
3. *$C_i \subseteq D' \cap U_i$.*

Theorem 2. *Let $\mathcal{C}$ be a minor-closed family. Algorithm* ApproxDS *finds in a connected graph $G \in \mathcal{C}$ a connected dominating set $\bar{D}$ such that*

$$|\bar{D}| \leq \left(1 + O\left(\frac{1}{\ln^q |G|}\right)\right)\gamma_c(G)$$

in $O(\ln\ln|G| \ln^* |G| \ln^{1+r} |G|)$ rounds where $r = 6(q+1)\rho_{\mathcal{C}} \ln 3$.

Proof. First note that the running time can be proved in the same way as in the case of Theorem 1. Also, clearly, $\bar{D}$ is a dominating set in G. We claim that $G[\bar{D}]$ is a connected subgraph in G. Clearly $G[\bar{D}_i]$ is a connected subgraph and since G is connected so is $\mathbf{A}$. Consider the graph $\mathbf{C}(\mathbf{A})$ obtained from $\mathbf{A}$ by contracting each $\mathbf{U}_i$ to a single vertex. $\mathbf{C}(\mathbf{A})$ is clearly a connected graph. Since $C_i \subseteq D_i$, it is enough to note that whenever there is an edge $\{\mathbf{U}_i, \mathbf{U}_j\}$ in $\mathbf{C}(\mathbf{A})$ then there is a path P_{ij} in $G[\bar{D}]$ connecting a vertex from C_i with a vertex from C_j. To estimate $|\bar{D}|$ let us first show that

$$\sum_{i=1}^{l} |P_i| = O(\gamma_c(G)/\ln^q |G|). \tag{4}$$

By Lemma 7 part 2, the sum of degrees of vertices in $\mathbf{C}(\mathbf{A})$ is $O(|\mathbf{E}|/\ln^{q+1}|G|) = O(|\gamma_c(G)|/\ln^q|G|)$. Consequently, the number of P_{ij}'s is $O(|\gamma_c(G)|/\ln^q|G|)$. In addition, $|V(P_{ij})| \leq 4$ as if there is an edge $\{W, W'\} \in \mathbf{E}$ with $W \in \mathbf{U}_i$ and $W' \in \mathbf{U}_j$ then there exist $w \in W$ and $w' \in W'$ such that $\{w, w'\}$ is the edge in G. Since the center of W is in D_i and the center of W' is in D_j, the shortest path between D_i and D_j contains at most four vertices. Consequently, $\sum_{i=1}^{l}|\bar{D}_i| = \sum_{i=1}^{l}|D_i| + O(\gamma_c(G)/\ln^q|G|)$.

Finally from Lemma 8, there exists a connected dominating set D' in G such that $|D'| \leq (1 + O(1/\ln^q|G|))\gamma_c(G)$, $G[D' \cap U_i]$ induces a connected subgraph, $D' \cap U_i$ dominates U_i, and $C_i \subseteq D' \cap U_i$ for every i. Since D_i, found in the step 4(b), is a set of the smallest size such that $G[D_i]$ is a connected subgraph, D_i dominates U_i and $C_i \subseteq D_i$, we must have $|D_i| \leq |D' \cap U_i|$. As a result, $|\bar{D}| = \sum_{i=1}^{l}|\bar{D}_i| = \sum_{i=1}^{l}|D_i| + O(\gamma_c(G)/\ln^q|G|) \leq \sum_{i=1}^{l}|D' \cap U_i| + O(\gamma_c(G)/\ln^q|G|) = |D'| + O(\gamma_c(G)/\ln^q|G|)$ and so $|\bar{D}| \leq \left(1 + O\left(\frac{1}{\ln^q|G|}\right)\right)\gamma_c(G)$.

3.3 Planar Graphs

Class of planar graphs $\mathcal{P}$ has $\rho_{\mathcal{P}} = 3$ and so by Theorem 1 and Theorem 2 we have almost exact approximations for the minimum dominating set problem and the minimum connected dominating set problem in planar graphs that achieve the approximation error of $O(1/\ln^q|G|)$ and run in $O(\ln\ln|G|\ln^*|G|\ln^{1+r}|G|)$ rounds where $r = 18(q+1)\ln 3$. In [CHS06] an approximation algorithm for the minimum dominating set problem with $q = 1$ is given for the special subclass of planar graphs. The algorithm from [CHS06] runs in $O(\ln\ln|G|\ln^*|G|\ln^{1+r}|G|)$ rounds where $r = 27.7$ and so it is slightly faster than the algorithms from Theorem 1 and Theorem 2 which run $O(\ln\ln|G|\ln^*|G|\ln^{1+r}|G|)$ rounds where $r = 36\ln 3$. We can however apply techniques from [CHS06] and reduce the time complexity significantly. Using the SmallCluster procedure from [CHS06] and the fact that star arboricity of a planar graph is at most five, we can achieve the approximation error of $O(1/\ln^q|G|)$ in $O(\ln\ln|G|\ln^*|G|\ln^{1+r}|G|)$ rounds where $r < 5.54(q+1)$. In fact, using Tutte's Theorem on the tree arboricity of a graph with a bounded density of any subgraph (see [D97]), we can apply SmallCluster procedure and reduce the time complexity of our algorithms for minor-closed families. Due to space limitations, we will not give this refinement here.

References

[AGLP89] B. Awerbuch, A. V. Goldberg, M. Luby, S. A. Plotkin, Network Decomposition and Locality in Distributed Computation, Proc. 30th IEEE Symp. on Foundations of Computer Science , 1989, pp. 364–369.

[CV86] R. Cole, U. Vishkin, Deterministic coin tossing with applications to optimal parallel list ranking, Information and Control, 1986, 70, pp. 32-53.

[CH04] A. Czygrinow, M. Hańćkowiak, Distributed algorithms for weighted problems in sparse graphs, *Journal of Discrete Algorithms*, in press, (2004).

[CHS06] A. Czygrinow, M. Hańćkowiak, E. Szymańska, Distributed approximation algorithms for planar graphs, CIAC-2006, (2006).

[CH06] A. Czygrinow, M. Hańćkowiak, Distributed approximation algorithms in unit disk graphs, *manuscript*.

[D97] R. Diestel, Graph Theory, Springer, New York, (1997).

[DPRS03] D. Dubhashi, A. Mei, A. Panconesi, J. Radhakrishnan, and A. Srinivasan, Fast Distributed Algorithms for (Weakly) Connected Dominating Sets and Linear-Size Skeletons, In Proc. of the ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 717-724, 2003.

[E04] M. Elkin, An Overview of Distributed Approximation, in ACM SIGACT News Distributed Computing Column Volume 35, Number 4 (Whole number 132), Dec. 2004, pp. 40-57.

[KW03] F. Kuhn, R. Wattenhofer, Constant-Time Distributed Dominating Set Approximation, 22nd ACM Symposium on the Principles of Distributed Computing (PODC), Boston, Massachusetts, USA, July 2003.

[KMW04] F. Kuhn, T. Moscibroda, and R. Wattenhofer, What Cannot Be Computed Locally!, Proceedings of 23rd ACM Symposium on the Principles of Distributed Computing (PODC), 2004, pp. 300-309.

[KMNW05a] F. Kuhn, T. Moscibroda, T. Nieberg, and R. Wattenhofer, Fast Deterministic Distributed Maximal Independent Set Computation on Growth-Bounded Graphs, 19th International Symposium on Distributed Computing (DISC), Cracow, Poland, September (2005).

[KMNW05b] F. Kuhn, T. Moscibroda, T. Nieberg, and R. Wattenhofer, Local Approximation Schemes for Ad Hoc and Sensor Networks, 3rd ACM Joint Workshop on Foundations of Mobile Computing (DIALM-POMC), Cologne, Germany, (2005).

[KP95] S. Kutten, D. Peleg, Fast distributed construction of k-dominating sets and applications, Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing, 1995, pp. 238 - 251.

[L92] N. Linial, Locality in distributed graph algorithms, SIAM Journal on Computing, 1992, 21(1), pp. 193-201.

[L86] M. Luby, A simple parallel algorithm for the maximal independent set problem, SIAM J. Comput. 15, no. 4, (1986), 1036–1053.

[NM05] J. Nesetril and P. Ossona de Mendez, Colorings and homomorphisms of minor closed classes, In B. Aronov, S. Basu, J. Pach, and M. Sharir, eds., Discrete and Computational Geometry, The Goodman-Pollack Festschrift, vol. 25 of Algorithms and Combinatorics, pp. 651-664, Springer, 2003.

Spectral Clustering by Recursive Partitioning

Anirban Dasgupta[1,⋆], John Hopcroft[2], Ravi Kannan[3], and Pradipta Mitra[3,⋆⋆]

[1] Yahoo! Research Labs
[2] Department of Computer Science, Cornell University
[3] Department of Computer Science, Yale University

Abstract. In this paper, we analyze the second eigenvector technique of spectral partitioning on the planted partition random graph model, by constructing a recursive algorithm using the second eigenvectors in order to learn the planted partitions. The correctness of our algorithm is not based on the ratio-cut interpretation of the second eigenvector, but exploits instead the stability of the eigenvector subspace. As a result, we get an improved cluster separation bound in terms of dependence on the maximum variance. We also extend our results for a clustering problem in the case of sparse graphs.

1 Introduction

Clustering of graphs is an extremely general framework that captures a number of important problems on graphs. In a general setting, the clustering problem is to partition the vertex set of a graph into "clusters", where each cluster contains vertices of only "one type". The exact notion of what the vertex "type" represents is dependent on the particular application of the clustering framework. We will deal with the clustering problem on graphs generated by the versatile planted partition model (See [18, 5]). In this probabilistic model, the vertex set of the graph is partitioned into k subsets $T_1, T_2, \ldots, T_k$. Each edge (u, v) is then a random variable that is independently chosen to be present with a probability A_{uv}, and absent otherwise. The probabilities A_{uv} depend only on the parts to which the two endpoints u and v belong. The adjacency matrix $\widehat{A}$ of the random graph so generated is presented as input. Our task then is to identify the latent clusters $T_1, T_2, \ldots, T_k$ from $\widehat{A}$.

Spectral methods have been widely used for clustering problems, both for theoretical analysis as well as empirical and application areas. The underlying idea is to use information about the eigenvectors of $\hat{A}$ to extract structure. There are different variations to this basic theme of spectral clustering, which can be essentially divided into 2 classes of algorithms.

1. Projection heuristics, in which the top few eigenvectors of the adjacency matrix $\widehat{A}$ are used to construct a low-dimensional representation of the data, which is then clustered.

⋆ Work done when author was at Cornell University.

⋆⋆ Supported by NSF's ITR program under grant number 0331548.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 256–267, 2006.

2. The second eigenvector heuristic, in which the coordinates of the second eigenvector of $\widehat{A}$ is used to find a split of the vertex set into two parts. This technique is then applied recursively to each of the parts obtained.

Experimental results claiming the goodness of both spectral heuristics abound. Relatively fewer are results that strive to demonstrate provable guarantees about the heuristics. Perhaps more importantly, the worst case guarantees [17] that have been obtained do not seem to match the stellar performance of spectral methods on most inputs, and thus it is still an open question to characterize the class of inputs for which spectral heuristics do work well. In order to be able to formalize the average case behavior of spectral analysis, researchers have analyzed its performance on graphs generated by random models with latent structure [4, 18]. These graphs are generated by zero-one entries from a independently chosen according to a low-rank probability matrix. The low rank of the probability matrix reflects the small number of vertex types present in the unperturbed data. The intuition developed by Azar et al. [4] is that in such models, the random perturbations may cause the individual eigenvectors to vary significantly, but the subspace spanned by the top few eigenvectors remains stable. From this perspective, however, the second eigenvector technique does not seem to be well motivated, and it remains an open question as to whether we can claim anything better than the worst case bounds for the second eigenvector heuristic in this setting.

In this paper, we prove the goodness of the second eigenvector partitioning for the planted partition random graph model [11, 4, 18, 10]. We demonstrate that in spite of the fact that the second eigenvector itself is not stable, we can use it to recover the embedded structure.

Our main aim in analyzing the planted partition model using the second eigenvector technique is to try to bridge the gap between the worst case analysis and the actual performance. However, in doing so, we achieve a number of other goals too. The most significant among these is that we can get tighter guarantees than [18] in terms of the dependence on the maximum variance. The required separation between the columns clusters T_r and T_s can now be in terms of $\sigma_r + \sigma_s$, the maximum variances in each of these two clusters, instead of the maximum variance $\sigma_{\max}$ in the entire matrix. This gain could be significant if the maximum variance $\sigma_{\max}$ is due to only one cluster, and thus can potentially lead to identification of a finer structure in the data. Our separation bounds are however worse than [18, 1] in terms of dependence on the number of clusters. Another contribution of the paper is to model and solve a restricted clustering problem for sparse (constant degree) graphs. Graphs clustered in practice are often "sparse", even of very low constant degree. A concern about analysis of many heuristics on random models [18, 10] is that they don't cover sparse graphs. In this paper, we propose a model motivated by random regular graphs (see [14, 6], for example) for the clustering problem that allows us to use strong concentration results which are available in that setting. We will use some extra assumptions on the degrees of the vertices and finally show that expansion properties of the model will allow us to achieve a clean clustering through a simple algorithm.

2 Model and Our Results

A is a matrix of probabilities where the entry A_{uv} is the probability of an edge being present between the vertices u and v. The vertices are partitioned into k clusters $T_1, T_2, \ldots, T_k$. The size of the r^{th} cluster T_r is n_r and the minimum size is denoted by $n_{\min} = \min_r\{n_r\}$. Let, $w_{\min} = n_{\min}/n$. We assume that the minimum size $n_{\min} \in \Omega(n/k)$. The characteristic vector of the cluster T_r is denoted by $\boldsymbol{g}^{(r)}$ defined as $\boldsymbol{g}^{(r)}(i) = 1/\sqrt{n_r}$ for $i \in T_r$ and 0 elsewhere. The probability A_{uv} depends only on the two clusters in which the vertices u and v belong to. Given the probability matrix A, the random graph $\widehat{A}$ is then generated by independently setting each $\widehat{A}_{uv} (= \widehat{A}_{vu})$ to 1 with probability A_{uv} and 0 otherwise. Thus, the expectation of the random variable $\widehat{A}_{uv}$ is equal to A_{uv}. The variance of $\widehat{A}_{uv}$ is thus $A_{uv}(1 - A_{uv})$. The maximum variance of any entry of $\widehat{A}$ is denoted σ^2, and the maximum variance for all vertices belonging to a cluster T_r as denoted as σ_r^2. We usually denote a matrix of random variables by $\widehat{X}$ and the expectation of $\widehat{X}$ as $X = \mathbf{E}\left[\widehat{X}\right]$. We will also denote vectors by boldface (e.g. $\boldsymbol{x}$). $\boldsymbol{x}$ has the i^{th} coordinate $\boldsymbol{x}(i)$. For a matrix X, X_i denotes the column i. The number of vertices is n. We will assume the following separation condition.

Separation Condition. Each of the variances σ_r satisfies $\sigma_r^2 \geq \log^6 n/n$. Furthermore, there exists a large enough constant c such that for vertices $u \in T_r$ and $v \in T_s$, the columns A_u and A_v of the probability matrix A corresponding to different clusters T_r and T_s satisfy

$$\|A_u - A_v\|_2^2 \geq 64c^2k^5 \left(\sigma_r + \sigma_s\right)^2 \frac{\log(n)}{w_{\min}} \tag{1}$$

For clarity of exposition, we will make no attempt to optimize the constants or exponents of k. Similarly, we will ignore the term $w_{\min}$ for the most part. We say that a partitioning $(S_1, \ldots, S_l)$, respects the original clustering if the vertices of each T_r lie wholly in any one of the S_j. We will refer to the parts S_j as super-clusters, being the union of one or more clusters T_r. We say that a partitioning $(S_1, \ldots, S_l)$ agrees with the underlying clusters if each S_i is exactly equal to some T_r (i.e. $l = k$). The aim is to prove the following theorem.

Theorem 1. *Given $\widehat{A}$ that is generated as above, i.e. $A = \mathbf{E}\left[\widehat{A}\right]$ satisfies condition 1, we can cluster the vertices such that the partitioning agrees with the underlying clusters with probability at least $1 - \frac{1}{n^\delta}$, for suitably large δ.*

3 Related Work

The second eigenvector technique has been analyzed before, but mostly from the viewpoint of constructing cuts in the graph that have a small ratio of edges cut to vertices separated. There has been a series of results [13, 2, 5, 19] relating

the gap between the first and second eigenvalues, known as the Fiedler gap, to the quality of the cut induced by the second eigenvector. Spielman and Teng [20] demonstrated that the second eigenvector partitioning heuristic is good for meshes and planar graphs. Kannan et al. [17] gave a bicriteria approximation for clustering using the second eigenvector method. Cheng et al. [7] showed how to use the second eigenvector method combined with a particular cluster objective function in order to devise a divide and merge algorithm for spectral clustering. In the random graph setting, there has been results by Alon et al. [3], and Cojaoghlan [8] in using the coordinates of the second eigenvector in order to perform coloring, bisection and other problems. In each of these algorithms, however, the cleanup phase is very specific to the particular clustering task at hand.

Experimental studies done on the relative benefits of the two heuristics often show that the two techniques outperform each other on different data sets [21]. In fact results by Meila et al.[21] demonstrate that the recursive methods using the second eigenvector are actually more stable than the multiway spectral clustering methods if the noise is high. Anopther paper by Zhao et al. [23] shows that recursive clustering using the second eigenvector performs better than a number of other hierarchical clustering algorithms.

4 Algorithm

For the sake of simplicity, in most of the paper, we will be discussing the basic bipartitioning step that is at the core of of our algorithm. In Section 4.2 we will describe how to apply it recursively to learn all the k clusters. Define the matrix $J = I - \frac{1}{n}\mathbf{1}\mathbf{1}^T$. Note that for any vector $\boldsymbol{z}$ such that $\sum_i \boldsymbol{z}_i = 0$, $J\boldsymbol{z} = \boldsymbol{z}$. Given the original matrix $\widehat{A}$ we will create $\Theta(n/(k \log n))$ submatrices by partitioning the set of rows into $\Theta(n/(k \log n))$ parts randomly. Suppose $\widehat{C}$ denotes any one of these parts. Given the matrix $\widehat{C}$ as input, we will first find the top right singular vector $\boldsymbol{u}$ of the matrix $\widehat{C}J$. The coordinates of this vector will induce a mapping from the columns (vertices) of $\widehat{C}$ to the real numbers. We will find a large "gap" such that substantial number of vertices are mapped to both sides of the gap. This gives us a natural bipartition of the set of vertices of $\widehat{C}$. We will prove that this classifies all vertices correctly, except possibly a small fraction. This will be shown in Lemmas 2 to 6. We next need to "clean up" this bi-partitioning, and this will be done using a correlation graph construction along with a Chernoff bound. The algorithm and a proof will be furnished in Lemma 7. This completes one stage of recursion in which we create a number of superclusters all of which respect the original clustering. Subsequent stages proceed similarly. In what follows, we will be using the terms "column" and "vertex" interchangeably, noting that vertex x corresponds to column $\widehat{C}_x$.

4.1 Proof

For the standard linear algebraic techniques used in this section, we refer the reader to [16]. Recall that each $\widehat{C}$ is a $\frac{n}{2k \log n} \times n$ matrix where the rows are

chosen randomly. Denote the expectation of $\widehat{C}$ by $\mathbf{E}\left[\widehat{C}\right] = C$, and by $\boldsymbol{u}$ the top right singular vector of $\widehat{C}J$, i.e. the top eigenvector of $(\widehat{C}J)^T\widehat{C}J$. In what follows, we demonstrate that for each of random submatrices $\widehat{C}$, we can utilize the second right singular vector $\boldsymbol{u}$ to create a partitioning of the columns of $\widehat{C}J$ that respects the original clustering. The following fact is intuitive and will be proven later in lemma 8, when we illustrate the full algorithm.

Fact 1. *$\widehat{C}$ has at least $\frac{n_r}{2k\log n}$ rows for each cluster T_r.*

Let $\sigma = \max_r\{\sigma_r\}$, where the maximum is taken only over clusters present in $\widehat{C}$ (and therefore, potentially much smaller than $\sigma_{\max}$). We also denote $C(r,s)$ for the entries of C corresponding to vertices of T_r and T_s. The following result is from Furedi-Komlos and more recently, Vu [22, 15] claiming that a matrix of i.i.d. random variables is close to its expectation in the spectral norm.

Lemma 2. (Furedi, Komlos; Vu) *If $\widehat{X}$ is a 0/1 random matrix with expectation $X = \mathbf{E}\left[\widehat{X}\right]$, and the maximum variance of the entries of $\widehat{X}$ is σ^2 which satisfies $\sigma^2 \geq \log^6 n/n$,[1] then with probability $1 - o(1)$,*

$$\|X - \widehat{X}\|_2 < 3\sigma\sqrt{n}$$

In particular, we have $\|C - \widehat{C}\|_2 < 3\sigma\sqrt{n}$.

The following lemmas will show that the top right singular vector $\boldsymbol{u}$ of $\widehat{C}J$ gives us an approximately good bi-partition.

Lemma 3. *The first singular value λ_1 of the expected matrix CJ satisfies $\lambda_1(CJ) \geq 2c(\sigma_r + \sigma_s)k^2\sqrt{n}$ for each pair of clusters r and s that belong to C. Thus, in particular, $\lambda_1(CJ) \geq 2c\sigma k^2\sqrt{n}$.*

Proof. Suppose $\widehat{C}$ has the clusters T_r and T_s, $r \neq s$. Assume $n_r \leq n_s$. Consider the vector $\boldsymbol{z}$ defined as :

$$\boldsymbol{z}_x = \begin{cases} \frac{1}{\sqrt{2n_r}} & \text{if } x \in T_r \\ -\frac{\sqrt{n_r}}{n_s\sqrt{2}} & \text{if } x \in T_s \\ 0 & \text{otherwise} \end{cases}$$

Now, $\sum_x \boldsymbol{z}(x) = \frac{n_r}{\sqrt{2n_r}} - \frac{\sqrt{n_r}}{\sqrt{2}n_s}n_s = 0$. Also, $\|\boldsymbol{z}\|^2 = \frac{n_r}{2n_r} + \frac{n_r n_s}{2n_s^2} = \frac{1}{2} + \frac{1}{2}\frac{n_r}{n_s} \leq 1$. Clearly, $\|\boldsymbol{z}\| \leq 1$. For any row C^j from a cluster T_t, it can be shown that $C^j \cdot \boldsymbol{z} = \sqrt{\frac{n_r}{2}}(C(r,t) - C(s,t))$. We also know from fact 1 that there are at least $n_t/(2k\log n)$ such rows. Now,

[1] In fact, in light of recent results in [12] this holds for $\sigma^2 \geq C'\log n/n$, with a different constant in the concentration bound.

$$\|CJ\boldsymbol{z}\|^2 \geq \sum_j (C^j \cdot \boldsymbol{z})^2 = \sum_t \sum_{j \in T_t} (C^j \cdot \boldsymbol{z})^2 \geq \sum_t \frac{n_t}{2k \log n} \frac{n_r}{2} (C(r,t) - C(s,t))^2$$
$$= \frac{n_r}{4k \log n} \sum_t n_t (C(r,t) - C(s,t))^2 \frac{n_r}{4k \log n} \|C_r - C_s\|_2^2$$
$$\geq 64 \frac{n_r}{4k \log n} c^2 k^5 \left(\sigma_r + \sigma_s\right)^2 \log(n)/w_{\min} \geq 16 c^2 n k^4 (\sigma_r + \sigma_s)^2$$

using the separation condition and the fact that n_r is at least $w_{\min} n$. And thus $\lambda_1(CJ)$ is at least $4c(\sigma_r + \sigma_s)k^2\sqrt{n}$. Note that the 4th step uses the separation condition (1). □

The above result, combined with the fact the the spectral norm of the random perturbation being small immediately implies that the norm of the matrix $\widehat{C}J$ is large too. Thus,

Lemma 4. *The top singular value of $\widehat{C}J$ is at least $c\sigma k^2\sqrt{n}$.*

Proof. Proof omitted.

Lemma 5. *The vector $\boldsymbol{u}$, the top right singular vector of $\widehat{C}J$ can be written as $\boldsymbol{u} = \boldsymbol{v} + \boldsymbol{w}$ where both $\boldsymbol{v}, \boldsymbol{w}$ are orthogonal to $\mathbf{1}$ and further, $\boldsymbol{v}$ is a linear combination of the indicator vectors $\boldsymbol{g}^{(1)}, \boldsymbol{g}^{(2)}, \ldots$ for clusters T_r that have vertices in the columns of $\widehat{C}$. Also, $\boldsymbol{w}$ sums to zero on each T_r. Moreover,*

$$\|\boldsymbol{w}\| \leq \frac{4}{ck^2} \tag{2}$$

Proof. We may define the two vectors $\boldsymbol{v}$ and $\boldsymbol{w}$ as follows: $\boldsymbol{v} = \sum_r (\boldsymbol{g}^{(r)} \cdot \boldsymbol{u})\boldsymbol{g}^{(r)}, \boldsymbol{w} = \boldsymbol{u} - \boldsymbol{v}$.

It is easy to check that $\boldsymbol{w}$ is orthogonal to $\boldsymbol{v}$, and that $\sum_{x \in T_r} \boldsymbol{w}(x) = 0$ on every cluster T_r. Thus both $\boldsymbol{v}$ and $\boldsymbol{w}$ are orthogonal to $\mathbf{1}$. As $\boldsymbol{v}$ is orthogonal to $\boldsymbol{w}$, $\|\boldsymbol{v}\|^2 + \|\boldsymbol{w}\|^2 = \|\boldsymbol{u}\|^2 = 1$. Now,

$$\lambda_1(\widehat{C}J) = \|\widehat{C}J\boldsymbol{u}\| \leq \|\widehat{C}J\boldsymbol{v}\| + \|\widehat{C}J\boldsymbol{w}\| \leq \lambda_1(\widehat{C}J)\|\boldsymbol{v}\| + \|CJ\boldsymbol{w}\| + \|CJ - \widehat{C}J\|\|\boldsymbol{w}\|$$
$$\leq \lambda_1(\widehat{C}J)(1 - \|\boldsymbol{w}\|^2/2) + \|C - \widehat{C}\|\|\boldsymbol{w}\|$$

using the fact that $(1-x)^{1/2} \leq 1 - \frac{x}{2}$ for $0 \leq x \leq 1$, and also noting that $J\boldsymbol{w} = \boldsymbol{w}$, and therefore $CJ\boldsymbol{w} = C\boldsymbol{w} = 0$. Thus, from the above, $\|\boldsymbol{w}\| \leq \frac{2\|C - \widehat{C}\|}{\lambda_1(\widehat{C}J)} \leq \frac{4\sigma\sqrt{n}}{c\sigma k^2\sqrt{n}} \leq \frac{4}{ck^2}$ using Lemma 2 and Lemma 4. □

We now show that in bi-partitioning each $\widehat{C}$ using the vector $\boldsymbol{u}$, we only make mistakes for a small fraction of the columns.

Lemma 6. *Given the top right singular vector $\boldsymbol{u}$ of $\widehat{C}$, there is a way to bipartition the columns of $\widehat{C}$ based on $\boldsymbol{u}$, such that all but $\frac{n_{\min}}{ck}$ columns respect the underlying clustering of the probability matrix C.*

Proof. Consider the following algorithm. Consider the real values $\boldsymbol{u}(x)$ corresponding to the columns $\widehat{C}_x$.

1. Find β such that at most $\frac{n}{ck^2}$ of the $\boldsymbol{u}(x)$ lies in $(\beta, \beta + \frac{2}{k\sqrt{n}})$. Moreover, define $L = \{x : \boldsymbol{u}(x) < \beta + \frac{1}{k\sqrt{n}}\}; R = \{x : \boldsymbol{u}(x) \geq \beta + \frac{1}{k\sqrt{n}}\}$. It must be that both $|L|$ and $|R|$ are at least $n_{\min}/2$. Note that $\widehat{C} = L \cup R$. If we cannot find any such gap, don't proceed (a cluster has been found that can't be partitioned further).
2. Take $L \cup R$ as the bipartition.

We must show that, if the vertices contain at least two clusters, a gap of $\frac{2}{k\sqrt{n}}$ exists with at least $n_{\min}/2$ vertices on each side. For simplicity, for this proof we assume that all clusters are of equal size (the general case will be quite similar). Let $\boldsymbol{v} = \sum_{r=1}^{k} \alpha_r \boldsymbol{g}^{(r)}$. Recall that $\boldsymbol{v}$ is orthogonal to $\mathbf{1}$, and thus $\sum_{r=1}^{k} \alpha_r \sqrt{\frac{k}{n}} = 0$. Now note that $1 = \|\boldsymbol{u}\|^2 = \|\boldsymbol{v}\|^2 + \|\boldsymbol{w}\|^2$. This and lemma 5 gives us

$$\sum_{r=1}^{k} \alpha_r^2 \geq 1 - \frac{16}{c^2 k} \geq \frac{1}{2} \tag{3}$$

We claim that there is an interval of $\Theta\left(\frac{1}{k\sqrt{k}}\right)$ on the real line such that no α_r lies in this interval and at least one α_r lies on each side of the interval. We will call such a gap a "proper gap". Note that a proper gap will partition the set of vertices into two parts such that there are at least $n_{\min}/2$ vertices on each side of it.

The above claim can be proved using basic algebra. We will omit details here. Thus, it can been seen that for some constant c, there will be a proper gap of $\frac{1}{ck\sqrt{n}}$ in the vector $\boldsymbol{v}$. We then argue that most of the coordinates of $\boldsymbol{w}$ are small and do not spoil the gap. Since the norm of $\|\boldsymbol{w}\|^2$ is bounded by $16/(c^2k^4)$, it is straightforward to show that at most $\frac{n}{ck^2}$ vertices x can have $\boldsymbol{w}(x)$ over $\frac{4}{k\sqrt{cn}}$. This shows that for most vertices $\boldsymbol{w}(x)$ is small and will not "spoil" the proper gap in $\boldsymbol{v}$. Thus, with high probability, the above algorithm of finding a gap in $\boldsymbol{u}$ always succeeds. Next we have to show that any such gap that is found from $\boldsymbol{u}$ actually corresponds to a proper gap in $\boldsymbol{v}$. Since there must be at least $n_{\min}/2$ vertices on each side of the gap in $\boldsymbol{u}$, and since the values $\boldsymbol{u}(x)$ and $\boldsymbol{v}(x)$ are close (i.e. $\boldsymbol{w}(x) = \boldsymbol{u}(x) - \boldsymbol{v}(x)$ is smaller than $1/(2k\sqrt{n})$) except for $\frac{n}{ck}$ vertices, it follows that a proper gap found in $\boldsymbol{u}$ must correspond to a proper gap in $\boldsymbol{v}$. Thus the only vertices that can be misclassified using this bi-partition are the vertices that are either in the gap, or have $\boldsymbol{w}(x)$ larger than $\frac{1}{k\sqrt{n}}$. Given this claim, it can be seen a using a proper gap a bi-partition of the vertices can be found with at most $\frac{n}{2ck^2} \approx \Theta\left(\frac{n_{\min}}{ck}\right)$ vertices on the wrong side of the gap. □

A natural idea for the "clean up" phase would be to use $\log n$ independent samples of $\widehat{C}$ (thus requiring the $\log n$ factor in the separation) and try to use a Chernoff bound argument. This argument doesn't work, unfortunately, the

reason being that the singular vector can induce different bi-partitions for each of the $\widehat{C}$'s. For instance, if there are 3 clusters in the original data, then in the first step we could split any one of the three clusters from the other two. This means a naive approach will need to account for all possible bi-partitionings and hence require an extra 2^k in the separation condition. The following lemma deals with this problem:

Lemma 7. *Suppose we are given set V that is the union of a number of clusters $T_1 \cup \ldots \cup T_t$. Given $p = ck \log n$ independent bi-partitions of the set of columns V, such that each bi-partition agrees with the underlying clusters for all but $\frac{n_{\min}}{4ck}$ vertices, there exists an algorithm that, with high probability, will compute a partitioning of the set V such that*

- *The partitioning respects the underlying clusters of the set V.*
- *The partitioning is non-trivial, that is, if the set V contains at least two clusters, then the algorithm finds at least two partitions.*

Proof. Consider the following algorithm. Denote $\varepsilon = \frac{1}{4ck}$.

1. Construct a (correlation) graph H over the vertex set V.
2. Two vertices x and y are adjacent if they are on the same L or R for at least $(1 - 2\varepsilon)$ fraction of the bi-partitions.
3. Let $N_1, \ldots, N_l$ be the connected components of this graph. Return $N_1, \ldots, N_l$.

We now need to prove that the following claims hold with high probability : 1. N_j respects the cluster boundary, i.e. each cluster T_r that is present in V satisfies $T_r \subseteq N_{j_r}$ for some j_r; and 2. If there are at least two clusters present in V, i.e. $t \geq 2$, then there are at least two components in H. For two vertices $x, y \in H$, let the **support** $s(x, y)$ equal the fraction of tests such that x and y are on the same side of the bi-partition. For the first claim, we define a vertex x to be a "bad" vertex for the i^{th} test if $|w(x)| > \frac{1}{k\sqrt{cn}}$. From lemma 6 the number of bad vertices is clearly at most $\frac{1}{ck} n_{\min}$. It is clear that a misclassified vertex x must either lie in the gap $(\beta, \beta + \frac{2}{k\sqrt{n}})$ or it must be a bad one. So for any vertex x, the probability that x is misclassified in the i^{th} test is at most $\varepsilon = 1/(4ck)$. If there are p tests, then the expected times that a vertex x is misclassified is at most εp. Supposing Y_x^i is the indicator random variable for the vertex x being misclassified in the i^{th} test. Thus, $\mathbf{Pr}\left[\sum_i Y_x^i > 2\varepsilon p\right] < \exp\left(-\frac{16p}{ck}\right) < \frac{1}{n^3}$ since $p = ck \log n$. Thus, each pair of vertices in a cluster, are on the same side of the bipartition for at least $(1 - 2\varepsilon)$ fraction of the tests. Clearly, the components N_j always obey the cluster partitions.

Next, we have to prove the second claim. For contradiction, assume there is only one connected component. We know, that if $x, y \in T_r$ for some r, the fraction of tests on which they landed on same side of partition is $s(x, y) \geq (1 - 2\varepsilon)$. Hence the subgraph induced by each T_r is complete. With at most k clusters in V, this means that any two vertices x, y (not necessarily in the same cluster) are

separated by a path of length at most k. Clearly $s(x,y) \geq (1-2k\varepsilon)$. Hence, the total support of inter-cluster vertex pairs is

$$\sum_{r\neq s}\sum_{x\in T_r, y\in T_s} s(x,y) \geq (1-2k\varepsilon)\sum_{r\neq s} n_r n_s \geq \sum_{r\neq s} n_r n_s - 2k\varepsilon \sum_{r\neq s} n_r n_s. \quad (4)$$

Let us count this same quantity by another method. From Lemma 6, it is clear that for each test at least one cluster was separated from the rest (apart from small errors). Since by the above argument, all but ε vertices are good, we have that, at least $n_{\min}(1-\varepsilon)$ vertices were separated from the rest. Hence the total support is

$$\sum_{r\neq s}\sum_{x\in T_r, y\in T_s} s(x,y) \leq \sum_{r\neq s} n_r n_s - n_{\min}(1-\varepsilon)(n - n_{\min}(1-\varepsilon)) < \sum_{r\neq s} n_r n_s - n_{\min} n/2$$

But this contradicts equation 4 if $2k\varepsilon \sum_{r\neq s} n_r n_s < n_{\min} n/2$ i.e. $\varepsilon < \frac{n_{\min} n/4}{k\sum_{r\neq s} n_r n_s} < \frac{n_{\min} n/2}{kn^2} \leq \frac{1}{2ck}$. With the choice of $\varepsilon = 1/(4ck)$, we get a contradiction. Hence the correlation graph satisfies the properties claimed. □

4.2 Final Algorithm

We now describe the complete algorithm. Basically, it is the bi-partitioning technique presented in the previous section repeated (at most) k times applied to the matrix $\widehat{A}$.

Algorithm 1. Cluster $(\widehat{A}, k)$

Partition the set of rows into k random equal parts, each part to be used in the corresponding step of recursion. Name the i^{th} part to be $\widehat{B}_i$.
Let $(S_1, \ldots, S_l)$ =**Bi-Partition** $(\widehat{B}_1, k)$.
Recursively call **Bi-Partition** on each of S_i, and on each of the results, using the appropriate columns of a separate $\widehat{B}_j$ for each call. The recursion ends when the current call returns only one S_i. Let $\widehat{T}_1, \ldots, \widehat{T}_k$ be the final groups.

As the split in every level is "clean", as we have shown above, the whole analysis goes through for recursive steps without any problems. In order to de-condition the steps of the recursion, we have to first create k independent instances of the data by partitioning the rows of the matrix $\widehat{A}$ into a k equally sized randomly chosen sets. This creates a collection of rectangular matrices $\widehat{B}_1, \ldots, \widehat{B}_k$. The module **Bi-Partition**$(\widehat{X}, k)$ on being invoked with the matrix $\widehat{X}$ and the cluster parameter k consists of two phases: an approximate partitioning by the singular vector, followed by a clean-up phase. The rows of matrix $\widehat{X}$ are further subdivided to create a number of rectangular matrices $\widehat{C}^{(i)}$, which correspond to C that we used in our analysis of the bi-partitioning phase. One thing we still need to prove that the fact 1 made for $\hat{C}$ in the beginning of section 4.1 is valid for $C^{(i)}$.

Algorithm 2. Bi-Partition $(\widehat{X}, k)$

Partition the set of rows into $c_1 \log n$ equal parts randomly. The i^{th} set of rows forms the matrix $\widehat{C}^{(i)}$.
For each $\widehat{C}^{(i)}$, find the right singular vector of $\widehat{C}^{(i)}J$ and call it $\boldsymbol{u}_i$.
Split:
Find a proper gap β, such that $(\beta, \beta + \frac{2}{k\sqrt{n}})$ has at most $\frac{n}{c_2 k}$ vertices and define

$$L_i = \{x : u_i(x) < \beta + \frac{1}{k\sqrt{n}}\}$$

$$R_i = \{x : u_i(x) \geq \beta + \frac{1}{k\sqrt{n}}\}$$

$$|L_i| \geq n_{\min}/2; |R_i| \geq n_{\min}/2$$

$$\widehat{C}^{(i)} = L_i \cup R_i$$

If no such gap exists, return.
Cleanup:
Construct a (correlation) graph with the columns of $\widehat{X}$ as the vertices.
Connect two vertices x and y if they are on the same L_i or R_i for at least $(1 - \frac{1}{2ck}) \log n$ times. Let $N_1, \ldots, N_l$ be the connected components of this graph. Return $N_1, \ldots, N_l$.

Lemma 8. *Consider each matrix* $C^{(i)} = \mathbf{E}\left[\widehat{C}^{(i)}\right]$. *W.h.p. there are at least* $\frac{n_r}{2c_1 k \log n}$ *rows in* $C^{(i)}$ *corresponding to* T_r.

Proof. In each $\widehat{C}^{(i)}$, the expected number of rows from each T_r is $\frac{n_j}{k \times c_1 \log n}$. Using Chernoff bound, the number of rows contributed by each cluster T_r to the matrix $\widehat{C}^{(i)}$ is at least $\frac{n_j}{2c_1 k \log n}$ with probability $1 - \exp[-\frac{n_j}{2c_1 k \log n}] \geq 1 - \frac{1}{n^3}$. Thus, over the all random partitions, w.p. $1 - \frac{1}{n^2}$, the statement is true.

5 Sparse Graphs

5.1 Our Model and Related Work

The input $\hat{A}$ is a n-vertex undirected graph. There will be k clusters in the graph, with $n_r = \Omega(n)$ being the size of cluster T_r. Let x $\in T_r$. Then we assume that the number of edges from x to vertices of T_s:

$$e(x, T_s) = d_{rs} \tag{5}$$

For some constant d_{rs}. We assume that these constants satisfy $n_r d_{rs} = n_s d_{sr}$.

Let $\hat{A}^{(rs)}$ be the submatrix of $\hat{A}$ containing rows corresponding to T_r and columns corresponding to T_s. Then $\hat{A}^{(rs)}$ is a matrix randomly chosen from all matrices satisfying equation 5 (to account for symmetry $\hat{A}^{(rs)} = (\hat{A}^{(rs)})^T$).

Let $A = \mathbf{E}\left[\hat{A}\right]$. If vertex $x \in T_r$, let $A_x = \mu_r$. Note that $\mu_r(x) = \mu_s(y)$ where $y \in T_s; x \in T_r$ due to symmetry. Let d be a upper bound for vertex degree in

the graph. We will assume, for all r, and some constant c_0, $d_{rr} \geq \frac{1}{2}d + c_0\sqrt{dk}$ Which will now imply something we need: $\|\mu_r - \mu_s\|_2^2 \geq c_0^2 k^2 \frac{d}{n}$.

Among previous works on "sparse" graphs are results by Alon and Kahale [3] (3 coloring) and Coja-Oghlan [8, 9] (bisection, clustering). Our results are not comparable to theirs as those models are only sparse "on average". Nevertheless, the gap required in [8], improving on [5], is $np' - np = \Theta(\sqrt{np' \log np'})$ in a $G(n, p')$ model, which put in our terminology is $\Theta(\sqrt{d \log d})$, similar to our separation (in fact we don't need the $\log d$ factor). The separation in [3] is d. It should be emphasized again that both settings are quite different from ours. A d-regular model for bisection was studied by Bui et. al. [6]. They present an algorithm that finds bisections of width (cardinality of the bisection) $o(n^{1-1/(d/2)})$ from a graph that is randomly chosen from d-regular graphs having such a bisection. We depend on having different d_{rr}'s for different clusters for a notion of partitioning, and in any case we seek to solve a more general problem.

5.2 Algorithm

For sets (of vertices) U and W, let $e(U, W)$ be the the number of edges between U and W. Here we only present the "clean up" phase as everything else remains essentially the same. Our result is that if the separation condition holds, this algirthm will successfully cluster the vertices. We omit the proof of this fact here.

Algorithm 3. SparseCleanup(P_1, P_2, d)

loop
  find a vertex v in P_2' such that $e(v, P_1') > e(v, P_2')(1 + \frac{1}{2\sqrt{d}})$
  if no vertex can be found, end loop.
  move v to P_1'
end loop
loop
  find a vertex v in P_1' such that $e(v, P_2') > e(v, P_1')(1 + \frac{1}{2\sqrt{d}})$
  if no vertex can be found, end loop.
  move v to P_2'
end loop

References

1. Dimitris Achlioptas and Frank McSherry, *On spectral learning of mixtures of distributions*, Conference on Learning Theory (COLT) 2005, 458-469.
2. Noga Alon, *Eigenvalues and expanders*, Combinatorica, **6(2)**, (1986) , 83-96.
3. Noga Alon and Nabil Kahale, *A spectral technique for coloring random 3-colorable graphs*, SIAM Journal on Computing **26** (1997), n. 6. 1733-1748.
4. Yossi Azar, Amos Fiat, Anna R. Karlin, Frank McSherry and Jared Saia, *Spectral analysis of data*, Proceedings of the 32^{nd} annual ACM Symposium on Theory of computing (2001), 619-626.

5. Ravi Boppana, *Eigenvalues and graph bisection: an average case analysis*, Proceedings of the 28^{th} IEEE Symposium on Foundations of Computer Science (1987).
6. Thang Bui, Soma Chaudhuri, Tom Leighton and Mike Sipser, *Graph bisection algorithms with good average case behavior*, Combinatorica, **7**, 1987, 171-191.
7. David Cheng, Ravi Kannan, Santosh Vempala and Grant Wang, *A Divide-and-Merge methodology for Clustering*, Proc. of the 24^th ACM SIGMOD-SIGACT-SIGART symposium on principles of database systems (PODS), 196 - 205.
8. Amin Coja-Oghlan, *A spectral heuristic for bisecting random graphs*, Proceedings of the 16^th Annual ACM-SIAM Symposium on Discrete Algorithms, 2005.
9. Amin Coja-Oghlan, *An adaptive spectral heuristic for partitioning random graphs*, Automata, Languages and Programming, 33^{rd} International Colloquium, ICALP, Lecture Notes in Computer Science 4051 Springer 2006.
10. Anirban Dasgupta, John Hopcroft and Frank McSherry, *Spectral analysis of random Graphs with skewed degree distributions*, Proceedings of the 42^{nd} IEEE Symposium on Foundations of Computer Science (2004), 602-610.
11. Martin Dyer and Alan Frieze, *Fast Solution of Some Random NP-Hard Problems*, Proceedings of the 27^{th} IEEE Symposium on Foundations of Computer Science (1986), 331-336
12. Uriel Feige and Eran Ofek, *Spectral techniques applied to sparse random graphs*, Random Structures and Algorithms, **27(2)**, 251–275, September 2005.
13. M Fiedler, *Algebraic connectibility of graphs*, Czechoslovak Mathematical Journal, **23(98)**, (1973), 298-305.
14. Joel Friedman, Jeff Kahn and Endre Szemeredi, *On the second eigenvalue of random regular graphs*, Proceedings of the 21^{st} annual ACM Symposium on Theory of computing (1989), 587 - 598.
15. Zoltan Furedi and Janos Komlos, *The eigenvalues of random symmetric matrices*, Combinatorica 1, **3**, (1981), 233–241.
16. G. Golub, C. Van Loan (1996), *Matrix computations, third edition*, *The Johns Hopkins University Press Ltd., London.*
17. Ravi Kannan, Santosh Vempala and Adrian Vetta, *On Clusterings : Good, bad and spectral*, Proceedings of the Symposium on Foundations of Computer Science (2000), 497 - 515.
18. Frank McSherry, *Spectral partitioning of random graphs*, Proceedings of the 42^{nd} IEEE Symposium on Foundations of Computer Science (2001), 529-537.
19. Alistair Sinclair and Mark Jerrum, *Conductance and the mixing property of markov chains*, the approximation of the permenant resolved, Proc. of the 20^{th} annual ACM Symposium on Theory of computing (1988), 235-244.
20. Daniel Spielman and Shang-hua Teng, *Spectral Partitioning Works: Planar graphs and finite element meshes*, Proc. of the 37th Annual Symposium on Foundations of Computer Science (FOCS '96), 96 - 105.
21. Deepak Verma and Marina Meila, *A comparison of spectral clustering algorithms*, TR UW-CSE-03-05-01, Department of Computer Science and Engineering, University of Washington (2005).
22. Van Vu, *Spectral norm of random matrices*, Proc. of the 36^{th} annual ACM Symposium on Theory of computing (2005), 619-626.
23. Ying Zhao and George Karypis, *Evaluation of hierarchical clustering algorithms for document datasets*, Proc. of the 11 International Conference on Information and Knowledge Management (2002), 515 - 524.

Finite Termination of "Augmenting Path" Algorithms in the Presence of Irrational Problem Data

Brian C. Dean[1], Michel X. Goemans[2], and Nicole Immorlica[3]

[1] Department of Computer Science, Clemson University
[2] Department of Mathematics, M.I.T.
[3] Microsoft Research

Abstract. This paper considers two similar graph algorithms that work by repeatedly increasing "flow" along "augmenting paths": the Ford-Fulkerson algorithm for the maximum flow problem and the Gale-Shapley algorithm for the stable allocation problem (a many-to-many generalization of the stable matching problem). Both algorithms clearly terminate when given integral input data. For real-valued input data, it was previously known that the Ford-Fulkerson algorithm runs in polynomial time if augmenting paths are chosen via breadth-first search, but that the algorithm might fail to terminate if augmenting paths are chosen in an arbitrary fashion. However, the performance of the Gale-Shapley algorithm on real-valued data was unresolved. Our main result shows that, in contrast to the Ford-Fulkerson algorithm, the Gale-Shapley algorithm always terminates in finite time on real-valued data. Although the Gale-Shapley algorithm may take exponential time in the worst case, it is a popular algorithm in practice due to its simplicity and the fact that it often runs very quickly (even in sublinear time) for many inputs encountered in practice. We also study the Ford-Fulkerson algorithm when augmenting paths are chosen via depth-first search, a common implementation in practice. We prove that, like breadth-first search, depth-first search also leads to finite termination (although not necessarily in polynomial time).

1 Introduction

The Ford-Fulkerson (FF) algorithm for the s-t maximum flow problem [2] is remarkably simple to describe and implement: as long as we can find an "augmenting path" along which additional flow can be sent from a source node s to a sink node t, send as much flow along the path as possible.

A close relative of the FF algorithm is the Gale-Shapley (GS) algorithm [3] adapted for the stable allocation problem, a many-to-many generalization of the stable matching problem. In this problem, we are assigning, say, a set of jobs to machines, where the jobs have varying processing times and the machines have varying capacities. Each job submits a ranked preference list over machines on

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 268–279, 2006.

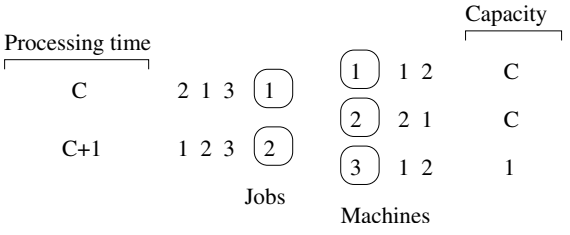


Fig. 1. An example instance on which the GS algorithm requires exponential time. Regardless of the proposal order of the jobs, the algorithm will perform at least C proposals before converging to the unique stable assignment.

which it may be processed, and each machine submits a ranked preference list over jobs that it may process. The stable allocation problem involves finding a feasible (fractional) assignment of jobs to machines in which no job/machine pair has an incentive to deviate from the assigned solution in a sense to be described later. The GS algorithm for the stable allocation problem is a generalization of the original GS algorithm for the simpler stable matching problem (where we are assigning n unit-sized jobs to n unit-sized machines): each job proceeds down its preference list issuing "proposals" to machines, and each machine tentatively accepts the best proposal received thus far. If a machine receives a proposal from a job it prefers more than its current tentative assignment, it accepts the proposal and rejects the job to which it is currently assigned, and this job which then continues issuing proposals to machines further down its preference list. In the stable allocation problem where jobs and machines have non-unit sizes, the GS algorithm operates in an identical fashion, except proposals and rejections now happen in non-unit quantities. When a job proposes to a machine, it proposes all of its unassigned load, and a machine may chose to "fractionally" accept only part of this load and reject the rest, depending on its preference among its current assignments. A sequence of proposals and rejections can be interpreted as an augmenting path.

When edge capacities are integral in the FF algorithm, or when processing times and capacities are integral in the GS algorithm, each augmentation pushes at least 1 unit of flow and so the algorithms clearly terminate. However, in the FF algorithm, for certain graphs with real-valued edge capacities, if we choose augmenting paths in a completely arbitrary fashion then we may fail not only to terminate, but also to converge to an optimum flow [2, 5]. The main contribution of this paper is to show that, in contrast to the FF algorithm, the GS algorithm always terminates in finite time for real-valued inputs. This resolves an open question of Baiou and Balinksi [1].

Unfortunately, convergence of the GS algorithm may take exponential time in the worst case (Figure 1), whereas an alternative algorithm has been proposed by Baiou and Balinksi [1] that runs in strongly polynomial time. However, the GS algorithm is perhaps more common in practice due to its simplicity and the fact that for many instances in practice, its running time can be significantly

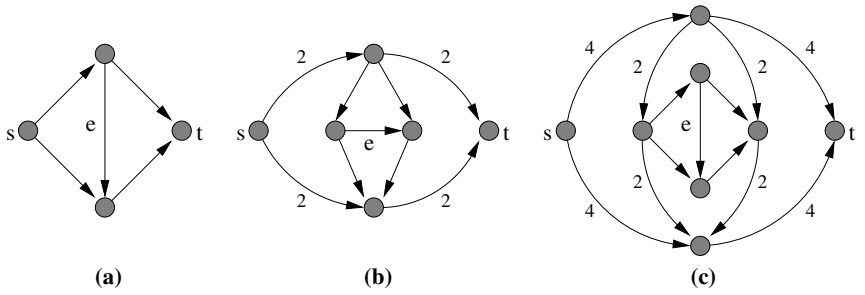


Fig. 2. Examples of graphs for which the FF algorithm takes exponential time to run, using DFS to locate augmenting paths. Unlabeled edges have unit capacity. In (a), a DFS that prioritizes edge e will lead to two augmenting paths that each utilize e (in alternating directions). In (b), we recursively expand the graph by replacing e with a copy of the entire graph, leading to four augmenting paths (each of which uses e in alternating directions). The graph in (c) requires eight augmenting paths, and so on. Although this example is not bipartite, one can construct bipartite graphs that behave similarly.

faster than that of the Baiou-Balinski algorithm (in fact, the GS algorithm often runs in sublinear time). It is therefore reassuring to know that termination of the GS algorithm is guaranteed for real-valued inputs.

We also study the FF algorithm when augmenting paths are chosen according to a depth-first search (DFS). While it is well-known that the selection of augmenting paths using breadth-first search (BFS) always leads to finite termination as well as a strongly polynomial running time, the use of DFS is another regrettably common approach for selecting augmenting paths. Even with integral capacities, the use of DFS is to be discouraged since it can lead to exponential worst-case running times (Figure 2); however, our techniques allow us to give a simple proof that at the very least, the FF algorithm implemented with DFS always terminates even when edge capacities are real-valued. Our results for the FF algorithm are primarily of theoretical interest, since there is little reason to use DFS rather than BFS in practice to locate augmenting paths.

The structure of this paper is as follows. The next section contains our proof of finite termination for the FF algorithm using DFS to locate augmenting paths. Following that, we give a more detailed introduction to both the stable allocation problem and the GS algorithm, and build towards our main result that the GS algorithm always terminates in finite time.

2 The FF Algorithm with Irrational Capacities

Consider the use of DFS to locate augmenting paths for the FF algorithm. In this case, every time we visit a node i during an augmenting path search, we recursively search the outgoing edges from i according to some deterministic

ordering, and we keep using the same outgoing edge as long as this enables us to find valid augmenting paths to t.

Observation 1. *An augmenting path will never include an edge directed into the source node.*

Theorem 1. *The Ford-Fulkerson algorithm terminates in finite time, even with real-valued edge capacities, if we use DFS to find augmenting paths.*

Proof. Starting from the source node s at the beginning of the algorithm, let i be the first neighboring node chosen by the algorithm to visit. All of our augmenting paths will continue to utilize the edge (s, i) until either (i) the flow along (s, i) reaches the capacity of (s, i), or (ii) no residual augmenting $i \rightsquigarrow t$ path exists that does not include s. We first argue that (i) or (ii) will occur within a finite amount of time. This follows by induction on the number of edges in our problem instance, since as long as the FF algorithm is choosing augmenting paths starting with (s, i), it is essentially performing a recursive maximum flow computation from node i to node t, in which augmenting paths through s are disallowed, and which will be terminated prematurely if it manages to accumulate a total amount of $i \rightsquigarrow t$ flow equal to the capacity of (s, i). Moreover, this recursive max flow computation is taking place on a graph that is smaller by one node, since its augmenting paths never contain s. By induction on the number of nodes in our instance, the recursive max flow computation therefore terminates in finite time.

Suppose now that our recursive max flow computation terminates due to condition (i). In this case, we will never use (s, i) again in any augmenting path, since to do so we would need to augment along the reverse residual edge (i, s), and an augmenting path will never utilize such an edge directed into the source. We can therefore effectively ignore (s, i) and (i, s) henceforth, and this gives us a smaller problem instance in which (by induction) the remainder of the FF algorithm will terminate in finite time. On the other hand, suppose condition (ii) occurs. In this case, we claim that i cannot appear on any future augmenting path, so again we can reduce the size of our instance by one node and claim by induction that the rest of the FF algorithm must terminate finitely. The fact that i cannot be part of any future augmenting path is argued as follows: at the point in time when condition (ii) occurs, let S_i be the set of all nodes that are reachable from i via a residual augmenting path that does not include s. All edges leaving S_i must be saturated except those directed into s (and those edges will never be part of any augmenting path). Therefore, in order for any node in S_i to ever again have an augmenting path to t (that doesn't use s), we would first need to augment along a path that enters S_i (to unsaturate one of the outgoing edges); however, such a path could never leave S_i.

3 The GS Algorithm with Irrational Data

In this section, we study the performance of the GS algorithm for the stable allocation problem in the presence of real-valued data.

3.1 The Stable Allocation Problem

The stable allocation problem is a many-to-many extension of the well-studied classical stable matching problem. Let $[n] := \{1, 2, \ldots, n\}$ denote a set of n jobs and $[m]$ denote a set of m machines (we will use scheduling terminology and speak of assigning "jobs" to "machines" since the traditional use of "men" and "women" for stable matching problems becomes somewhat awkward once we generalize to the many-to-many case). Job i requires p_i units of processing time, machine j has a capacity of c_j units, and at most $u_{ij} \leq \min(p_i, c_j)$ units of job i can be assigned to machine j. A fractional assignment $x \in \mathbf{R}^{m \times n}$ between jobs and machines is *feasible* if it satisfies

$$\begin{aligned} \sum_{j \in [m]} x_{ij} &\leq p_i \quad \forall i \in [n] \\ \sum_{i \in [n]} x_{ij} &\leq c_j \quad \forall j \in [m] \\ 0 \leq x_{ij} &\leq u_{ij} \quad \forall (i,j) \in [n] \times [m]. \end{aligned} \tag{1}$$

Just as in a stable matching problem, each job i submits a ranked preference list over all machines, and each machine j submits a ranked preference list over all jobs. Our goal is to compute a feasible assignment that is *stable* with respect to these preference lists, defined as follows.

Definition 1. *Job i and machine j form a* blocking pair *in an assignment x if $x_{ij} < u_{ij}$ and both i and j are partially assigned to partners they prefer less than each-other (in other words, both i and j would be "happier" if x_{ij} were increased).*

Definition 2. *A job i is* saturated *if $\sum_j x_{ij} \geq p_i$. A machine j is* saturated *if $\sum_i x_{ij} \geq c_j$.*

Definition 3. *In an assignment x, we say job i is* popular *if there exists a machine j such that $x_{ij} < u_{ij}$ and j prefers i to one of its current assignments. Similarly, machine j is popular if there exists some job i such that $x_{ij} < u_{ij}$ and i prefers j to one of its current assignments.*

Definition 4. *An assignment x is* stable *if (i) it admits no blocking pairs, and (ii) all popular jobs and machines are saturated.*

A feasible stable assignment x is said to be *job-optimal* (*machine-optimal*) if every job (machine) prefers x to any other feasible stable assignment x'.

As a final assumption, we assume that $\sum_i p_i \leq \sum_j c_j$. This comes without loss of generality by introducing a "dummy" machine of large capacity that is ranked last by every job.

3.2 A Stable Allocation Algorithm

Job-optimal and machine-optimal stable assignments always exist, and one can compute either one of them using a strongly-polynomial algorithm of Baiou and

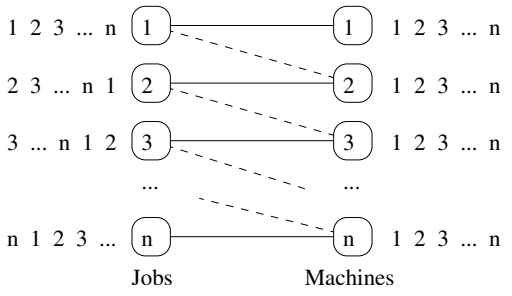


Fig. 3. A simple stable allocation instance where the GS algorithm runs much faster than the BB algorithm. Each job has unit processing time and each machine has unit capacity (so this is really an instance of the stable matching problem). Solid lines indicate the unique job-optimal stable assignment. Dashed lines indicate the unique job-optimal stable assignment when machine 1 is removed (job n ends up unassigned).

Balinski [1]. The Baiou-Balinski (BB) algorithm (say, applied to computing a job-optimal solution) operates by introducing one machine at a time. After each machine is introduced the current assignment is appropriately modified so as to remain job-optimal with respect to the current set of machines. Unfortunately, this might involve a significant amount of "thrashing" in some instances, since the introduction of each new machine can result in significant, or even wholesale change to the current assignment. An example is shown in Figure 3.

An alternative to the BB algorithm is the Gale-Shapley (GS) algorithm, familiar to anyone who has studied stable matchings in the past. The GS algorithm operates as follows. Each job i maintains a sorted list L_i of machines, initially equal to the preference list. Similarly, each machine j maintains an initially empty list R_j of the jobs currently assigned to j. Again, the list is sorted in order of the preference list. The algorithm iteratively performs the following steps until all jobs are saturated.

1. Select an unsaturated job i.
2. Let j be the first machine on list L_i.
3. Set $x_{ij} = \min(p_i - \sum_{j'} x_{ij'}, u_{ij})$ and add job i to list R_j of machine j making sure to maintain the sorted order. If $x_{ij} = u_{ij}$, then remove j from L_i.
4. While $\sum_{i'} x_{i'j} > c_j$, reject $p = \min(x_{i''j}, c_j - \sum_{i'} x_{i'j})$ processing time from the last job i'' in R_j (i.e., decrement $x_{i''j}$ by p). For any such rejected job i'', remove j from $L_{i''}$.

The GS algorithm can be viewed as an "augmenting path" algorithm just like the FF algorithm. In fact, when we apply the FF algorithm to a bipartite assignment problem, each augmenting path alternates back and forth between the left-hand and right-hand side of our graph, and we can interpret the meaning of such a path as a sequence of "proposals" and "rejections".

In many instances in practice, one expects the GS algorithm to run no slower and potentially much faster than the BB algorithm. For example, if our capacity

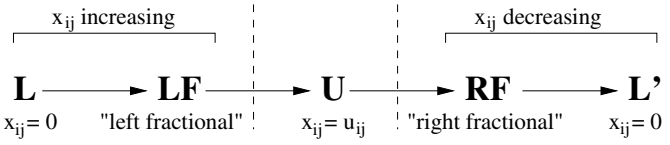


Fig. 4. The sequence of labelings through which an edge progresses

constraints are reasonably loose such that most jobs receive their first choices (a very common occurrence in practice), the GS algorithm terminates almost immediately, in time sublinear in the input size. The BB algorithm, on the other hand, might take substantially longer due to the need to continually readjust its solution as new machines are introduced. Figure 3 is a good example — the GS algorithm makes one proposal for each job, running in only $O(n)$ total time, and the BB algorithm reassigns each job at least $n-1$ times when the machines are added in the order $n, n-1, \ldots, 1$, for a total running time of $\Omega(n^2)$. While in the GS algorithm each job moves down its preference list proposing first to its must-preferred partner, the BB algorithm moves the opposite direction, potentially assigning a job to many machines as it moves up the list towards the most preferred stable partner. If capacity constraints are fairly loose and most jobs end up with highly-preferred partners, the GS algorithm tends to spend much less work.

3.3 Termination of the Gale-Shapley Algorithm

A technique for proving termination of graph-based algorithms is as follows. First, define a finite set of labels for edges and an ordering on this set of labels. Next, prove that during the course of the algorithm, the labeling of an edge can only advance according to this ordering and that each advancement is guaranteed to occur after a finite amount of time. In the rest of this section, we use this technique to prove termination of the GS algorithm.

Edge Labelings. After every iteration of the GS algorithm, we label the edges of our bipartite assignment graph as follows:

- The set L contains all edges (i, j) at their lower capacities ($x_{ij} = 0$) along which a proposal has never been issued. If $(i, j) \in L$, then job i has yet to reach all the way down to j on its preference list.
- The set LF contains all *left fractional* edges. An edge (i, j) is left fractional if $0 < x_{ij} < u_{ij}$ and if j has never issued a rejection to i. Among its two endpoints i and j, the left endpoint i was therefore the "responsible party" that prevented x_{ij} from growing any larger (i.e., j would have happily accepted more load, but i was reluctant so far to offer it).
- The set U contains all edges (i, j) at their upper capacities ($x_{ij} = u_{ij}$).
- The set RF contains all *right fractional* edges. An edge (i, j) is right fractional if $0 < x_{ij} < u_{ij}$ and if j has rejected i in the past. Among its two

endpoints i and j, the right endpoint j is the "responsible party" in this case for having prevented x_{ij} from growing any larger (i might want to send more load to j, but j refuses).
- The set L' contains all edges (i, j) at their lower capacities ($x_{ij} = 0$) along which a proposal has been issued in the past. If $(i, j) \in L'$, then job i has already exhausted its proposals to j and moved on down its preference list. Any assignment from i to j has since been completely rejected, and x_{ij} will never become positive again.

We call $F = LF \cup RF$ the set of all *fractional* edges, since their assignments lie strictly between their lower and upper bounds.

As we see in Figure 4, during the course of the GS algorithm an edge will progress monotonically through the labelings in the order L, LF, U, RF, and L'. It is possible that an edge skips over some of these labels — for example if an edge $(i, j) \in U$ is subject to a massive rejection by j it may become labeled L'. The monotonicity of this progression is significant. If we look at an edge (i, j) as the GS algorithm executes, the value of x_{ij} may increase as (i, j) moves from L to LF to U and then it may decrease if (i, j) further proceeds to RF and L'. By way of contrast, the value of x_{ij} can fluctuate up and down many times in the case of the FF algorithm for a "flow based" assignment problem.

Any time an edge changes its label, we say the edge is *promoted*. If we can prove that an iteration of the GS algorithm promotes an edge, then this indicates significant progress towards termination, since each edge can be promoted at most 4 times.

Observation 2. *If $(i, j) \in L$ and i proposes to j, then (i, j) will be promoted.*

Observation 3. *If $(i, j) \in L \cup LF \cup U$ and j issues a rejection to i, then (i, j) will be promoted.*

Properties of Edge Labelings. The structure of the GS algorithm becomes somewhat clearer when we look at properties of edge labelings that hold after each of its iterations.

Observation 4. *For every job i, $\delta_{LF}(i) \leq 1$. That is, i can have at most one outgoing left fractional edge (we use $\delta_S(i)$ to denote the degree of node i restricted to the subset S of edges).*

Recall in the GS algorithm, each job i maintains a pointer to the "current" machine j to which it is issuing proposals. Once j starts to reject i or x_{ij} reaches u_{ij}, this pointer advances down i's preference list. Therefore, among all the machines to which i has ever proposed, there is only at most one that may still be accepting proposals from i, and this corresponds to the LF edge emanating from i. We will have $(i, j) \in U \cup RF \cup L'$ for all other machines j to which i has proposed in the past.

Observation 5. *For every machine j, $\delta_{RF}(j) \leq 1$. That is, j can have at most one incoming right fractional edge.*

To see this, suppose j somehow had two incoming edges $(i, j) \in RF$ and $(i', j) \in RF$ where j prefers i to i'. This contradicts the behavior of the GS algorithm, since j would never have rejected i when it had the opportunity to reject i' instead.

Lemma 1. *Let G' be any connected subgraph of our assignment graph consisting only of fractional edges (LF and RF edges). Among the fractional edges in G' there can be at most one cycle.*

Proof. Let A denote the nodes of G' on the left hand side (the jobs) and B the nodes of G' on the right hand side (the machines). If $n = |A| + |B|$ denotes the total number of nodes in G' and m is the number of fractional edges, then $m = \sum_{i \in A} \delta_{LF}(i) + \sum_{j \in B} \delta_{RF}(j) \leq |A| + |B| = n$. So either $m = n - 1$, in which case the fractional edges in G' form a tree, or $m = n$, in which case the fractional edges in G' form a tree plus one additional edge that creates a unique cycle.

Observation 6. *Once a machine becomes saturated, it stays saturated forever. (The same is not true for jobs)*

Observation 7. *If $(i, j) \in RF \cup L'$, then machine j is saturated.*

Observation 8. *Let C be a cycle of fractional edges after some non-terminal iteration of the GS algorithm. Then (i) the edges along C alternate between being labeled LF and RF, and (ii) C cannot contain every node in our bipartite assignment graph.*

The alternating labels are a direct consequence of observations 4 and 5. As a result of each machine j in C having an incoming RF edge, we know that each machine j in C must be saturated. Therefore, it cannot be the case that C contains every machine, due to our assumption that $\sum_i p_i \leq \sum_j c_j$ (if every machine were saturated, then $\sum_i p_i$ units of total load would be assigned and the algorithm would have terminated)

Another observation we could make, but that is not crucial to our ensuing discussion, is that *if* the GS algorithm terminates, then it does so with no fractional cycles, since we could perform a rotation (see [4]) around such a cycle to make the assignment more job-optimal, and we know that the GS algorithm produces a job-optimal assignment. So we could strengthen the preceding observation to say that even in its terminal iteration, the GS algorithm cannot produce a fractional cycle containing all nodes.

Transient Configurations. We are now ready to prove the following:

Theorem 2. *The GS algorithm terminates after a finite number of iterations, even with irrational problem data.*

Our proof of the theorem rests on analyzing certain types of labeled subgraphs, which we call *transient configurations*, that are known to last only a finite number of iterations before they disappear and never return. As long as the GS

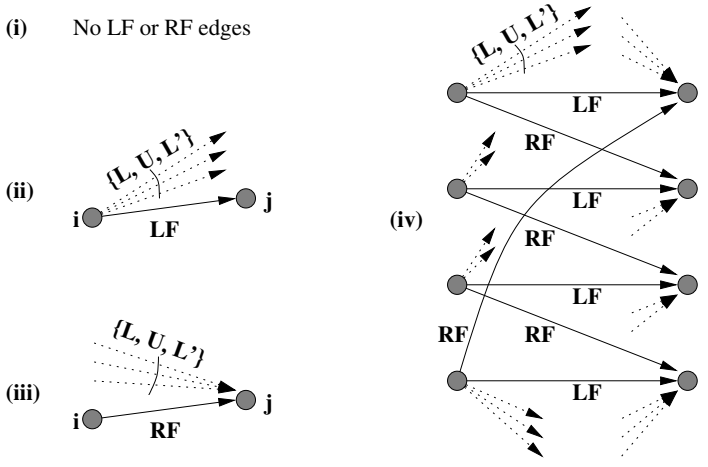


Fig. 5. Transient configurations. Solid edges represent fractional edges (LF or RF) and dotted edges represent edges labeled either L, U, or L'.

algorithm has not terminated, we argue that at least one of these configurations must be present, and since there are only finitely many such configurations, this implies that the GS algorithm must eventually terminate after a finite number of iterations.

Definition 5. *Consider the labeled edges in our bipartite assignment graph after some iteration of the GS algorithm. A subset of edges and their associated labels is called a* transient configuration *if after a finite number of iterations, either the GS algorithm must terminate or one of these edges must be promoted.*

For example, consider configuration type (i) listed in Figure 5, where simply every edge in our graph belongs to $L \cup U \cup L'$ (i.e., there are no fractional edges). This is a transient configuration because, if the GS algorithm has not terminated, its next proposal will utilize and promote one of the edges in L.

The remaining transient configuration types we consider are, as shown in Figure 5:

- Type (ii): The set of all edges emanating from a job i, where precisely one of these edges is in LF and the remaining are in $L \cup U \cup L'$.
- Type (iii): The set of all incoming edges to a machine j into which precisely one of these edges is in RF and the remaining are in $L \cup U \cup L'$.
- Type (iv): The set of all edges adjacent to the nodes in a fractional cycle, where edges on the cycle are in $LF \cup RF$ and all other edges are in $L \cup U \cup L'$.

Lemma 2. *After each iteration of the GS algorithm, we will find at least one configuration of type (i), (ii), (iii), or (iv).*

Proof. If there are no fractional edges then we have a configuration of type (i), so let us assume that either LF or RF edges exist. Consider only the subgraph of

our assignment containing edges in $LF \cup RF$. Due to our previous observations, we know that each connected component in this subgraph is either a tree or a tree plus one additional edge (forming a unique cycle). If we have a tree component, then at one of its leaf edges we must find a configuration of either type (ii) or (iii). If we have a cycle component, then either it is a "pure" cycle (which is a configuration of type (iv)), or it consists of a cycle with trees branching off it, and again in this case we must find configurations of type (ii) or (iii) at the leaf edges of these trees.

Lemma 3. *The type (iii) configuration is transient.*

Proof. Consider a machine j whose incoming edges currently form a type (iii) configuration. If subsequent iterations of the GS algorithm are to avoid promoting edges incoming to j, then there can be no proposals to j. The next proposal j receives must come from one of its incoming L edges (the others are in $RF \cup U \cup L'$ and hence are "spent" and will issue no further proposals). However, as we have observed, any proposal along an edge in L will promote the edge. Therefore, the set of edges incoming to j will retain their labeling for exactly as long as j receives no further proposals.

We argue using induction on the number of jobs and machines in our instance that the time until j must receive a proposal is finite. Let us form a strictly smaller problem instance by removing j and all its incident edges, and by subtracting x_{ij} from p_i for all machines i. Any sequence of iterations in our original instance with no proposals to j corresponds to an analogous sequence of proposals that is a valid GS sequence for the reduced problem instance. Since the reduced instance is strictly smaller, we know by applying Theorem 2 inductively that the GS algorithm terminates finitely on it. Hence, in the original problem instance we must encounter a proposal to j after a finite number of iterations, and this will promote one of the edges incoming to j.

Lemma 4. *The type (ii) configuration is transient.*

Proof. This proof is similar and essentially symmetric to the preceding proof. Consider a job i whose outgoing edges currently form a type (ii) configuration. We consider two cases, the first of which involves i being saturated. Here, i will not issue any proposals until some of its load is rejected, but any such rejection will result in the promotion of one of i's outgoing edges. We therefore want to know how long the GS algorithm can continue to operate before some machine rejects i, and as before we argue by induction that this amount of time must be finite. This is done by removing i from the instance and subtracting x_{ij} from c_j for each machine j, and (just as before) noting a correspondence between any sequence of proposals for the GS algorithm in the original instance where no machine rejects i, and a valid sequence of proposals for the GS algorithm on the reduced instance.

Next, we consider the case where i is not saturated. Here, we repeat the argument above twice: once to argue that a finite amount of time must elapse before i must propose along the edge $(i, j) \in LF$ (at which point it becomes

saturated), and again to argue as above that a finite amount of time must elapse before some machine rejects i.

Lemma 5. *The type (iv) configuration is transient.*

Proof. Consider a cycle C of fractional edges (alternating between LF and RF) that forms a configuration of type (iv), and recall that C does not include every node in the entire graph. When the jobs in C issue proposals, they will use their outgoing LF edges and hence propose "inside" the cycle rather than to machines outside the cycle. When such proposals occur, machines inside the cycle will issue rejections along their incoming RF edges (also within the cycle). This behavior will finally stop when the cycle "breaks" due to one its edges being promoted (for example, if one of its RF edges joins L'). We can therefore view the remaining iterations of the GS algorithm as consisting of 3 types of proposals: (a) proposals from jobs in C to machines in C, (b) proposals from jobs not in C to machines in C, and (c) proposals from jobs not in C to machines not in C. A proposal of type (b) will promote an edge, so we need to determine how long the GS algorithm can continue to run while only issuing proposals of types (a) and (c). For this purpose, we can consider the cycle and the rest of the instance as two independent entities, and since these are both strictly smaller than the main problem instance, we know by induction that the GS algorithm can spend only a finite amount of time in both cases.

The proof of Theorem 2 is now complete.

References

1. M. Baiou and M. Balinski. Erratum: The stable allocation (or ordinal transportation) problem. *Mathematics of Operations Research*, 27(4):662–680, 2002.
2. L.R. Ford and D.R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.
3. D. Gale and L.S. Shapley. College admissions and the stability of marriage. *American Mathematical Monthly*, 69(1):9–14, 1962.
4. D. Gusfield and R. Irving. *The Stable Marriage Problem: Structure and Algorithms*. MIT Press, 1989.
5. U. Zwick. The smallest networks on which the ford-fulkerson maximum flow procedure may fail to terminate. *Theoretical Computer Science*, 148:165–170, 1995.

Dynamic Programming and Fast Matrix Multiplication

Frederic Dorn*

Department of Informatics, University of Bergen, PO Box 7800, 5020 Bergen, Norway

Abstract. We give a novel general approach for solving NP-hard optimization problems that combines dynamic programming and fast matrix multiplication. The technique is based on reducing much of the computation involved to matrix multiplication. We show that our approach works faster than the usual dynamic programming solution for any vertex subset problem on graphs of bounded branchwidth. In particular, we obtain the fastest algorithms for Planar Independent Set of runtime $O(2^{2.52\sqrt{n}})$, for Planar Dominating Set of runtime exact $O(2^{3.99\sqrt{n}})$ and parameterized $O(2^{11.98\sqrt{k}}) \cdot n^{O(1)}$, and for Planar Hamiltonian Cycle of runtime $O(2^{5.58\sqrt{n}})$. The exponent of the running time is depending heavily on the running time of the fastest matrix multiplication algorithm that is currently $o(n^{2.376})$.

1 Introduction

Dynamic programming is a useful tool for the fastest algorithms solving NP-hard problems. We give a new technique for combining dynamic programming and matrix multiplication and apply this approach to problems like Dominating Set and Independent Set for improving the best algorithms on graphs of bounded treewidth.

Fast matrix multiplication gives the currently fastest algorithms for some of the most fundamental graph problems. The main algorithmic tool for solving the All Pair Shortest Paths problem for both directed and undirected graphs with small and large integer weights is to iteratively apply the distance product on the adjacency matrix of a graph [18],[20],[3],[25]. Next to the distance product, another variation of matrix multiplication—the boolean matrix multiplication—is solved via fast matrix multiplication. Boolean matrix multiplication is used to obtain the fastest algorithm for Recognizing Triangle-Free Graphs [16]. Recently,Vassilevska and Williams [23] applied the distance product to present the first truly sub-cubic algorithm for finding a Maximum Node-Weighted Triangle in directed and undirected graphs.

The fastest known matrix multiplication of two $n \times n$-matrices by Coppersmith and Winograd [6] in time $O(n^\omega)$ for $\omega < 2.376$ is also used for the fastest boolean matrix multiplication in same time. Rectangular matrix multiplication of an $(n \times p)$- and $(p \times n)$-matrix with $p < n$ gives the runtime $O(n^{1.85} \cdot p^{0.54})$. If

* Email: frederic.dorn@ii.uib.no. Supported by the Research Council of Norway.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 280–291, 2006.

$p > n$, we get time $O(p \cdot n^{\omega-1})$. The time complexity of the current algorithm for distance product is $O(n^3/\log n)$, but for integer entries less than m, where m is some small number, there is an $\widetilde{O}(mn^{\omega})$ algorithm [25]. For the arbitrarily weighted distance product no truly sub-cubic algorithm is known. Though, [23] show that the most significant bit of the distance product can be computed in sub-cubic time, and they conjecture that their method may be extended in order to compute the distance product.

Numerous problems are solved by matrix multiplication. However, for NP-hard problems the common approaches do not involve fast matrix multiplication. Williams [24] established new connections between fast matrix multiplication and hard problems. He reduces the instances of the well-known problems Max-2-SAT and Max-Cut to exponential size graphs dependent on some parameter k, arguing that the optimum weight k-clique corresponds to an optimum solution to the original problem instance.

The idea of applying fast matrix multiplication is basically to use the information stored in the adjacency matrix of a graph in order to fast detect special subgraphs such as shortest paths, small cliques—as in the previous example—or fixed sized induced subgraphs. Uncommonly—as in [24]—we do not use the technique on the graph directly. Instead, it facilitates a fast search in the solution space. In the literature, there has been some approaches speeding up linear programming using fast matrix multiplication, e.g. see [22]. For our problems, we consider dynamic programming, which is a method for reducing the runtime of algorithms exhibiting the properties of overlapping subproblems and optimal substructure. A standard approach for getting fast exact algorithms for NP-hard problems is to apply dynamic programming across subsets of the solution space. We present a novel approach to fast computing these subsets by applying the distance product on the structure of dynamic programming.

Many NP-complete graph problems turn out to be solvable in polynomial time or even linear time when restricted to the class of graphs of bounded treewidth. The tree decomposition detects how "tree-like" a graph is and the graph parameter treewidth is a measure of this "tree-likeness". The corresponding algorithms typically rely on a dynamic programming strategy. Telle and Proskurowski [21] gave an algorithm based on tree decompositions having width ℓ that computes the Dominating Set of a graph in time $O(9^{\ell}) \cdot n^{O(1)}$. Alber et al. [1] not only improved this bound to $O(4^{\ell}) \cdot n^{O(1)}$ by using several tricks, but also were the first to give a subexponential fixed parameter algorithm for Planar Dominating Set.

Recently there have been several papers [11, 4, 8, 12, 13], showing that for planar graphs or graphs of bounded genus the base of the exponent in the running time of these algorithms could be improved by instead doing dynamic programming along a branch decomposition of optimal branchwidth—both notions are closely related to tree decomposition and treewidth. Fomin and Thilikos [11] significantly improved the result of [1] for Planar Dominating Set to $O(2^{15.13\sqrt{k}}k + n^3)$ where k is the size of the solution. The same authors [13] achieve small constants in the running time of a branch decomposition based

exact algorithms for Planar Independent Set and Planar Dominating Set, namely $O(2^{3.182\sqrt{n}})$ and $O(2^{5.043\sqrt{n}})$, respectively. Dorn et al. [8] use the planar structure of sphere cut decompositions to obtain fast algorithms for problems like Planar Hamiltonian Cycle in time $O(2^{6.903\sqrt{n}})$. Dynamic programming along either a branch decomposition or a tree decomposition of a graph both share the property of traversing a tree bottom-up and combining tables of solutions to problems on certain subgraphs that overlap in a bounded-size separator of the original graph.

Our contribution. We introduce a new dynamic programming approach on branch decompositions. Instead of using tables, it stores the solutions in matrices that are computed via distance product. Since distance product is not known to have a fast matrix multiplication in general, we only consider unweighted and small integer weighted problems with weights of size $O(m) = n^{O(1)}$.

Our approach is fully general. It runs faster than the usual dynamic programming for any vertex subset problem on graphs of bounded branchwidth. It also can be used for tree decompositions with a structure proposed in [10]. To simplify matters, we first introduce our technique on the Independent Set problem on graphs of branchwidth bw and show the improvement from $O(2^{1.5\,\mathrm{bw}}) \cdot n^{O(1)}$ to $O(2^{\frac{\omega}{2}\,\mathrm{bw}}) \cdot n^{O(1)}$ where ω is the exponent of fast matrix multiplication (currently $\omega < 2.376$).

Next, we give the general technique and show how to apply it to several optimization problems such as Dominating Set, that we improve from $O(3^{1.5\,\mathrm{bw}}) \cdot n^{O(1)}$ to $O(4^{\mathrm{bw}}) \cdot n^{O(1)}$—please note that here ω influences the runtime indirectly. Finally, we show the significant improvement of the low constants of the runtime for the approach on planar graph problems. On Planar Dominating Set we reduce the time to even $O(2^{0.793\omega\,\mathrm{bw}}) \cdot n^{O(1)}$ and hence an improvement of the fixed parameter algorithm in [11] to $O(2^{11.98\sqrt{k}}) \cdot n^{O(1)}$ where k is the size of the dominating set. For exact subexponential algorithms as on Planar Independent Set and Planar Dominating Set, this means an improvement to $O(2^{1.06\omega\sqrt{n}})$ and $O(2^{1.679\omega\sqrt{n}})$, respectively. We also achieve an improvement for several variants in [2] and [10].

Since the treewidth tw and branchwidth bw of a graph satisfy the relation $\mathrm{bw} \leq \mathrm{tw}+1 \leq \frac{3}{2}\,\mathrm{bw}$, it is natural to formulate the following question as done in [10]: Given a tree decomposition and a branch decomposition, for which graphs is it better to use a tree decomposition based approach and for which is branch decomposition the appropriate tool? Table 1 compares our results to [10]. It illustrates that dynamic programming is almost always faster on branch decompositions when using fast matrix multiplication rather than dynamic programming on tree decompositions. For Planar Dominating Set it turns out that our approach is always the better one in comparison to [1], i.e., we achieve $O(3.688^{\mathrm{bw}}) < O(4^{\mathrm{tw}})$. For Planar Hamiltonian Cycle, we preprocess the matrices in order to apply our method using boolean matrix multiplication in time $O(2^{2.347\omega\sqrt{n}})$. In Table 1, we also add the runtimes for solving related problems and the runtime improvement compared to [8], [9], and [11], and [13].

Table 1. Worst-case runtime in the upper part expressed also by treewidth tw and branchwidth bw of the input graph. The problems marked with '*' are the only one where treewidth may be the better choice for some cutpoint $\mathrm{tw} \leq \alpha \cdot \mathrm{bw}$ with $\alpha = 1.19$ and 1.05 (compare with [10]). The lower part gives a summary of the most important improvements on exact and parameterized algorithms with parameter k. Note that we use the fast matrix multiplication constant $\omega < 2.376$.

	Previous results	New results
Dominating Set	$O(n2^{\min\{2\,\mathrm{tw}, 2.38\,\mathrm{bw}\}})$	$O(n2^{2\,\mathrm{bw}})$
Independent Set*	$O(n2^{\mathrm{tw}})$	$O(n2^{\min\{\mathrm{tw}, 1.19\,\mathrm{bw}\}})$
Independent Dominating Set	$O(n2^{\min\{2\,\mathrm{tw}, 2.38\,\mathrm{bw}\}})$	$O(n2^{2\,\mathrm{bw}})$
Perfect Code*	$O(n2^{\min\{2\,\mathrm{tw}, 2.58\,\mathrm{bw}\}})$	$O(n2^{\min\{2\,\mathrm{tw}, 2.09\,\mathrm{bw}\}})$
Perfect Dominating Set*	$O(n2^{\min\{2\,\mathrm{tw}, 2.58\,\mathrm{bw}\}})$	$O(n2^{\min\{2\,\mathrm{tw}, 2.09\,\mathrm{bw}\}})$
Maximum 2-Packing*	$O(n2^{\min\{2\,\mathrm{tw}, 2.58\,\mathrm{bw}\}})$	$O(n2^{\min\{2\,\mathrm{tw}, 2.09\,\mathrm{bw}\}})$
Total Dominating Set	$O(n2^{\min\{2.58\,\mathrm{tw}, 3\,\mathrm{bw}\}})$	$O(n2^{2.58\,\mathrm{bw}})$
Perfect Total Dom Set	$O(n2^{\min\{2.58\,\mathrm{tw}, 3.16\,\mathrm{bw}\}})$	$O(n2^{2.58\,\mathrm{bw}})$
Planar Dominating Set	$O(2^{5.04\sqrt{n}})$	$O(2^{3.99\sqrt{n}})$
Planar Independent Set	$O(2^{3.18\sqrt{n}})$	$O(2^{2.52\sqrt{n}})$
Planar Hamiltonian Cycle	$O(2^{6.9\sqrt{n}})$	$O(2^{5.58\sqrt{n}})$
Planar Graph TSP	$O(2^{9.86\sqrt{n}})$	$O(2^{8.15\sqrt{n}})$
Planar connected Dom Set	$O(2^{9.82\sqrt{n}})$	$O(2^{8.11\sqrt{n}})$
Planar Steiner Tree	$O(2^{8.49\sqrt{n}})$	$O(2^{7.16\sqrt{n}})$
Planar Feedback Vertex Set	$O(2^{9.26\sqrt{n}})$	$O(2^{7.56\sqrt{n}})$
Parameterized Planar Dom Set	$O(2^{15.13\sqrt{k}}k + n^3)$	$O(2^{11.98\sqrt{k}}k + n^3)$
Param Planar Longest Cycle	$O(2^{13.6\sqrt{k}}k + n^3)$	$O(2^{10.5\sqrt{k}}k + n^3)$

2 Definitions

Branch decompositions. A *branch decomposition* $\langle T, \mu\rangle$ of a graph G is a ternary tree T with a bijection μ from $E(G)$ to the leaf-set $L(T)$. For every $e \in E(T)$ define *middle set* $\mathrm{mid}(e) \subseteq V(G)$ as follows: For every two leaves ℓ_1, ℓ_2 with vertex v adjacent to both $\mu^{-1}(\ell_1)$ and $\mu^{-1}(\ell_2)$, we have that $v \in \mathrm{mid}(e)$ for all edges e along the path from ℓ_1 to ℓ_2. The *width* bw of $\langle T, \mu\rangle$ is the maximum order of the middle sets over all edges of T, i.e., $\mathrm{bw}(\langle T, \mu\rangle) := \max\{|\,\mathrm{mid}(e)| \colon e \in T\}$. An optimal branch decomposition of G is defined by the tree T and the bijection μ which together provide the minimum width, the *branchwidth* $\mathrm{bw}(G)$.

Dynamic programming. For a graph G with $|V(G)| = n$ of bounded branchwidth bw the weighted Independent Set problem with positive node weights w_v for all $v \in V(G)$ can be solved in time $O(f(\mathrm{bw})) \cdot n^{O(1)}$ where $f(\cdot)$ is an exponential time function only dependent on bw. The algorithm is based on dynamic programming on a rooted branch decomposition $\langle T, \mu\rangle$ of G. The independent set is computed by processing T in post-order from the leaves to the root. For each middle set $\mathrm{mid}(e)$ an optimal independent set intersects with some subset U of $\mathrm{mid}(e)$. Since $\mathrm{mid}(e)$ may have size up to bw, this may give 2^{bw} possible

subsets to consider. The separation property of $\mathrm{mid}(e)$ ensures that the problems in the different subtrees can be solved independently.

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 $\mathrm{mid}(e') = \mathrm{mid}(e'') = \mathrm{mid}(e)$. Create a new node *root* r, connect it to s and set $\mathrm{mid}(\{r, s\}) = \emptyset$. Each internal node v of T now has one adjacent edge on the path from v to r, called the *parent edge*, and two adjacent edges towards the leaves, called the *children edges*. To simplify matters, we call them the *left child* and the *right child*.

Let T_e be a subtree of T rooted at edge e. G_e is the subgraph of G induced by all leaves of T_e. For a subset U of $V(G)$ let $w(U)$ denote the total weight of nodes in U. That is, $w(U) = \sum_{u \in U} w_u$. Define a set of subproblems for each subtree T_e. Each set corresponds to a subset $U \subseteq \mathrm{mid}(e)$ that may represent the intersection of an optimal solution with $V(G_e)$. Thus, for each independent set $U \subseteq \mathrm{mid}(e)$, we denote by $\mathcal{V}_e(U)$ the maximum weight of an independent set S in G_e such that $S \cap \mathrm{mid}(e) = U$, that is $w(S) = \mathcal{V}_e(U)$. We set $\mathcal{V}_e(U) = -\infty$ if U is not an independent set since U cannot be part of an optimal solution. There are $2^{|\,\mathrm{mid}(e)|}$ possible subproblems associated with each edge e of T. Since T has $O(|E(G)|)$ edges, there are in total at most $2^{\mathrm{bw}} \cdot |E(G)|$ subproblems. The maximum weight independent set is determined by taking the maximum over all subproblems associated with the root r.

For each edge e the information needed to compute $\mathcal{V}_e(U)$ is already computed in the values for the subtrees. Since T is ternary, we have that a parent edge e has two children edges f and g. For f and g, we simply need to determine the value of the maximum-weight independent sets S_f of G_f and S_g of G_g, subject to the constraints that $S_f \cap \mathrm{mid}(e) = U \cap \mathrm{mid}(f)$, $S_g \cap \mathrm{mid}(e) = U \cap \mathrm{mid}(g)$ and $S_f \cap \mathrm{mid}(g) = S_g \cap \mathrm{mid}(f)$.

With independent sets $U_f \subseteq \mathrm{mid}(f)$ and $U_g \subseteq \mathrm{mid}(g)$ that are not necessarily optimal, the value $\mathcal{V}_e(U)$ is given as follows:

$$\begin{aligned}\mathcal{V}_e(U) = w(U) + \max\{&\mathcal{V}_f(U_f) - w(U_f \cap U) + \mathcal{V}_g(U_g) - w(U_g \cap U)\\ &- w(U_f \cap U_g \setminus U)\} \ \text{ s.t. } U_f \cap \mathrm{mid}(e) = U \cap \mathrm{mid}(f),\\ &U_g \cap \mathrm{mid}(e) = U \cap \mathrm{mid}(g), \text{ and } U_f \cap \mathrm{mid}(g) = U_g \cap \mathrm{mid}(f). \quad (1)\end{aligned}$$

The brute force approach computes for all $2^{|\,\mathrm{mid}(e)|}$ sets U associated with e the value $\mathcal{V}_e(U)$ in time $O(2^{|\,\mathrm{mid}(f)|} \cdot 2^{|\,\mathrm{mid}(g)|})$. Hence, the total time spent on edge e is $O(8^{\mathrm{bw}})$.

Matrix multiplication. Two $(n \times n)$-matrices can be multiplied using $O(n^\omega)$ algebraic operations, where the naive matrix multiplication shows $\omega \leq 3$. The best upper bound on ω is currently $\omega < 2.376$ [6].

For rectangular matrix multiplication between two $(n \times p)$- and $(p \times n)$-matrices $B = (b_{ij})$ and $C = (c_{ij})$ we differentiate between $p \leq n$ and $p > n$. For the case $p \leq n$ Coppersmith [5] gives an $O(n^{1.85} \cdot p^{0.54})$ time algorithm (under the assumption that $\omega = 2.376$). If $p > n$, we get $O(\frac{p}{n} \cdot n^{2.376} + \frac{p}{n} \cdot n^2)$ by matrix splitting: Split each matrix into $\frac{p}{n}$ many $n \times n$ matrices $B_1, \ldots, B_{\frac{p}{n}}$ and $C_1, \ldots, C_{\frac{p}{n}}$ and multiply each $A_\ell = B_\ell \cdot C_\ell$ (for all $1 \leq \ell \leq \frac{p}{n}$). Sum up each entry a_{ij}^ℓ overall matrices A_ℓ to obtain the solution.

The *distance product* of two $(n \times n)$-matrices B and C, denoted by $B \star C$, is an $(n \times n)$-matrix A such that

$$a_{ij} = \min_{1 \le k \le n} \{b_{ik} + c_{kj}\}, 1 \le i, j \le n. \tag{2}$$

The distance product of two $(n \times n)$-matrices can be computed naively in time $O(n^3)$. Zwick [25] describes a way of using fast matrix multiplication, and fast integer multiplication, to compute distance products of matrices whose elements are taken from the set $\{-m, \ldots, 0, \ldots, m\}$ The running time of the algorithm is $\widetilde{O}(m \cdot n^{\omega})$. For distance product of two $(n \times p)$- and $(p \times n)$-matrices with $p > n$ we get $\widetilde{O}(p \cdot (m \cdot n^{\omega-1}))$ again by matrix splitting: Here we take the minimum of the entries a_{ij}^{ℓ} overall matrices A_ℓ with $1 \le \ell \le \frac{p}{n}$.

3 Dynamic Programming and Distance Product

In this section, we will continue our Independent Set example and oppose two techniques on how to obtain faster dynamic programming approaches. The previous algorithms use tables in order to decrease the number of times a subset is queried. As a second approach, we introduce a technique using matrices that allows to highly make use of the structure of branch decompositions and of the fast matrix multiplication.

Tables. We will see now a more sophisticated approach that exploits properties of the middle sets and uses tables as data structure. With a table, one has an object that allows to store all sets $U \subseteq \mathrm{mid}(e)$ in an ordering such that the time used per edge is reduced to $O(2^{1.5\,\mathrm{bw}})$.

By the definition of middle sets, a vertex has to be in at least two of three middle sets of adjacent edges e, f, g. You may simply recall that a vertex has to be in all middle sets along the path between two leaves of T.

For the sake of a refined analysis, we partition the middle sets of parent edge e and left child f and right child g into four sets L, R, F, I as follows:

- *Intersection* $I := \mathrm{mid}(e) \cap \mathrm{mid}(f) \cap \mathrm{mid}(g)$,
- *Forget* $F := \mathrm{mid}(f) \cap \mathrm{mid}(g) \setminus I$,
- *Symmetric difference* $L := \mathrm{mid}(e) \cap \mathrm{mid}(f) \setminus I$ and $R := \mathrm{mid}(e) \cap \mathrm{mid}(g) \setminus I$.

We thus can restate the constraints of (1) for the computation of value $\mathcal{V}_e(U)$. Weight $w(U)$ is already contained in $w(U_f \cup U_g)$ since $\mathrm{mid}(e) \subseteq \mathrm{mid}(f) \cup \mathrm{mid}(g)$. Hence, we can change the objective function:

$$\begin{aligned} \mathcal{V}_e(U) \;=\; & \max\{\mathcal{V}_f(U_f) + \mathcal{V}_g(U_g) - w(U_f \cap U_g)\} \\ & \text{s.t. } U_f \cap (I \cup L) = U \cap (I \cup L),\ U_g \cap (I \cup R) = U \cap (I \cup R), \\ & \text{and } U_f \cap (I \cup F) = U_g \cap (I \cup F). \end{aligned} \tag{3}$$

Turning to tables, each edge e is assigned a table $Table_e$ that is labeled with the sequence of vertices $\mathrm{mid}(e)$. More precisely, the table is labeled with the concatenation of three sequences out of $\{L, R, I, F\}$. Define the concatenation

'$\|$' of two sequences λ_1 and λ_2 as $\lambda_1\|\lambda_2$. Then, concerning parent edge e and left child f and right child g we obtain the labels: '$I\|L\|R$' for $Table_e$, '$I\|L\|F$' for $Table_f$, and '$I\|R\|F$' for $Table_g$. $Table_f$ contains all sets U_f with value $\mathcal{V}_f(U_f)$ and analogously, $Table_g$ contains all sets U_g with value $\mathcal{V}_g(U_g)$.

For computing $\mathcal{V}_e(U)$ of each of the $2^{|I|+|L|+|R|}$ entries of $Table_e$, we thus only have to consider $2^{|F|}$ sets U_f and U_g subject to the constraints in (3). Since $\text{mid}(e)\cup\text{mid}(f)\cup\text{mid}(g) = I\cup L\cup R\cup F$, we have that $|I|+|L|+|R|+|F| \leq 1.5\cdot\text{bw}$. Thus we spend in total time $O(2^{1.5\,\text{bw}})$ on each edge of T.

A technical note: for achieving an efficient running time, one uses an adequate encoding of the table entries. First define a coloring $c : V(G) \to \{0,1\}$: For an edge e, each set $U \subseteq \text{mid}(e)$, if $v \in \text{mid}(e) \setminus U$ then $c(v) = 0$ else $c(v) = 1$. Then sort $Table_f$ and $Table_g$ to get entries in an increasing order in order to achieve a fast inquiry.

Matrices. In the remaining section we show how to use matrices instead of tables as data structure for dynamic programming. Then we apply the distance product of two matrices to compute the values $\mathcal{V}(U)$. With $U \cap I = U_f \cap I = U_g \cap I$, one may observe that every independent set S_e of G_e is determined by the independent sets S_f and S_g such that all three sets intersect in some subset $U^I \subseteq I$. The idea is to not compute $\mathcal{V}_e(U)$ for every subset U separately but to simultaneously calculate for each subset $U^I \subseteq I$ the values $\mathcal{V}_e(U)$ for all $U \subseteq \text{mid}(e)$ subject to the constraint that $U \cap I = U^I$. For each of these sets U the values $\mathcal{V}_e(U)$ are stored in a matrix A. A row is labeled with a subset $U^L \subseteq L$ and a column with a subset $U^R \subseteq R$. The entry determined by row U^L and column U^R is filled with $\mathcal{V}_e(U)$ for U subject to the constraints $U \cap L = U^L$, $U \cap R = U^R$, and $U \cap I = U^I$.

We will show how matrix A is computed by the distance product of the two matrices B and C assigned to the children edges f and g: For the left child f, a row of matrix B is labeled with $U^L \subseteq L$ and a column with $U^F \subseteq F$ that appoint the entry $\mathcal{V}_f(U_f)$ for U_f subject to the constraints $U_f \cap L = U^L$, $U_f \cap F = U^F$ and $U_f \cap I = U^I$. Analogously we fill the matrix C for the right child with values for all independent sets U_g with $U_g \cap I = U^I$. Now we label a row with $U^F \subseteq F$ and a column with $U^R \subseteq R$ storing value $\mathcal{V}_g(U_g)$ for U_g subject to the constraints $U_g \cap F = U^F$ and $U_g \cap R = U^R$. Note that entries have value '$-\infty$' if they are determined by two subsets where at least one set is not independent.

Lemma 1. *Given an independent set $U^I \subseteq I$. For all independent sets $U \subseteq \text{mid}(e)$, $U_f \subseteq \text{mid}(f)$ and $U_g \subseteq \text{mid}(g)$ subject to the constraint $U \cap I = U_f \cap I = U_g \cap I = U^I$ let the matrices B and C have entries $\mathcal{V}_f(U_f)$ and $\mathcal{V}_g(U_g)$. The entries $\mathcal{V}_e(U)$ of matrix A are computed by the distance product $A = B \star C$.*

Proof. The rows and columns of A, B and C must be ordered that two equal subsets stand at the same position, i.e., U^L must be at the same position in either row of A and B, U^R in either column of A and C, and U^F must be in the same position in the columns of B as in the rows of C. In order to apply the distance product of (2), we change the signs of each entry in B and C since we deal with a maximization rather than a minimization problem. Another difference

between (2) and (3) is the additional term $w(U_f \cap U_g)$. Since U_f and U_g only intersect in U^I and U^F, we substitute entry $\mathcal{V}_g(U_g)$ in C for $\mathcal{V}_g(U_g) - |U^I| - |U^F|$ and we get a new equation:

$$\begin{aligned}\mathcal{V}_e(U) \;=\; & \min\{-\mathcal{V}_f(U_f) - (\mathcal{V}_g(U_g) - |U^I| - |U^F|)\} \\ & \text{s.t. } U \cap I = U_f \cap I = U_g \cap I = U^I,\ U_f \cap L = U \cap L = U^L, \\ & \text{and } U_g \cap R = U \cap R = U^R, \text{ and } U_f \cap F = U_g \cap F = U^F\}. \qquad (4)\end{aligned}$$

Since we have for the worst case analysis that $|L| = |R|$ due to symmetry reason, we may assume that $|U^L| = |U^R|$ and thus A is a square matrix. Every value $\mathcal{V}_e(U)$ in matrix A can be calculated by the distance product of matrix B and C, i.e., by taking the minimum over all sums of entries in row U^L in B and column U^R in C.

Theorem 1. *Dynamic programming for the* Independent Set *problem on weights* $O(m) = n^{O(1)}$ *on graphs of branchwidth* bw *takes time* $\widetilde{O}(m \cdot 2^{\frac{\omega}{2}\cdot \mathrm{bw}})$ *with* ω *the exponent of the fastest matrix multiplication.*

Proof. For every U^I we compute the distance product of B and C with absolute integer values less than m. We show that, instead of a $O(2^{|L|+|R|+|F|+|I|})$ running time, dynamic programming takes time $\widetilde{O}(m \cdot 2^{(\omega-1)|L|} \cdot 2^{|F|} \cdot 2^{|I|})$. We need time $O(2^{|I|})$ for considering all subsets $U^I \subseteq I$. Under the assumption that $2^{|F|} \geq 2^{|L|}$ we get the running time for rectangular matrix multiplication: $\widetilde{O}(m \cdot \frac{2^{|F|}}{2^{|L|}} \cdot 2^{\omega|L|})$. If $2^{|F|} < 2^{|L|}$ we simply get $\widetilde{O}(m \cdot 2^{1.85|L|} \cdot 2^{0.54|F|})$ (for $\omega = 2.376$), so basically the same running time behavior. By the definition of the sets L, R, I, F we obtain four constraints:

- $|I| + |L| + |R| \leq \mathrm{bw}$, since $\mathrm{mid}(e) = I \cup L \cup R$,
- $|I| + |L| + |F| \leq \mathrm{bw}$, since $\mathrm{mid}(f) = I \cup L \cup F$,
- $|I| + |R| + |F| \leq \mathrm{bw}$, since $\mathrm{mid}(g) = I \cup R \cup F$, and
- $|I| + |L| + |R| + |F| \leq 1.5 \cdot \mathrm{bw}$, since $\mathrm{mid}(e) \cup \mathrm{mid}(f) \cup \mathrm{mid}(g) = I \cup L \cup R \cup F$.

When we maximize our objective function $\widetilde{O}(m \cdot 2^{(\omega-1)|L|} \cdot 2^{|F|} \cdot 2^{|I|})$ subject to these constraints, we get the claimed running time of $\widetilde{O}(m \cdot 2^{\frac{\omega}{2}\cdot \mathrm{bw}})$.

4 A General Technique

In this section we formulate the dynamic programming approach using distance product in a more general way than in the previous section in order to apply it to several optimization problems. In the literature these problems are often called *vertex-state* problems. That is, we have given an alphabet λ of vertex-states defined by the corresponding problem. E.g., for the considered Independent Set we have that the vertices in the graph have two states relating to an independent set U: state '1' means "element of U" and state '0' means "not an element of U". We define a coloring $c : V(G) \to \lambda$ and assign for an edge e of the branch decomposition $\langle T, \mu \rangle$ a color c to each vertex in $\mathrm{mid}(e)$. Given an ordering of

$\text{mid}(e)$, a sequence of vertex-states forms a string $S_e \in \lambda^{|\text{mid}(e)|}$. For a further details, please consult for example [10].

Recall the definition of concatenating two strings S_1 and S_2 as $S_1 \| S_2$. We then define the strings $S_x(\rho)$ with $\rho \in \{L, R, F, I\}$ of length $|\rho|$ as substrings of S_x with $x \in \{e, f, g\}$ with e parent edge, f left child and g right child. We set $S_e = S_e(I) \| S_e(L) \| S_e(R)$, $S_f = S_f(I) \| S_f(L) \| S_f(F)$ and $S_g = S_g(I) \| S_g(F) \| S_g(R)$. We say S_e is *formed* by the strings S_f and S_g if $S_e(\rho)$, $S_f(\rho)$ and $S_g(\rho)$ suffice some problem dependent constraints for some $\rho \in \{L, R, F, I\}$. For Independent Set we had in the previous section that S_e is formed by the strings S_f and S_g if $S_e(I) = S_f(I) = S_g(I)$, $S_e(L) = S_f(L)$, $S_e(R) = S_g(R)$ and $S_f(F) = S_g(F)$. For problems as Dominating Set it is sufficient to mention that "formed" is differently defined, see for example [10]. With the common dynamic programming approach of using tables, we get to proceed $c_1^{|L|} \cdot c_1^{|R|} \cdot c_2^{|F|} \cdot c_3^{|I|}$ update operations of polynomial time where c_1, c_2 and c_3 are small problem dependent constants. Actually, we consider $|\lambda|^{|L|} \cdot |\lambda|^{|F|} \cdot |\lambda|^{|I|}$ solutions of G_f and $|\lambda|^{|R|} \cdot |\lambda|^{|F|} \cdot |\lambda|^{|I|}$ solutions of G_g to obtain $|\lambda|^{|L|} \cdot |\lambda|^{|R|} \cdot |\lambda|^{|I|}$ solutions of G_e. In every considered problem, we have $c_1 \equiv |\lambda|$, $c_2, c_3 \leq |\lambda|^2$ and $c_1 \leq c_2, c_3$. We construct the matrices as follows: For the edges f and g we fix a string $S_f(I) \in \lambda^I$ and a string $S_g(I) \in \lambda^I$ such that $S_f(I)$ and $S_g(I)$ form a string $S_e(I) \in \lambda^I$. Recall the definition of value $\mathcal{V}_e$ as the maximum (minimum) weight of a solution class. We compute a matrix A with $c_1^{|L|}$ rows and $c_1^{|R|}$ columns and with entries $\mathcal{V}_e(S_e)$ for all strings S_e that contain $S_e(I)$. That is, we label monotonically increasing both the rows with strings $S_e(L)$ and the columns with strings $S_e(R)$ that determine the entry $\mathcal{V}_e(S_e)$ subject to the constraint $S_e = S_e(I) \| S_e(L) \| S_e(R)$.

Using the distance product, we compute matrix A from matrices B and C that are assigned to the child edges f and g, respectively. Matrix B is labeled monotonically increasing row-wise with strings $S_f(L)$ and column-wise with strings $S_f(F)$. That is, B has $c_1^{|L|}$ rows and $c_2^{|F|}$ columns. A column labeled with string $S_f(F)$ is duplicated depending on how often it contributes to forming the strings $S_e \supset S_e(I)$. The entry determined by $S_f(L)$ and $S_f(F)$ consists of the value $\mathcal{V}_f(S_f)$ subject to $S_f = S_f(I) \| S_f(L) \| S_f(F)$. Analogously, we compute for edge g the matrix C with $c_2^{|F|}$ rows and $c_1^{|R|}$ columns and with entries $\mathcal{V}_g(S_g)$ for all strings S_g that contain $S_g(I)$. We label the columns with strings $S_g(R)$ and rows with strings $S_g(F)$ with duplicates as for matrix B. However, we do not sort the rows by increasing labels. We order the rows such that the strings $S_g(F)$ and $S_f(F)$ match, where $S_g(F)$ is assigned to row k in C and $S_f(F)$ is assigned to column k in B. I.e., for all $S_f(L)$ and $S_g(R)$ we have that $S_f = S_f(I) \| S_f(L) \| S_f(F)$ and $S_g = S_g(I) \| S_g(F) \| S_g(R)$ form $S_e = S_e(I) \| S_e(L) \| S_e(R)$. The entry determined by $S_g(F)$ and $S_g(R)$ consists of the value $\mathcal{V}_g(S_g)$ subject to $S_g = S_g(I) \| S_g(F) \| S_g(R)$ minus an *overlap*. The overlap is the contribution of the vertex-states of the vertices of $S_g(F) \cap F$ and $S_g(I) \cap I$ to $\mathcal{V}_g(S_g)$. That is, the part of the value that is contributed by $S_g(F) \| S_g(R)$ is not counted since it is already counted in $\mathcal{V}_f(S_f)$.

Lemma 2. *Consider fixed strings $S_e(I)$, $S_f(I)$ and $S_g(I)$ such that there exist solutions $S_e \supset S_e(I)$ formed by some $S_f \supset S_f(I)$ and $S_g \supset S_g(I)$. The values $\mathcal{V}_f(S_f)$ and $\mathcal{V}_g(S_g)$ are stored in matrices B and C, respectively. Then the values $\mathcal{V}_e(S_e)$ of all possible solutions $S_e \supset S_e(I)$ are computed by the distance product of B and C, and are stored in matrix $A = B \star C$.*

The following theorem refers to all the problems enumerated in Table 1.

Theorem 2. *Let ω be the exponent of the fastest matrix multiplication and c_1, c_2 and c_3 the number of algebraic update operations for the sets $\{L, R\}$, F and I, respectively. Then, dynamic programming for solving vertex-state problems on weights $O(m) = n^{O(1)}$ on graphs of branchwidth* bw *takes time $\widetilde{O}(m \cdot \max\{c_1^{(\omega-1)\cdot\frac{\mathrm{bw}}{2}} c_2^{\frac{\mathrm{bw}}{2}}, c_2^{\mathrm{bw}}, c_3^{\mathrm{bw}}\})$.*

5 Application of the New Technique

In this section, we show how one can apply the technique for several optimization problems such as Dominating Set and its variants in order to obtain fast algorithms. We also apply our technique to planar graph problems. The branchwidth of a planar graph is bounded by $2.122\sqrt{n}$. There exist optimal branch decompositions whose middle sets are closed Jordan curves in the planar graph embedding [8]. Such a *sphere cut decomposition* has the property that the I-set is of size at most 2, that is, the runtime stated in Theorem 2 has no part 'c_3^{bw}'.

For Dominating Set we have that $c_1 \equiv c_2 = 3$ and $c_3 = 4$. The former running time was $O(3^{1.5\,\mathrm{bw}}) \cdot n^{O(1)}$. We have $\widetilde{O}(m \cdot \max\{3^{(\omega-1)\cdot\frac{\mathrm{bw}}{2}} 3^{\frac{\mathrm{bw}}{2}}, 3^{\mathrm{bw}}, 4^{\mathrm{bw}}\}) = \widetilde{O}(m \cdot 4^{\mathrm{bw}})$ for node weights $O(m)$ if we use a matrix multiplication algorithm with $\omega < 2.5$ and thus hide the factor ω.

Sphere cut decompositions of planar graphs can be computed in time $O(n^3)$ by an improvement of the famous rat catcher method ([19] and [14]). With the nice property that $|I| \leq 2$ for all middle sets, we achieve a running time in terms of $\widetilde{O}(m \cdot \max\{c_1^{(\omega-1)\cdot\frac{\mathrm{bw}}{2}} c_2^{\frac{\mathrm{bw}}{2}}, c_2^{\mathrm{bw}}\})$ for planar graph problems. Thus, we improve for Planar Dominating Set with node weights $O(m)$ the runtime $O(4^{\mathrm{bw}}) \cdot n^{O(1)}$ to $\widetilde{O}(m \cdot 3^{1.188\,\mathrm{bw}}) = \widetilde{O}(m \cdot 3.688^{\mathrm{bw}})$. This runtime is strictly better than the actual runtime of the treewidth based technique of $O(4^{\mathrm{tw}}) \cdot n^{O(1)}$.

For Planar Hamiltonian Cycle, it is not immediately clear how to use matrices since here it seems necessary to compute the entire solution at a dynamic programming step. I.e., in [8] the usual dynamic programming step is applied with the difference that a postprocessing step uncovers forbidden solutions and changes the coloring of the vertices in the L- and R-set. The idea that helps is that we replace the latter step by a preprocessing step, changing the matrix entries of the child edges depending on the change of the coloring. That coloring is only dependent on the coloring of the F-set in both matrices. Hence we do not query the coloring of all three sets L, R and F simultaneously. This means that this step does not increase the runtime of our algorithm that is improved to $\widetilde{O}(m \cdot 2^{1.106\omega\,\mathrm{bw}})$ by applying boolean matrix multiplication.

6 Conclusions

We established a combination of dynamic programming and fast matrix multiplication as an important tool for finding fast exact algorithms for NP-hard problems. Even though the currently best constant $\omega < 2.376$ of fast matrix multiplication is of rather theoretical interest, there exist indeed some practical sub-cubic runtime algorithms that help improving the runtime for solving all mentioned problems. An interesting side-effect of our technique is that any improvement on the constant ω has a direct effect on the runtime behavior for solving the considered problems. E.g., for Planar Dominating Set; under the assumption that $\omega = 2$, we come to the point where the constant in the computation is 3 what equals the number of vertex states, which is the natural lower bound for dynamic programming. Currently, [23] have made some conjecture on an improvement for distance product, which would enable us to apply our approach to optimization problems with arbitrary weights. Is there anything to win for dynamic programming if we use 3-dimensional matrices as a data structure? That is, if we have the third dimension labeled with $S_e(I)$?

Acknowledgments. Many thanks to Fedor Fomin for his useful comments and his patience, Artem Pyatkin for some fruitful discussions, and an anonymous referee for his comments, and Charis Papadopoulos, and Laura Toma.

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 and R. Niedermeier, *Improved tree decomposition based algorithms for domination-like problems*, in LATIN'02: Theoretical informatics (Cancun), vol. 2286 of Lecture Notes in Computer Science, Berlin, 2002, Springer, pp. 613–627.
3. N. Alon, Z. Galil, and O. Margalit, *On the exponent of the all pairs shortest path problem*, Journal of Computer and System Sciences, 54 (1997), pp. 255–262.
4. W. Cook and P. Seymour, *Tour merging via branch-decomposition*, INFORMS Journal on Computing, 15 (2003), pp. 233–248.
5. D. Coppersmith, *Rectangular matrix multiplication revisited*, Journal of Complexity, 13 (1997), pp. 42–49.
6. D. Coppersmith and S. Winograd, *Matrix multiplication via arithmetic progressions*, Journal of Symbolic Computation, 9 (1990), pp. 251–280.
7. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Second Edition*, The MIT Press and McGraw-Hill Book Company, 2001.
8. F. Dorn, E. Penninkx, H. Bodlaender, and F. V. Fomin, *Efficient exact algorithms on planar graphs: Exploiting sphere cut branch decompositions*, in Proceedings of the 13th Annual European Symposium on Algorithms (ESA 2005), vol. 3669 of LNCS, Springer, Berlin, 2005, pp. 95–106.

9. ———, *Efficient exact algorithms on planar graphs: Exploiting sphere cut decompositions*, 2006. manuscript, http://archive.cs.uu.nl/pub/RUU/CS/techreps/CS-2006/2006-006.pdf.
10. F. Dorn and J. A. Telle, *Two birds with one stone: the best of branchwidth and treewidth with one algorithm*, in LATIN'06: 7th Latin American Theoretical Informatics Symposium (Valdivia), vol. 3887 of Lecture Notes in Computer Science, Berlin, 2006, Springer, pp. 386–397.
11. F. V. Fomin and D. M. Thilikos, *Dominating sets in planar graphs: branch-width and exponential speed-up*, in SODA'03: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (Baltimore, MD, 2003), New York, 2003, ACM, pp. 168–177.
12. F. V. Fomin and D. M. Thilikos, *Fast parameterized algorithms for graphs on surfaces: Linear kernel and exponential speed-up*, in Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP 2004), vol. 3142 of LNCS, Berlin, 2004, Springer, pp. 581–592.
13. ———, *A simple and fast approach for solving problems on planar graphs*, in Proceedings of the 21st International Symposium on Theoretical Aspects of Computer Science (STACS 2004), vol. 2996 of LNCS, Springer, Berlin, 2004, pp. 56–67.
14. Q.-P. Gu and H. Tamaki, *Optimal branch-decomposition of planar graphs in $O(n^3)$ time*, in Proceedings of the 32nd International Colloquium on Automata, Languages and Programming (ICALP 2005), vol. 3580 of LNCS, Springer, Berlin, 2005, pp. 373–384.
15. P. Heggernes, J. A. Telle, and Y. Villanger, *Computing minimal triangulations in time $O(n^\alpha log n) = o(n^{2.376})$*, SIAM Journal on Discrete Mathematics, 19 (2005), pp. 900–913.
16. A. Itai and M. Rodeh, *Finding a minimum circuit in a graph*, SIAM Journal on Computing, 7 (1978), pp. 413–423.
17. D. Kratsch and J. Spinrad, *Between $O(nm)$ and $O(n^\alpha)$*, in SODA'03: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (Baltimore, MD, 2003), New York, 2003, ACM, pp. 158–167.
18. R. Seidel, *On the all-pairs-shortest-path problem in unweighted undirected graphs*, Journal of Computer and System Sciences, 51 (1995), pp. 400–403.
19. P. D. Seymour and R. Thomas, *Call routing and the ratcatcher*, Combinatorica, 14 (1994), pp. 217–241.
20. A. Shoshan and U. Zwick, *All pairs shortest paths in undirected graphs with integer weights*, in 40th Annual Symposium on Foundations of Computer Science, (FOCS '99), Lecture Notes in Computer Science, Springer, 1999, pp. 605–615.
21. J. A. Telle and A. Proskurowski, *Algorithms for vertex partitioning problems on partial k-trees*, SIAM J. Discrete Math, 10 (1997), pp. 529–550.
22. P. M. Vaidya, *Speeding-up linear programming using fast matrix multiplication*, in 30th Annual Symposium on Foundations of Computer Science (FOCS 1989), 1989, pp. 332–337.
23. V. Vassilevska and R. Williams, *Finding a maximum weight triangle in $n^{(3-\delta)}$ time, with applications*, 2006. To appear in ACM Symposium on Theory of Computing (STOC 2006), http://www.cs.cmu.edu/ ryanw/max-weight-triangle.pdf.
24. R. Williams, *A new algorithm for optimal constraint satisfaction and its implications*, in Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP 2004), vol. 3142 of LNCS, Springer, Berlin, 2004, pp. 1227–1237.
25. U. Zwick, *All pairs shortest paths using bridging sets and rectangular matrix multiplication*, Journal of the ACM, 49 (2002), pp. 289–317.

Near-Entropy Hotlink Assignments

Karim Douïeb* and Stefan Langerman**

Département d'Informatique, Université Libre de Bruxelles, Belgique
{kdouieb, stefan.langerman}@ulb.ac.be

Abstract. Consider a rooted tree T of arbitrary maximum degree d representing a collection of n web pages connected via a set of links, all reachable from a source home page represented by the root of T. Each web page i carries a weight w_i representative of the frequency with which it is visited. By adding hotlinks — shortcuts from a node to one of its descendents — we wish to minimize the expected number of steps l needed to visit pages from the home page, expressed as a function of the entropy $H(p)$ of the access probabilities p. This paper introduces several new strategies for effectively assigning hotlinks in a tree. For assigning exactly one hotlink per node, our method guarantees an upper bound on l of $1.141H(p)+1$ if $d > 2$ and $1.08H(p)+2/3$ if $d = 2$. We also present the first efficient general methods for assigning at most k hotlinks per node in trees of arbitrary maximum degree, achieving bounds on l of at most $\frac{2H(p)}{\log(k+1)}$ and $\frac{H(p)}{\log(k+d)-\log d}$, respectively. Finally, we present an algorithm implementing these methods in $O(n \log n)$ time, an improvement over the previous $O(n^2)$ time algorithms.

1 Introduction

There are many ways to speed up the access to information on the Web. The solution discussed in this paper doesn't change the original hyperlink structure but enhances it with additional hyperlinks in order to speed up the access to a destination. This addition of hyperlinks is called a *hotlink assignment*. The problem of the *hotlink assignment* was originally introduced by Perkowitz and Etzioni [13] to improve the search in Web sites.

The *hotlinks* are defined as additional pointers to a structure with the goal of improving its design by reducing the expected number of steps to reach an element. A hotlink can be seen as a shortcut from a web page to another one that is accessible from it (see Fig. 1.b).

The problem: Formally, a *web site* can be modeled as a directed graph $\mathcal{G} = (V, E)$ where the nodes V correspond to the web pages and the edges E represent the links. Each node carries a weight representative of its access frequency. We assume that all web pages are reached starting from the *homepage* r. Our goal in adding hotlinks (one or up to k directed edges from a node to one accessible

* Boursier FRIA.
** Chercheur qualifié du FNRS.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 292–303, 2006.

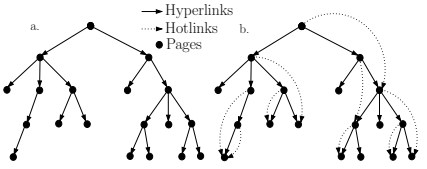


Fig. 1. a. Modeled site web, b. Example of hotlink assignment

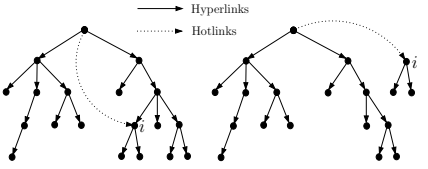


Fig. 2. Consequence due to the *greedy user* model assumption

from it) is to minimize the expected number of steps to reach a page from the homepage r.

We restrict our attention to the case when $\mathcal{G}$ is a rooted directed tree T with n nodes and maximum degree d (maximum number of children for a node). The stated results extend to general graphs by taking T to be the shortest-path tree of $\mathcal{G}$ from the homepage r. Every leaf i in T is associated with a weight w_i representative of its access frequency, and $W = \sum_{i \in T} w_i$. We thus assume that only the leaves of the tree are accessed. This restriction can easily be removed by adding a leaf child to all nodes, with a weight corresponding to the access frequency of the node. This transformation only increases the length of the search paths by 1. We use T_x to denote the subtree rooted at x and $W(T_x)$ to denote its weight, i.e. the sum of the weights of its leaves.

Following the *greedy user* model assumption [9], we assume that from a node the user always takes the pointer that leads him as close as possible to the desired destination. Due to that assumption, the assignment of one hotlink which points to a node i can be seen as the deletion of the other hyperlink that ends in i (i.e. an adoption) because if the user doesn't follow the hotlink then he will not access this subtree (see Fig. 2).

Let T^A be the tree resulting from an assignment A of hotlinks. A measure of the average access time to the nodes is $E[T^A, p] = \sum_{i=1}^{n} d_A(i) p_i$, where $d_A(i)$ is the distance of the node i from the root, and $p = \langle p_i = w_i/W : i = 1, \ldots, n \rangle$ is the probability distribution on the nodes of the original tree T. We are interested in finding an assignment A which minimizes $E[T^A, p]$.

A lower bound on the average access time $E[T^A, p]$ was given in [2] using information theory [12]. Let $H(p)$ be the entropy of the probability distribution p, defined by $H(p) = \sum_{i=1}^{n} p_i \log(1/p_i)$, then for any assignment of at most k hotlinks per node the expected number of steps to reach a node from the root of a tree of maximum degree d is at least $H(p)/\log(d+k)$ in the best case. The tree could be a list, in which case we have a lower bound of $H(p)/\log(1+k)$.

We focus on recursive algorithms which first choose the hotlink(s) of the root of the tree T, perform the adoption (see Fig. 2), and recursively assign hotlinks to the children of the root (including the hotlink). We characterize these algorithms as *top-down* if the hotlink assignment of a subtree only depends on the subtree itself minus the subtrees adopted by its own ancestors.

Related work: The idea of hotlinks was suggested by Perkowitz and Etzioni [13] to improve the search in Web sites (seen as DAGs). Later Bose et al. [2] proved

that finding the optimal hotlink assignment for a DAG is NP-hard, and analyzed several heuristics for assigning hotlinks.

The problem might become easier when the graph considered is a rooted tree. Kranakis, Krizanc and Shende [11] give a $O(n^2)$ time algorithm for assigning one hotlink per node so that the expected number of steps to search a node from the root of the tree attains the entropy bound within a constant factor. Several results on adding hotlinks to nodes of d-regular complete trees are also reported by Fuhrmann et al. [8]. Recently, Gerstel et al.[9], and A.A. Pessoa et al. [14] independently discovered a polynomial time dynamic programming algorithms for finding the optimal placement of hotlinks on a tree whose depth is logarithmic in the number of nodes, the running time of the algorithm of Gerstel et al. is $O(n3^D)$ where D is the height of the tree. Experimental results showing the validity of the hotlinks approach are given in [5], and a software tool to structure websites efficiently by automatic assignment of hotlinks has been developed [10].

The concept of hotlinks can be applied to other problems than that of web structuring. For instance, Bose et al.[3] use hotlink assignments to design efficient asymmetric communication protocols. Hotlinks can also be used to design data structures as was demonstrated by Brönnimann, Cazals and Durand [4] with their *jumplist* dynamic dictionary data structure. The jumplist structure can be seen as randomized hotlink assignment on a list, and is meant as a simplification of the skiplist structure [15]. A deterministic version of the randomized jumplist was developed by Elmasry [7] and by Douïeb and Langerman [6], independently.

Using this deterministic jumplist, we recently introduced a linear time algorithm [6] to allow the assignment of one hotlink per node in such a way that the number of steps to reach a node i from the root of a tree is bounded by the entropy, namely by $(3+\epsilon)H(p)$ for any $\epsilon > 0$. The method was then dynamized to maintain hotlinks when nodes are added, deleted or their weights modified, in amortized time $O(\log W/w_i)$ per update.

Our results: Known exact algorithms [9, 14] for finding the optimal assignment of hotlinks have a polynomial running time only for trees of logarithmic depth, and are slow, so our work was focused on finding an assignment approaching the entropy bound. The best previous algorithm, the KKS method [11], guarantees that the average access time to the elements is at most $\frac{H(p)}{\log(d+1)-(d/(d+1))\log d} + \frac{d+1}{d}$, its asymptotic behavior is $H(p)\frac{d}{\log d}$ for sufficiently large values of d (maximum degree of the tree). The running time of this algorithm is $O(n^2)$ where n is the number of elements in the tree.

After showing some preliminary lemmas in the next section, a new top-down method for assigning one hotlink per node is presented in Section 3. The h/p_h method guarantees an average access time of at most $1.141H(p)+1$. This near-entropy bound, in contrast to that of KKS, is completely independent of the maximum degree of the tree and is better than KKS for all values of $d > 2$. Furthermore, h/p_h method matches the bound of KKS for $d = 2$.

In Section 4, we present a natural generalization of the algorithm of Bose *et al.* [3] for assigning k hotlinks per node of trees of arbitrary maximum degree d instead of binary trees, it guarantees an upper bound on the average access time of $\frac{H(p)}{\log(k+d)-\log d}$. As the performance guarantee of this method degrades when d grows, we show a second method whose average access cost is at most $\frac{2H(p)}{\log(k+1)}$ constituting the first multiple hotlink assignment method giving a near-entropy bound that is independent of the degree.

Finally in the Section 5 we develop a fast algorithm for the methods seen in the preceding section; it uses an enhanced version of the link-cut trees of Sleator and Tarjan [16] and performs the hotlink assignment in $O(n \log n)$ time for all our methods and the KKS method [11]. This is an improvement over the previous $O(n^2)$ algorithms. Omitted proofs appear in the full version.

2 Top-Down Methods

Before giving some hotlinks assignment methods and their analysis we present a useful Lemma concerning entropy. Consider a probability distribution $p = \langle p_1, p_2, \ldots, p_n \rangle$ and a partition $A_1, A_2, \ldots, A_k$ of the index set $\{1, 2, \ldots, n\}$ into k non-empty subsets. Define $S_i = \sum_{j \in A_i} p_j$ for $i = 1, 2, \ldots, k$. Consider the new distributions: $p^{(i)} = \langle p_j^{(i)} := \frac{p_j}{S_i} : j \in A_i \rangle$ for $i = 1, 2, \ldots, k$. Kranakis, Krizanc and Shende [11] proved the following lemma:

Lemma 1. *For any partition $A_1, A_2, \ldots, A_k$ of the index set of the probability distribution we have the identity $H(p) = \sum_{i=1}^{k} S_i H(p^{(i)}) - \sum_{i=1}^{k} S_i \log S_i$, where S_i and $p^{(i)}$ are defined in the above equations.*

A hotlink method $\mathcal{A}$ determines the hotlink assignment $A = \mathcal{A}(T)$ to be applied on any tree T. Let T^A be the tree T enhanced by the hotlink assignment A and $T^{\mathcal{A}} = T^{\mathcal{A}(T)}$. Consider that a selection of successive hotlinks starting from the root node partitions the leaves of the tree $T^{\mathcal{A}}$ into several subsets or subtrees $T_1^{\mathcal{A}}, T_2^{\mathcal{A}}, \ldots, T_k^{\mathcal{A}}$ with corresponding weights $S_1, S_2, \ldots, S_k$. These subtrees have a depth in the tree corresponding to the number of pointers that we must follow to reach them, called $d(T_i^{\mathcal{A}})$.

We defined a *top-down* hotlink assignment method $\mathcal{A}$ to be a method beginning by the assignment of the hotlink of the root of a tree and where the hotlink assignment of any subtree T_i only depends on the subtree itself minus the subtrees adopted by its own ancestors.

Lemma 2. *Given a top-down hotlink assignment method $\mathcal{A}$, if we can fix a constant a such that for all tree T there exists a partition in the subtrees $T_1^{\mathcal{A}}, T_2^{\mathcal{A}}, \ldots, T_k^{\mathcal{A}}$ of weights $S_1, S_2, \ldots, S_k$ which satisfies $a \geq -\frac{\sum_{i=1}^{k} S_i d(T_i^{\mathcal{A}})}{\sum_{i=1}^{k} S_i \log S_i}$, then the expected number of steps needed to reach a leaf from the root of a tree $T^{\mathcal{A}}$ is $E[T^{\mathcal{A}}, p] \leq aH(p) + 1$.*

Finally we generalize Lemma 5 of [11]:

Lemma 3. *For any fixed constant $0 \leq \alpha \leq 1/2$, the solutions of the optimization problem maximize $f(s_1, s_2, \ldots, s_k) = \sum_{i=1}^{k} s_i \log s_i$ subject to $\{0 \leq s_i \forall i, \sum_{i=1}^{k} s_i = 1, \alpha \leq s_k \leq 1 - \alpha\}$ are obtained, when $s_k = \alpha$ and one among the quantities $s_1, s_2, \ldots, s_{k-1}$ attains the value $1 - \alpha$ and all the rest are equal to 0.*

3 Single Hotlink Assignment: h/ph Method

The KKS method [11] assigns one hotlink per node for trees with a constant maximum degree d. It is a top-down method which simply chooses as hotlink of the root of the tree a node h defining a subtree T_h of weight satisfying $\frac{W(T)}{(d+1)} \leq W(T_h) \leq \frac{dW(T)}{(d+1)}$.

The running time of this algorithm is quadratic in the number of vertices of the tree and assigns for any probability distribution $p = \langle p_1, p_2, \ldots, p_n \rangle$ on the n leaves of a tree one hotlink per node such a way that the expected number of steps to reach a leaf of the tree from the root is at most $\frac{H(p)}{\log(d+1)-(d/(d+1))\log d} + \frac{d+1}{d}$ (see [11]). This bound is asymptotically tight for the KKS method, and is achieved by a caterpillar with uniform distribution on its leaves.

The problem of this method is that its average access time degrades as the maximum degree d of the trees considered grows. To avoid this increase in the expected number of steps to reach a leaf from the root of a tree, we introduce a new method in the next section. The h/p_h *Method* :

The idea of this hotlink assignment method remains the same, the difference lies on the number of candidate nodes that we consider for the choice of the hotlink of the root of a subtree T. Namely, the candidates are firstly the node h of weight $w_{1/2}$ the nearest to $W(T)/2$ and secondly the parent node of h, denoted p_h. The method that determines how to choose among those candidates is called the h/p_h *method*: Let α be the unique solution of $\frac{\alpha}{1-\alpha} = \alpha^{\frac{1}{2(1-\alpha)}}$ (i.e. $\alpha \approx 0.2965$), if the weight $w_{1/2}$ of the node h is greater than the threshold α we take h as the hotlink of the root and we take p_h otherwise.

Before beginning the analysis of the method, we give a property of the nodes of weight $w_{1/2}$. Define the heavy path of a tree to be the path from the root to a leaf such that each node on the path is the heaviest child of its parent.

Lemma 4. *If $w_{1/2}$ is the weight nearest to $W(T)/2$ among all nodes in the subtree T, then there is a node of weight $w_{1/2}$ on its heavy path.*

We can now begin to analyze the expected access time to reach a leaf from the root of a tree after the hotlink assignment according to the h/p_hmethod.

Theorem 1. *Consider a tree T of arbitrary maximum degree and T^A the same tree after the hotlink assignment of the h/p_h method. The maximum average access time to the leaves of T^A is at most $\frac{H(p)}{\log 3-(2/3)} + 2/3$ if $d = 2$ and $H(p)\frac{2}{\log 1/\alpha} + 1 \approx 1.141H(p) + 1$ if $d > 2$.*

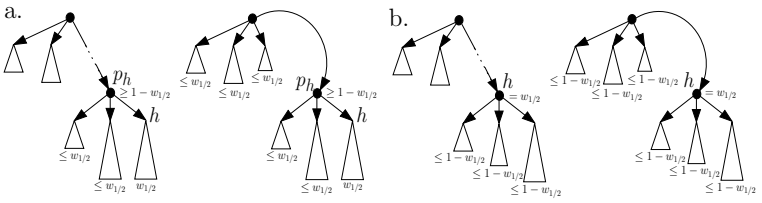


Fig. 3. Before and after assigning the hotlink of the root to the node p_h (a.) or h (b.)

Proof. We can make an analysis of the worst average access time by selecting 3 ranges for the value of $w_{1/2}$:

1. $0 \leq w_{1/2} < \alpha$, we are in the case where we must choose p_h as hotlink of the root. We know that the node h defines a subtree of weight $w_{1/2}$ nearest to $1/2$, thus the weight of the subtree defined by its parent node p_h is greater than $1 - w_{1/2}$ and the brother nodes of h define subtrees of weight smaller than $w_{1/2}$. After the assignment of the hotlink of the root to the node p_h, we know that none of the direct children of the root can have a weight greater than $1 - W(T_{p_h}) \leq w_{1/2}$.

That guarantees that after two steps of search from the root of the tree after the hotlink assignment we can not reach a subtree of weight greater than $w_{1/2} \leq \alpha$ (see Fig. 3.a). Using the notation of Lemma 2, we can express the worst expected number of steps to reach a leaf from the root of the tree in the case where we choose the p_h node as hotlink of the root in the current range: $E[T^A, p] \leq aH(p) + 1$ with $a \geq -\frac{2}{\log \alpha}$.

2. $\alpha \leq w_{1/2} < 1 - \alpha$, we are in a range where we must choose the node h as hotlink of the root. Note that the children $\{c_1, c_2, \ldots, c_k\}$ of the root of T^A other then h have a weight of at most $(1 - w_{1/2})$ and the node h has weight $w_{1/2}$. All subtrees of the root have a depth of 1. This partition gives a worst average access time to the leaves equal to $E[T^A, p] \leq aH(p) + 1$ with $a \geq -\frac{1}{(w_{1/2}) \log(w_{1/2}) + \sum_{i=1}^{k} (W(T_{c_i})) \log W(T_{c_i})}$ (see Lemma 2). The maximum value of this this last function subject to the constraints $\{\alpha \leq w_{1/2} \leq 1 - \alpha, \sum_{i=1}^{k} W(T_{c_i}) = 1 - w_{1/2}\}$ is given by Lemma 3, i.e. when $w_{1/2} = \alpha$, $W(T_{c_1}) = 1 - \alpha$ and $W(T_{c_i}) = 0$ for all $2 \leq i \leq k$. Thus the maximum expected number of steps to reach a leaf in this current range is $\frac{H(p)}{-(\alpha \log \alpha + (1-\alpha) \log(1-\alpha))} + 1$.

3. $1 - \alpha \leq w_{1/2} \leq 1$, we choose h as hotlink of the root. The node h defines a subtree of weight $w_{1/2}$ nearest to $1/2$, thus its heaviest child has a weight smaller than $1 - w_{1/2} \leq \alpha$, and the weight of the direct children of the root of the tree after the hotlink assignment can not exceed $1 - w_{1/2} \leq \alpha$ (see Fig. 3.b). By those facts, we know that none of the subtrees reachable after two steps of search can have a weight greater than α. That is exactly the same situation as in the first case where $0 \leq w_{1/2} \leq \alpha$, thus the worst average access time to the leaves will be the same, i.e. $-\frac{2H(p)}{\log \alpha} + 1$.

We saw that if $\alpha \leq w_{1/2} \leq 1-\alpha$ then the worst access time is equal to $\frac{H(p)}{-(\alpha \log \alpha + (1-\alpha)\log(1-\alpha))} + 1$, and for any other value of $w_{1/2}$ we have $-\frac{2H(p)}{\log \alpha} + 1$. Thus we can compute the value of α for which both expressions are equal, i.e. for which value of α the choice of h or p_h is equivalent. This occurs when $\frac{\alpha}{1-\alpha} = \alpha^{\frac{1}{2(1-\alpha)}}$ (i.e. $\alpha \approx 0.2965$), and the maximum expected number of steps needed to reach a leaf from the root of a tree T^A is no more than $H(p)\frac{2}{\log \frac{1}{\alpha}} + 1 \approx 1.141H(p) + 1$.

Thus for any tree with a maximum degree $d > 2$, the h/p_h method gives a better ratio for the approximation of the optimum hotlink assignment than the *KKS* method. But we can remark that if $d = 2$ then the h/p_h method cannot be worse than the *KKS* method. Indeed, in this case the value of $w_{1/2}$ is bounded above by $1/3$ and below by $2/3$, that implies that the h/p_h method always chooses the node h as hotlink of the root. This choice will be better or at least equivalent to the choice of the *KKS* hotlink assignment. So the h/p_h method is better in all the cases. □

4 Multiple Hotlink Assignment

In the preceding sections we saw the hotlink assignment problem in the case where just one hotlink per node of a tree is allowed. Now we consider the addition of k hotlinks for each node. Some studies have already been done on this topic, namely S. Fuhrmann *et al.* [8] present algorithms to reduce the height of a tree by a constant factor. The algorithms for optimal hotlink assignment by dynamic programming allow k hotlinks assignments per node [9, 14]. The KKS method [11] has been generalized by Bose *et al.* [3] to assign k hotlinks per node, but is restricted to binary trees, it guarantees an average access time at most $\frac{H(p)}{\log(k+2)-1} + 1$.

We introduce in this paper a recursive top-down method which performs up to k hotlink assignments, seen as adoptions, to the root of a tree T of arbitrary degree d to obtain an enhanced tree T'. Then the procedure is iterated for each child of the root in T'. This method is a natural generalization of the algorithm of Bose *et al.* [3] for trees of arbitrary maximum degree. When processing a node x, we perform hotlink assignments of the node x until each original child y of x is either a leaf or its weight satisfies $W(T_y) \leq dW(T_x)/(k+d)$. To determine which descendant z to assign next for a hotlink of the node x, we start at the non-leaf original heaviest child of x and we traverse its heavy path until reaching the node z of maximum weight smaller than $dW(T_x)/(k+d)$.

Thus all the hotlink nodes h_i of x have a weight greater than $W(T_x)/(k+d)$ implying that at most k hotlinks can be assigned by node, indeed the original non-leaf children of x after k assignments cannot have a weight greater than $W(T_x) - kW(T_x)/(k+d) = dW(T_x)/(k+d)$ which is the condition to stop.

Theorem 2. *Consider a tree T of maximum degree d and T^A the same tree after the hotlink assignment of the generalized method of [3]. The maximum average access time to the leaves of T^A is at most $\frac{H(p)}{\log(k+d)-\log d}$.*

The performances of this generalized method degrades as d grows. The next method avoids this dependence on d. Here, we perform hotlink assignments for the node x until each original child y of x is either a leaf or its weight satisfies $W(T_y) \leq W(T_x)/(k+1)$. To determine which descendant z to assign next for a hotlink of x, we start at the non-leaf original heaviest child of x and we traverse its heavy path until reaching the node z of minimum weight greater than $W(T_x)/(k+1)$. While processing a node x, at most k hotlinks are assigned. Indeed the assignment stops when each original child of x is either a leaf or its weight is smaller than $W(T_x)/(k+1)$. After k hotlink assignments, a non-leaf child of the node x cannot have a weight greater than $W(T_x) - kW(T_x)/(k+1) = W(T_x)/(k+1)$.

Theorem 3. *Consider a tree T of maximum degree d and T^A the same tree after the hotlink assignment of the above multiple hotlink assignment method. The maximum average access time to the leaves of T^A is at most $2H(p)/\log(k+1)$ for $d > \sqrt{k+1}$, and $H(p)/(\log(k+1) - \log d)$ otherwise.*

5 Fast Hotlink Assignment Algorithm

In order to perform the hotlink assignment according to the methods introduced previously, a naive $O(n^2)$ running time algorithm can be easily found. Here we present an $O(n \log n)$ running time algorithm which uses an enhanced version of the *Link-Cut Trees*.

The *Link-Cut Trees* or *ST Trees* of D.D.Sleator and R.E.Tarjan is a data structure for the *Dynamic trees problem* [16]. Namely, we are given a collection of vertex-disjoint rooted trees. We want to represent the trees by a data structure that allows us to easily extract certain informations (the cost of an edge, the minimum cost on a precise path, the parent of an node, the root of an node) about the trees and to easily update the structure to reflect changes in the trees caused by these two kinds of operations: *link* the root of a tree to any node of an other tree making this node the parent of the root, and *cut* a tree into two trees by deleting the edge from a selected node to its parent.

They develop a solution to the dynamic trees problem by using an implicit representation of the forest, which sees dynamic trees as sets of *solid paths* connected together with *dashed* edges (see Fig. 5.a). Each tree operation is carried out by means of one or more path operations. These dynamic solid paths are represented as biased binary trees(BBT) [1] (or splay trees [17]) whose external nodes correspond to the vertices of the solid paths and internal nodes correspond to subpaths (see Fig. 5.b). This data structure guarantees that each dynamic tree operation takes $O(\log n)$ time in the worst-case but only if the partition in solid paths is done by size (number of leaves inside the tree defined by a node), i.e. if the *solid paths* are defined to be the paths from the root of a subtree to a leaf where each node is the child of its parents which has the greatest size.

The remainder of this section modifies the Link-Cut tree structure, we refer the reader to [16] for more details.

Enhanced Link-Cut trees: Remember that the weight $W(T_v)$ of a vertex v is the sum of the weight of its children in the original tree T, where each leaf i in T is associated with a weight w_i representative of its access frequency.

Now we shall see how to enhance the Link-Cut Trees to use it for the hotlink assignment. First we add an extra part to the structure. For each vertex v on a solid path, we maintain a vertex set containing the children of v excepted the child corresponding to the next vertex $next[v]$ on the solid path of v (if it exists). We allow three kinds of operations on the vertex sets: (1) *maxw*(**vertex** v), return the vertex of maximum weight in the vertex set of v; return **null** if the vertex set is empty. (2) *insert*(**vertex** u,**vertex** v), insert vertex u into the vertex set of v. (3) *delete*(**vertex** u,**vertex** v), delete vertex u from the vertex set of v. We represent the vertex set of a vertex by a globally biased binary tree [1], the vertices appearing as external nodes, exactly as for the structures used in the original Link-Cut trees.

Finally we add one more field to each internal node or leaf x in the associated BBT of the solid paths: If the node x is a leaf of a BBT then the value wt_x is set to the sum of the weights of its children in the original tree excepted its next vertex on the solid path. Else the node x is an internal node of the BBT and wt_x is set to the sum of the value wt of its children in the BBT. We note that this information can be updated in a constant time after any rotation operation.

If the node x in the original tree T is contained in the solid path S, then the weight $W(T_x)$ can be computed with the value wt stored in the nodes of the BBT associated to the solid path S, i.e. $W(T_x) = \sum_{i \in R(x)} wt_i$ where $R(x)$ represent the right siblings (set of right child of nodes) on the path to the root of the BBT associated to S.

Note that these two extra structures, i.e. for the vertex sets and the values wt, are nearly identical to some structures present in the original Link-Cut trees, used to maintain the solid paths and to compute the size (in number of nodes) of the subtree of a node. Thus the added structures will be updated using the same techniques, achieving the same performances.

Search: Consider now the hotlink assignment and see how to use the enhanced Link-Cut tree to perform the search of the candidate node which will be pointed to by one of the k hotlinks of the root of the original tree. For all methods presented here, this candidate node will be found from a node h which defines a subtree of minimum weight greater than $W(T)/c$ for any fixed constant $c \geq 1$ depending on the method used. We can deduce from Lemma 4 or from the method itself that this node h is always located on the *heavy path*, this heavy path is defined as the path from the root of T to a leaf connecting each node on the path to its heaviest child. But in the Link-Cut tree, the decomposition of the initial tree is done by *solid paths* (decomposition by size), thus the heavy path could traverse several solid paths

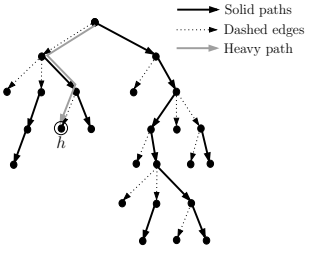


Fig. 4. The heavy path could intersect several solid paths

(see Fig. 4). The search of the node h is thus a succession of searches in multiple BBTs each associated to a solid path which intersects the heavy path.

The search is performed as follows: we begin from the BBT associated with the solid path containing the root of the original tree T, and use it to locate the lowest vertex v greater than $W(T)/c$. In order to perform this search efficiently we use the information wt stored in the nodes of this associated BBT. We walk down the BBT from its root r and we maintain a value $Z = \sum_{i \in R(j)} wt_i$ where j is the current node, i.e. Z is equal to the sum of the value wt of the right siblings of nodes on the path from the current node j to the root r. Thus $Z + wt_j$ is the maximum weight of any leaf reachable from the node j. We initially start from the root r and we set $Z = 0$. If $Z + wt_{right[r]} \geq W(T)/c$ we go down by the right child $rigth[r]$ of the root else we go by the left child and we update $Z = Z + wt_{right[r]}$. We iterate the process until we find the vertex v on the solid path of minimum weight greater than $W(T)/c$. Note that the value Z is equal to $W(T_{next[v]})$ when the node v is found. An illustration of this search is shown in Fig. 5.b.

If the the weight of the next vertex of v in its solid path is greater than $maxw(v)$, i.e. if $W(T_{next[v]}) \geq maxw(v)$ then v corresponds to the node h that we are looking for. Else we must check if the node h is present in the next solid path beginning by the vertex of weight $maxw(v)$. For that, we perform the same search in the associated BBT of this next solid path. We iterate the process until we find the node h.

According to the Link-Cut tree performance, we can find a node i contained in an associated BBT rooted at r in $O(\log \frac{Size(r)}{Size(i)})$ time, corresponding to the height of the BBT. The sum of the running times of the successive searches in the different solid paths is $\log \frac{Size(T)}{Size(x_1)} + \log \frac{Size(x_1)}{Size(x_2)} + \cdots + \log \frac{Size(x_k)}{Size(h)} \leq \log \frac{Size(T)}{Size(h)} \leq \log n$, where $x_1, x_2, \ldots, x_k$ are vertices leading to the successive solid paths traversed by a search. We must add to that the number of times that we use $maxw()$ for a vertex set to check if the node h is deeper in the tree (takes $O(1)$ time), this number is bounded by $\log n$ because of the definition of the solid path. Thus the maximum total running time needed to find the node h which gives the necessary information to find the candidate for the hotlink assignment of the root is $O(\log n)$.

Cut: To perform the hotlink assignment, we just need the *Cut* operation which consists in cuting a tree T into two trees by deleting the edge from a selected node to its parent. The cut operation with an enhanced Link-Cut tree is done as in the original structure excepted that the extra structures (vertex sets and fields wt) have to be updated.

Cutting a subtree rooted at a node h consists first in making an *expose* operation on the node h. That operation creates a single solid path, ending in h and beginning at the root of the original tree T, by converting *dashed edges* (connecting two distinct solid paths) to *solid* (connecting two vertices of the same solid path) along the tree path from h to the root of the original tree T and converting solid edges incident to this path to dashed.

Those kinds of edge conversions may change the vertex sets associated to several vertices of the original tree T. The dashed edges converted in solid must be deleted from the corresponding vertex set and respectively the solid edges converted in dashed must be inserted in their corresponding vertex set. The value wt of the nodes is also affected by those changes and have to be updated following the Link-Cut tree methods.

After the expose of the node h, we have to cut the subtree rooted at h and update the values of some nodes in the associated BBT of the solid path containing h, i.e. all nodes i where we walk to their right child during a search for h in the associated BBT have to update their value wt_i to $wt_i - W(T_h)$. Once this is done, we restructure the associated BBT and we repair the damage caused by the expose. Namely after the cut, the decomposition into solid paths could have changed and we must update the structure by an operation which can be seen as an expose running backwards. This operation is fully described in [16].

Thus the cut in a enhanced Link-Cut tree has the same asymptotic running time than in the original structure, i.e. each cut operation takes $O(\log n)$ time, where n is the number of nodes in the initial tree.

Lemma 5. *The hotlink assignment of a tree T according to the methods described in the previous sections can be done in $O(n \log n)$ time using the enhanced Link-Cut trees data structure seen above.*

Proof. Consider that we use an enhanced Link-Cut tree data structure as described above for a tree T. The hotlink assignment consists in finding a node h for one hotlink of the root according to the desired method. We have seen above that this search is performed in $O(\log n)$ time (Fig. 5.b). Once h is found, we cut the edge between h and its parent. This cut takes $O(\log n)$ time using the enhanced Link-Cut trees. For the multiple assignment methods we carry out the same operation as long as necessary. Once all the hotinks of the root has been assigned we cut all the edges connecting the root to its children, those cuts are done in $O(\log n)$ (Fig. 5.c), thus we obtain at most $d + k$ subtrees for which we iterate the same process recursively (Fig. 5.d).

Although each node could have up to k hotlinks, the total number of hotlinks assigned is smaller than n because there cannot be more than one hotlink point-

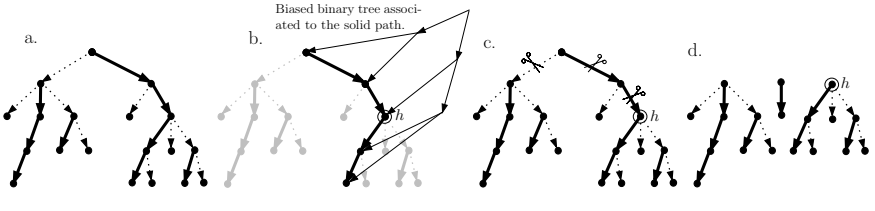


Fig. 5. a. Decomposition of a tree T in solid paths. b. A search in the biased binary tree representing the solid path. c. Cut of the node h and the children of the root. d. Resulting trees.

ing to each node. Thus the enhanced link-cut trees allow to perform a hotlink assignment for a node in $O(\log n)$ time, this must be done at most n times which implies that the entire hotlink assignment takes $O(n \log n)$ time. □

References

1. S. W. Bent, D. D. Sleator, and R. E. Tarjan. Biased seach trees. *SIAM J. Comput.*, 14 , number 3:545–568, 1985.
2. P. Bose, E. Kranakis, D. Krizanc, M. V. Martin, J. Czyzowicz, A. Pelc, and L. Gasieniec. Strategies for hotlink assignments. In *Proc. 11th Ann. Int. Symp. on Algorithms and Computation*, volume 1969 of LNCS, pages 23–34, 2000.
3. P. Bose, D. Krizanc, S. Langerman, and P. Morin. Asymmetric communication protocols via hotlink assignments. In *Proc. 9th Int. Coll. on Structural Information and Communication Complexity (SIROCCO 2002)*, pages 33–40, 2002.
4. H. Brönnimann, F. Cazals, and M. Durand. Randomized jumplists : A jump-and-walk dictionary data structure. *Proc. 20th Ann. Symp. on Theoretical Aspects of Computer Science (STACS 2003)*, 2607 of LNCS, 2003.
5. J. Czyzowicz, E. Kranakis, D. Krizanc, A. Pelc, and M. Martin. Evaluation of hotlink assignment heuristics for improving web access. In *Proc. 2nd Int. Conf. on Internet Computing (IC'2001)*, pages 793–799, 2001.
6. K. Douïeb and S. Langerman. Dynamic hotlinks. In *Proc. of the Workshop on Algorithms and Data Structures (WADS 2005)*, volume 3608 of LNCS, pages 271–280, 2005.
7. A. Elmasry. Deterministic jumplists. *Nordic Journal of Computing*, 12:27–39, 2005.
8. S. Fuhrmann, S. O. Krumke, and H.-C. Wirth. Multiple hotlink assignment. In *27th Int. Workshop on Graph-Theoric Concepts in Computer Science*, volume 2204 of LNCS, pages 189–200, 2001.
9. O. Gerstel, S. Kutten, R. Matichin, and D. Peleg. Hotlink enhancement algorithms for web directories. In *Proc. 14th Ann. Int. Symp. on Algorithms and Computation*, volume 2906 of LNCS, pages 68–77, 2003.
10. E. Kranakis, D. Krizanc, and M. V. Martin. The hotlink optimizer. In *Proc. 3rd Int.Conf. on Internet Computing (IC'2002)*, pages 33–40, 2002.
11. E. Kranakis, D. Krizanc, and S. Shende. Approximate hotlink assignment. In *Proc. 12th Ann. Int. Symp. on Algorithms and Computation*, volume 2223 of LNCS, pages 756–767, 2001.
12. N.Abramson. Information theory and coding. *McGraw Hill*, 1963.
13. M. Perkowitz and O. Etzioni. Towards adaptive Web sites: conceptual framework and case study. *Computer Networks*, 31(11-16):1245–1258, 1999.
14. A. Pessoa, E. Laber, and C. de Souza. Efficient algorithms for the hotlink assignment problem: The worst case search. In *Proc. 15th Ann. Int. Symp. on Algorithms and Computation*, volume 3341 of LNCS, page 778, 2004.
15. W. Pugh. Skip lists: a probabilistic alternative to balanced trees. In F. Dehne, J.-R. Sack, and N. Santoro, editors, *Proc. Workshop on Algorithms and Data Structures*, volume 382 of LNCS, pages 437–449, 1989.
16. D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–381, 1983.
17. D. D. Sleator and R. E. Tarjan. Self-adjusting binary trees. *Proc. 15th Ann. ACM Symp. on Theory of Computing*, pages 235–245, 1983.

Subspace Sampling and Relative-Error Matrix Approximation: Column-Row-Based Methods

Petros Drineas[1], Michael W. Mahoney[2,⋆], and S. Muthukrishnan[3]

[1] Department of Computer Science, RPI
[2] Yahoo Research Labs
[3] Department of Computer Science, Rutgers University

Abstract. Much recent work in the theoretical computer science, linear algebra, and machine learning has considered matrix decompositions of the following form: given an $m \times n$ matrix A, decompose it as a product of three matrices, C, U, and R, where C consists of a small number of columns of A, R consists of a small number of rows of A, and U is a small carefully constructed matrix that guarantees that the product CUR is "close" to A. Applications of such decompositions include the computation of matrix "sketches", speeding up kernel-based statistical learning, preserving sparsity in low-rank matrix representation, and improved interpretability of data analysis methods. Our main result is a randomized, polynomial algorithm which, given as input an $m \times n$ matrix A, returns as output matrices C, U, R such that

$$\|A - CUR\|_F \leq (1+\epsilon)\,\|A - A_k\|_F$$

with probability at least $1-\delta$. Here, A_k is the "best" rank-k approximation (provided by truncating the Singular Value Decomposition of A), and $\|X\|_F$ is the Frobenius norm of the matrix X. The number of columns in C and rows in R is a low-degree polynomial in k, $1/\epsilon$, and $\log(1/\delta)$. Our main result is obtained by an extension of our recent relative error approximation algorithm for ℓ_2 regression from overconstrained problems to general ℓ_2 regression problems. Our algorithm is simple, and it takes time of the order of the time needed to compute the top k right singular vectors of A. In addition, it samples the columns and rows of A via the method of "subspace sampling," so-named since the sampling probabilities depend on the lengths of the rows of the top singular vectors, and since they ensure that we capture entirely a certain subspace of interest.

1 Introduction

1.1 Motivation and Overview

Recent work in the theoretical computer science, linear algebra, and machine learning has considered matrix decompositions of the following form: given an $m \times n$ matrix A, decompose it as a product of three matrices, C, U, and R, where

⋆ Part of this work was done while at the Department of Mathematics, Yale University.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 304–314, 2006.

C consists of a few columns of A, R consists of a few rows of A, and U is a small, carefully constructed matrix that guarantees that the product CUR is "close" to A. Applications of such decompositions include constructing "sketches" for large matrices in a pass-efficient manner, matrix reconstruction, speeding up kernel-based statistical learning computations, sparsity-preservation in low-rank approximations, and improved interpretability of data analysis methods. See [6, 7, 21, 12, 11, 19, 20, 1] for examples of applications for matrix decompositions of this form.

Let us consider the application to data analysis methods in more detail. In many applications, the data are represented by a real $m \times n$ matrix A. Such a matrix may arise if the data consist of n objects, each of which is described by m features. The most common compressed representation of A used by data analysts is that obtained by truncating the Singular Value Decomposition at some number $k \ll \min\{m, n\}$ terms, in large part because this provides the "best" rank-k approximation to A when measured with respect to any unitarily invariant matrix norm. However, there is a fundamental difficulty with this representation: the new "dimensions" (the so-called eigencolumns and eigenrows) of A_k are linear combinations of the original dimensions. As such, they are notoriously difficult to interpret in terms of the underlying data and processes generating that data. For example, the vector $[(1/2)$ age - $(1/\sqrt{2})$ height + $(1/2)$ income], being one of the significant uncorrelated "factors" from a dataset of people's features is not particularly informative. From an analyst's point of view, it would be highly preferable to have a low-rank approximation that is nearly as good as that provided by the SVD but that is expressed in terms of a small number of *actual columns* and *actual rows* of a matrix, rather than linear combinations of those columns and rows.

For example, consider recent data analysis work in DNA microarray and DNA Single Nucleotide Polymorphism (SNP) analysis [14, 15, 17]. Researchers interested in analyzing DNA SNP data often model the data as an $m \times n$ matrix A, where m is the number of individuals in the study, n is the number of SNPs being analyzed, and A_{ij} is an encoding of the i-th SNP value for the j-th indivudual. Since biologists do not have an understanding or intuition about the behavior of, e.g., $30,000$ genes or $1,000,000$ SNPs or 1000 individuals, that they do have about a single gene or a single SNP or a single individual, linear algebraic methods have been employed to extract *actual SNPs* from the computed *eigen-SNPs* in order to be used for further analysis [14, 15, 17]. Our problem of approximating a matrix A by CUR is a direct formulation of this problem; in particular, our problem will determine a small number of actual SNPs to serve as a basis to express the remaining SNPs, and a small number of individuals to serve as a basis to express the remaining individuals.

1.2 Review of Linear Algebra

Let $[n]$ denote the set $\{1, 2, \ldots, n\}$. For any matrix $A \in \mathbb{R}^{m \times n}$, let $A_{(i)}, i \in [m]$ denote the i-th row of A as a row vector, and let $A^{(j)}, j \in [n]$ denote the j-th column of A as a column vector. The Singular Value Decomposition (SVD) of

A will be denoted by $A = U\Sigma V^T$, where $U \in \mathbb{R}^{m\times\rho}$, $\Sigma \in \mathbb{R}^{\rho\times\rho}$, $V \in \mathbb{R}^{n\times\rho}$, and where ρ is the rank of A. The "best" rank-k approximation to A (with respect to, e.g., the Frobenius norm, $||A||_F = \sqrt{\sum_{i,j} A_{ij}^2}$) will be denoted by $A_k = U_k\Sigma_k V_k^T$, where $U_k \in \mathbb{R}^{m\times k}$ is the first k columns of U, etc. The SVD and hence the best rank-k approximation of a general matrix A can be computed in $O(\min\{n^2m, nm^2\})$ time, and optimal rank-k approximations to it can be computed more rapidly with, e.g., Lanczos methods. We will use $SVD(A_k)$ to denote the time to compute A_k. For more details on linear algebra, see [2, 10, 13, 16], and for more details on notation and our sampling matrix formalism, see [4, 9].

1.3 Problem Definition

We start with the following definition.

Definition 1. *Let A be an $m\times n$ matrix, let C be an $m\times c$ matrix whose columns consist of a small number c of columns of the matrix A, and let R be an $r\times n$ matrix whose rows consist of a small number r of rows of the matrix A. Then the $m\times n$ matrix A' is a* column-row-based low-rank matrix approximation *to A, or a* CUR matrix approximation*, if it may be explicitly written as $A' = CUR$ for some $c\times r$ matrix U.*

Note that the combined size of C, U and R is $O(mc + rn + cr)$, which is an improvement over A's size of $O(nm)$ when $c, r \ll n, m$.

The quality of a CUR matrix approximation depends on the choice of C and R as well as on the matrix U. We consider the following problem.

Problem 1 ***(Column-row-based low-rank matrix approximation problem).*** *Given a matrix $A \in \mathbb{R}^{m\times n}$, choose a sufficient number of columns and rows of A and construct a matrix U of appropriate dimensions such that*

$$\|A - CUR\|_F \leq (1+\epsilon)\,\|A - A_k\|_F\,. \tag{1}$$

Here, C is a matrix consisting of the chosen columns of A, R is a matrix consisting of the chosen rows of A, and A_k is the best rank k approximation to A. The number of columns of C and rows of R should be a function of k, $1/\epsilon$, and – in the case of randomized algorithms – a failure probability δ. The running time of the algorithms should be a low-degree polynomial in m, n.

Note that is not obvious whether there exist, and if so whether one can efficiently find, a small (depending on k, $1/\epsilon$, and $1/\delta$, but independent of m and n) number of columns and rows that provide such relative-error guarantees.

1.4 "Subspace Sampling" and Our Main Result

Our main result is the following theorem, which asserts the existence of an algorithm to solve Problem 1.

Theorem 1. *There exists a randomized algorithm that solves Problem 1. In the algorithm, $c = O(k^2 \log(1/\delta)/\epsilon^2)$ columns of A are chosen to construct C, and then $r = O(c^2 \log(1/\delta)/\epsilon^2)$ rows of A are chosen to construct R, and the matrix U is a weighted Moore-Penrose inverse of the intersection between C and R. The algorithm satisfies (1) with probability at least $1 - \delta$, it runs in time $O(SVD(A_k))$, and it uses the method of "subspace sampling" to sample columns to form C and rows to form R.*

For the moment, assume that we are given a set of columns, and consider the following theorem.

Theorem 2. *Let $\epsilon \in (0, 1]$. Let an $m \times n$ matrix A and an $m \times c$ matrix C consisting of c columns of A be given. There exists a randomized algorithm that runs in $O(mn)$ time and constructs an $r \times n$ matrix R consisting of $r = O(c^2 \ln(1/\delta)/\epsilon^2)$ rows of A and a $c \times r$ matrix U such that, with probability at least $1 - \delta$*

$$\|A - CUR\|_F \leq (1 + \epsilon) \left\|A - CC^+A\right\|_F .$$

This algorithm is described in Section 2. This result is a CUR matrix approximation that applies to any subset of columns C of the original matrix A, and has relative error with respect to CC^+A, i.e.,the projection of A on the subspace spanned by the columns of C.

Given Theorem 2, in order to establish Theorem 1 it suffices to find a C for which CC^+A is relative-error approximation to the best rank-k approximation provided by the SVD. It is known that such columns *exist* [18, 3], and recently we designed the first *polynomial time algorithm* to find such a C [8]. We summarize these results in the following theorem. See [8] for details.

Theorem 3. *Let $\epsilon \in (0, 1]$. Let an $m \times n$ matrix A and any positive integer k be given, and let A_k be the best rank k approximation to A. There exists a randomized algorithm that runs in $O(SVD(A_k))$ time and selects $c = O(k^2 \ln(1/\delta)/\epsilon^2)$ columns of A such that, if C is the $m \times c$ matrix whose columns are the selected columns of A, then with probability at least $1 - \delta$*

$$\left\|A - CC^+A\right\|_F \leq (1 + \epsilon) \|A - A_k\|_F .$$

Given a matrix A, it follows from Theorem 3 that we can either choose a column matrix C with this relative-error property or a row matrix R that has an analogous relative-error property. But combining those two matrices C and R does not immediately provide a CUR approximation with the relative-error guarantees. However, by combining Theorem 3 with Theorem 2, we establish Theorem 1 and obtain the relative-error CUR approximation.

The bulk of the technical work is the proof of Theorem 2. We will show that, given a matrix A and a set of its columns C, we can choose a set of its rows R such that CW^+R captures almost as much of A as does CC^+A in a relative error sense, where W^+ is a weighted Moore-Penrose generalized inverse of the intersection between C and R. This is obtained by extending our earlier ℓ_2-regression result [9] to a generalized ℓ_2-regression problem defined below. This extension is the main technical contribution of this paper.

The key technical insight that leads to the relative-error guarantees is that the rows are selected by a novel sampling procedure that we call "subspace sampling." Rather than sample rows from the input matrix with a probability distribution that depends on the Euclidean norms of its rows (which gives provable additive-error bounds [4, 5, 6]), in "subspace sampling" we randomly sample rows of the input matrix with a probability distribution that depends on the Euclidean norms of the rows of the top k singular vectors of the input matrix. This allows us to capture entirely a certain subspace of interest. This is required since we will be performing operations such as pseudoinversion that are not well-behaved to missing a dimension, no matter how insignificant its singular vlaue is. This is different than sampling to capture coarse statistics up to an additive error of $\epsilon||A||_F$, and it requires the use of more complex probabilities and more sophisticated analysis. The precise form of sampling is somewhat complicated and is shown in (4) and (11). It is similar to the sampling method we developed recently to solve the l_2 regression problem [9] that we extend here.

1.5 Related Work

To the best of our knowledge, ours is the first CUR matrix approximation with relative error. Previously, the only know CUR matrix approximations had a large additive error $\epsilon \|A\|_F$ [6, 7]. In fact, previous to our result, it was not even known whether such a relative-error CUR representation existed. Note that in the linear algebra community, there are several algorithms [19, 20, 1, 12, 11] to get C, U, R like ours for low-rank approximation, but none that is comparable in proven guarantees.

2 The Column-Row-Based Low-Rank Approximation

Assume that we are given an $m \times n$ matrix A and any set of c columns of A forming an $m \times c$ matrix C, and consider the following idea for approximating A. The columns of C are a set of "basis vectors" that are, of course, in general neither orthogonal nor normal. Thus, we can express every column of A as a linear combination of the columns of C. If m and n are large and $c = O(1)$, then this is an overconstrained least-squares fit problem. Thus, for all columns $A^{(j)}, j \in [n]$, we can solve

$$\min_{x_j \in \mathbb{R}^c} \left| A^{(j)} - Cx_j \right|_2 \tag{2}$$

in order to find a c-vector of coefficients x_j and get the optimal least-squares fit for $A^{(j)}$. Equivalently, we seek to solve

$$\min_{X \in \mathbb{R}^{c \times n}} \|A - CX\|_F \tag{3}$$

in order to express A as $A \approx CX_{opt}$, where $X_{opt} = C^+ A$ is a $c \times n$ matrix whose columns are the coefficient vectors $x_j, j \in [n]$ that minimize Equation (2).

We will now use a generalization of the ideas in [9] to modify the above approach to get a CUR decomposition for the matrix A, given C. Instead of solving the generalized least squares problem of Equation (3) we will solve a *sampled* version of the problem, constructed as follows:

1. Compute the SVD of C, $C = U_C \Sigma_C V_C^T$, where $U_C \in \mathbb{R}^{m \times \rho}$, $\Sigma_C \in \mathbb{R}^{\rho \times \rho}$, $V_C \in \mathbb{R}^{c \times \rho}$, and ρ is the rank of C.
2. Compute sampling probabilities p_i for all $i \in [m]$:

$$p_i = \frac{(1/3)\left|(U_C)_{(i)}\right|_2^2}{\sum_{j=1}^{n}\left|(U_C)_{(j)}\right|_2^2} + \frac{(1/3)\left|(U_C)_{(i)}\right|_2\left|\left(U_C^{\perp}{U_C^{\perp}}^T A\right)_{(i)}\right|_2}{\sum_{j=1}^{n}\left|(U_C)_{(j)}\right|_2\left|\left(U_C^{\perp}{U_C^{\perp}}^T A\right)_{(j)}\right|_2} + \frac{(1/3)\left|\left(U_C^{\perp}{U_C^{\perp}}^T A\right)_{(i)}\right|_2^2}{\sum_{j=1}^{n}\left|\left(U_C^{\perp}{U_C^{\perp}}^T A\right)_{(j)}\right|_2^2}. \quad (4)$$

   (Notice that $\sum_{i\in[m]} p_i = 1$.)
3. Create an $m \times r$ sampling matrix S and a $r \times r$ diagonal rescaling matrix D, as defined in [9], in r (an input parameter not greater than m) i.i.d. trials, using the p_i of Equation (4).
4. Return as output the $r \times n$ matrix $R = S^T A$ and the $c \times r$ matrix $U = \left(DS^T C\right)^+ D$.

Note that the time required to compute the SVD of C is $O(c^2 m)$, and computing the probabilities p_i of (4) takes an additional $O(cmn)$ time. Overall, the running time of the algorithm is $O(mn)$ since c, r are constants independent of m, n.

In order to obtain some intuition on the construction of U and R, consider the following "sampled and rescaled" version of Equation (3):

$$\min_{X \in \mathbb{R}^{c\times n}} \left\|DS^T A - DS^T CX\right\|_F. \quad (5)$$

Note that in this "sampled and rescaled" problem, the matrix X has the same dimensions as the matrix X of Equation (3), but that the number of constraints in the overconstrained problems has been reduced from m to r. We will see that solving this "sampled and rescaled" problem, and substituting the solution back into the original problem provides a CUR decomposition with a provable error bound. That is, let $\tilde{X}_{opt} = \left(DS^T C\right)^+ DS^T A = UR$ and use $\tilde{X}_{opt}$ as an approximation to X_{opt} (which achieves the optimal value for the full problem of Equation (3)). Then, we will be able to bound the error

$$\left\|A - C\tilde{X}_{opt}\right\|_F = \left\|A - C\underbrace{\left(DS^T C\right)^+ D}_{U}\underbrace{S^T A}_{R}\right\|_F = \|A - CUR\|_F. \quad (6)$$

Here we have let $W = S^T C$ be the $r \times c$ matrix that corresponds to rows of C that are in R, i.e., equivalently, W contains the common elements of C and R. In addition, C consists of a few columns of A, R consists of a few rows of the matrix A, and $U = (DW)^+ D$; note that in general, $(DW)^+ D \neq W^+$.

Rather than proving that this algorithm leads to a choice of U and R such that $\|A - CUR\|_F \leq (1+\epsilon) \|A - CC^+ A\|_F$, we establish in the next section a more general result (of independent interest), of which this is a corollary.

3 Approximating Generalized ℓ_2 Regression

3.1 The Generalized ℓ_2 Regression Problem

In this section, we present and analyze a random sampling algorithm to approximate the following generalized ℓ_2 regression (or least-squares fit) problem: given as input a matrix $A \in \mathbb{R}^{m \times n}$ of rank not greater than k (thus, $A = A_k$ in this section) and a target matrix $B \in \mathbb{R}^{m \times p}$, compute

$$\mathcal{Z} = \min_{X \in \mathbb{R}^{n \times p}} \|B - A_k X\|_F . \tag{7}$$

That is, compute the "best" approximation to the matrix B in the basis provided by the matrix $A = A_k$. Also of interest is the computation of matrices that achieve the minimum $\mathcal{Z}$. The "smallest" matrix among those minimizing $\|B - A_k X\|_F$ is

$$X_{opt} = A_k^+ B. \tag{8}$$

Note that we have not placed any constraints on the relationship between m and n. Since we allow $m > n$, $m = n$, and also $m < n$, our approximation algorithm for generalized ℓ_2 regression can be applied to both overconstrained and underconstrained problems. We do, however, constrain the rank of the matrix A (in this section only).

In the special case that $m \gg n$ and $p = 1$, we have the traditional (very overconstrained) ℓ_2 regression (or least-squares fit) problem: given as input a matrix $A \in \mathbb{R}^{m \times d}$ and a target vector $b \in \mathbb{R}^m$, compute $\mathcal{Z} = \min_{x \in \mathbb{R}^d} |b - Ax|_2$. If $m > d$ there are more constraints than variables and the problem is an *overconstrained* least-squares fit problem; in this case, there does not in general exist a vector x such that $Ax = b$. It is well-known that the minimum-length vector among those minimizing $|b - Ax|_2$ is $x_{opt} = A^+ b$, where A^+ denotes the Moore-Penrose generalized inverse of the matrix A.

In [9], we presented a sampling algorithm for this special case. The algorithm of [9] was the first to use SVD-based sampling probabilities similar to those we use in this paper to solve approximately the generalized ℓ_2 regression problem (7) and (8). Generalizing from $m \gg n$ and $p = 1$ to arbitrary m, n, and p constitutes the main technical contribution of this paper. Generalizing the analysis of [9] to $p > 1$ right-hand side vectors is straightforward; on the other hand, generalizing the analysis of [9] from $m \gg n$, i.e., very overconstrained ℓ_2 problems, to general m and n is more subtle.

3.2 Our Main Algorithm and Theorem for This Problem

We present and analyze an algorithm that constructs and solves an induced subproblem of the ℓ_2 regression problem of Equations (7) and (8). Let DS^TA be the $r \times n$ matrix consisting of the sampled and appropriately rescaled rows of the matrix $A = A_k$, and let DS^TB be the matrix consisting of the sampled and appropriately rescaled rows of B. Then consider the problem

$$\tilde{\mathcal{Z}} = \min_{X \in \mathbb{R}^{n \times p}} \left\| DS^TB - DS^TA_kX \right\|_F. \tag{9}$$

The "smallest" matrix $\tilde{X}_{opt} \in \mathbb{R}^{n \times p}$ among those that achieve the minimum value $\tilde{\mathcal{Z}}$ in the *sampled* generalized ℓ_2 regression problem of Equation (9) is

$$\tilde{X}_{opt} = \left(DS^TA\right)^+ DS^TB. \tag{10}$$

Since we will sample a number of rows $r \ll m$ of the original problem, we will compute (10), and thus (9), exactly. Our main theorem, Theorem 4, states that under appropriate assumptions on the original problem and on the sampling probabilities, the computed quantities $\tilde{\mathcal{Z}}$ and $\tilde{X}_{opt}$ will provide very accurate *relative error* approximations to the exact solution $\mathcal{Z}$ and the optimal matrix X_{opt}.

In more detail, our main algorithm for approximating the solution to the generalized ℓ_2 regression problem takes as input an $m \times n$ matrix A, an $m \times p$ matrix B, and a positive integer $r \leq m$. It returns as output a number $\tilde{\mathcal{Z}}$ and a $n \times p$ matrix $\tilde{X}_{opt}$ by doing the following:

1. Compute A_k, the "best" rank-k approximation to A.
2. Compute sampling probabilities p_i for all $i \in [m]$:

$$p_i = \frac{(1/3)\left|(U_{A,k})_{(i)}\right|_2^2}{\sum_{j=1}^n \left|(U_{A,k})_{(j)}\right|_2^2} + \frac{(1/3)\left|(U_{A,k})_{(i)}\right|_2 \left(U^{\perp}_{A,k}{U^{\perp}_{A,k}}^TB\right)_i}{\sum_{j=1}^n \left|(U_{A,k})_{(j)}\right|_2 \left(U^{\perp}_{A,k}{U^{\perp}_{A,k}}^TB\right)_j} + \frac{(1/3)\left(U^{\perp}_{A,k}{U^{\perp}_{A,k}}^TB\right)_i^2}{\sum_{j=1}^n \left(U^{\perp}_{A,k}{U^{\perp}_{A,k}}^TB\right)_j^2}. \tag{11}$$

3. Create an $m \times r$ sampling matrix S and an $r \times r$ diagonal rescaling matrix D, as defined in [9], in r i.i.d. trials, using the p_i of Equation (11).
4. Solve the induced subproblem, i.e., compute and return the number $\tilde{\mathcal{Z}}$ and the $n \times p$ matrix $\tilde{X}_{opt}$ given by (9) and (10), respectively.

The algorithm (implicitly) forms a sampling matrix S, the transpose of which samples with replacement a few rows of A_k and also the corresponding rows of B, and a rescaling matrix D, which is a diagonal matrix scaling the sampled rows of A_k and the elements of B. Since r rows of A_k and the corresponding

r rows of B are sampled, the algorithm randomly samples with replacement r of the m constraints in the original ℓ_2 regression problem. Thus, intuitively, the algorithm approximates the solution of $A_k X \approx B$ with the exact solution of the downsampled problem $DS^T A_k X \approx DS^T B$. Note that it is the space of constraints that is sampled and that the dimension of the unknown matrix X is the same in both problems.

An important aspect of the algorithm will be the nonuniform sampling probabilities (11). Computing these probabilities clearly takes time of the order of computing the best rank-k approximation to the matrix A plus computing the product ${U_{A,k}^{\perp}}{U_{A,k}^{\perp}}^T B$. Note that the probabilities (4) are a special case of (11).

Theorem 4 below is our main quality-of-approximation result for this generalized ℓ_2 regression problem. This result states that if the matrix achieving the minimum in the sampled problem is substituted back into the original problem then a good approximation to the original generalized ℓ_2 regression problem is obtained.

Theorem 4. *Let $\epsilon \in (0,1]$. Let an $m \times n$ matrix A that has rank no greater than k, an $m \times p$ matrix B, and the sampling probabilities $\{p_i\}_{i=1}^m$ be given. Let $\mathcal{Z}$ and X_{opt} be the solution to the full generalized ℓ_2-regression problem given by (7) and (8), repsectively, and let $\tilde{\mathcal{Z}}$ and $\tilde{X}_{opt}$ be the solution to the sampled generalized ℓ_2 regression problem, given by (9) and (10), respectively. If the sampling probabilities satisfy (11) and if $r \geq O(d^2 \ln(1/\delta)/\epsilon^2)$, then with probability at least $1-\delta$*

$$\left\| B - A_k \tilde{X}_{opt} \right\|_F \leq (1+\epsilon) \left\| B - A_k X_{opt} \right\|_F .$$

By considering the special case where B is any $m \times n$ matrix A, where A is any $m \times c$ matrix consisting of c actual columns of A, and where $k = \mathrm{rank}(C)$, then Theorem 2 follows as a corollary of Theorem 4.

3.3 Proof of Theorem 4

Due to space limitations, the proof is omitted. It is a generalization of the proof of the main theorem of [9]. Alternatively, see [8].

4 Concluding Remarks

We have presented the first known polynomial time algorithm for obtaining a $(1+\epsilon)$ relative error CUR approximation to a given matrix A. It was previously not even known if such a CUR representation exists, and the best known prior work involved large additive error of $\epsilon \|A\|_F$. This problem is of interest in data analysis, and improved bounds will be useful. Further, it is an interesting open problem whether deterministic results can be btained that match our randomized results. Finally, the sampling method we use has found a few applications since we introduced it in [9]; it is of interest to either find other applications or replace it by simpler sampling methods.

References

1. M.W. Berry, S.A. Pulatova, and G.W. Stewart. Computing sparse reduced-rank approximations to sparse matrices. Technical Report UMIACS TR-2004-32 CMSC TR-4589, University of Maryland, College Park, MD, 2004.
2. R. Bhatia. *Matrix Analysis*. Springer-Verlag, New York, 1997.
3. A. Deshpande, L. Rademacher, S. Vempala, and G. Wang. Matrix approximation and projective clustering via volume sampling. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1117–1126, 2006.
4. P. Drineas, R. Kannan, and M.W. Mahoney. Fast Monte Carlo algorithms for matrices I: Approximating matrix multiplication. *To appear in: SIAM Journal on Computing*.
5. P. Drineas, R. Kannan, and M.W. Mahoney. Fast Monte Carlo algorithms for matrices II: Computing a low-rank approximation to a matrix. *To appear in: SIAM Journal on Computing*.
6. P. Drineas, R. Kannan, and M.W. Mahoney. Fast Monte Carlo algorithms for matrices III: Computing a compressed approximate matrix decomposition. *To appear in: SIAM Journal on Computing*.
7. P. Drineas and M.W. Mahoney. On the Nyström method for approximating a Gram matrix for improved kernel-based learning. *Journal of Machine Learning Research*, 6:2153–2175, 2005.
8. P. Drineas, M.W. Mahoney, and S. Muthukrishnan. Polynomial time algorithm for column-row based relative-error low-rank matrix approximation. Technical Report 2006-04, DIMACS, March 2006.
9. P. Drineas, M.W. Mahoney, and S. Muthukrishnan. Sampling algorithms for ℓ_2 regression and applications. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1127–1136, 2006.
10. G.H. Golub and C.F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, 1989.
11. S.A. Goreinov and E.E. Tyrtyshnikov. The maximum-volume concept in approximation by low-rank matrices. *Contemporary Mathematics*, 280:47–51, 2001.
12. S.A. Goreinov, E.E. Tyrtyshnikov, and N.L. Zamarashkin. A theory of pseudoskeleton approximations. *Linear Algebra and Its Applications*, 261:1–21, 1997.
13. R.A. Horn and C.R. Johnson. *Matrix Analysis*. Cambridge University Press, New York, 1985.
14. F.G. Kuruvilla, P.J. Park, and S.L. Schreiber. Vector algebra in the analysis of genome-wide expression data. *Genome Biology*, 3:research0011.1–0011.11, 2002.
15. Z. Lin and R.B. Altman. Finding haplotype tagging SNPs by use of principal components analysis. *American Journal of Human Genetics*, 75:850–861, 2004.
16. M.Z. Nashed, editor. *Generalized Inverses and Applications*. Academic Press, New York, 1976.
17. P. Paschou, M.W. Mahoney, J.R. Kidd, A.J. Pakstis, S. Gu, K.K. Kidd, and P. Drineas. Intra- and inter-population genotype reconstruction from tagging SNPs. *Manuscript submitted for publication*.
18. L. Rademacher, S. Vempala, and G. Wang. Matrix approximation and projective clustering via iterative sampling. Technical Report MIT-LCS-TR-983, Massachusetts Institute of Technology, Cambridge, MA, March 2005.

19. G.W. Stewart. Four algorithms for the efficient computation of truncated QR approximations to a sparse matrix. *Numerische Mathematik*, 83:313–323, 1999.
20. G.W. Stewart. Error analysis of the quasi-Gram-Schmidt algorithm. Technical Report UMIACS TR-2004-17 CMSC TR-4572, University of Maryland, College Park, MD, 2004.
21. C.K.I. Williams and M. Seeger. Using the Nyström method to speed up kernel machines. In *Annual Advances in Neural Information Processing Systems 13: Proceedings of the 2000 Conference*, pages 682–688, 2001.

Finding Total Unimodularity in Optimization Problems Solved by Linear Programs*

Christoph Dürr and Mathilde Hurand

Laboratoire de Recherche en Informatique de l'Ecole Polytechnique (LIX),
CNRS UMR 7161, Palaiseau, France
{durr, hurand}@lix.polytechnique.fr

Abstract. A popular approach in combinatorial optimization is to model problems as integer linear programs. Ideally, the relaxed linear program would have only integer solutions, which happens for instance when the constraint matrix is totally unimodular. Still, sometimes it is possible to build an integer solution with same cost from the fractional solution. Examples are two scheduling problems [4,5] and the single disk prefetching /caching problem [3]. We show that problems such as the three previously mentioned can be separated into two subproblems: (1) finding an optimal feasible set of slots, and (2) assigning the jobs or pages to the slots. It is straigthforward to show that the latter can be solved greedily. We are able to solve the former with a totally unimodular linear program, from which we obtain simple combinatorial algorithms with improved worst case running time.

1 Introduction

In this work, we propose a specific approach to give simpler solutions to several optimization problems. Herein we considered three such optimization problems: the first two are scheduling problems : the Tall-Small Jobs Problem [4] and the Equal Length Jobs Problem [5]. The last one is about Offline prefetching and caching to minimize stall time [3]. In the Tall-Small Jobs Problem, we have m machines, n unit length jobs, some of which need to execute on all the machines at the same time. In the Equal Length Jobs Problem, jobs have a given equal length $p \geq 1$ and each job executes on a single machine. In both problems jobs have given release times and deadlines in between which they need to execute. The goal is to find a feasible schedule, and moreover, for the equal length jobs problem, a feasible schedule that minimizes total completion time of the jobs. The third optimization problem, Offline prefetching and caching to minimize stall time belongs to a different field: we are given a sequence of n page requests and a cache of size k. We can evict a page from the cache and fetch a new page to replace it. This operation cannot be done in parallel and costs F time units. When a page request is served it costs 1 time unit, unless the page is not yet in the cache, then a stall time is

* Supported by CNRS/NSF grant 17171 and ANR Alpage.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 315–326, 2006.

generated until the corresponding fetch completes. The goal is to decide when to evict and fetch pages so as to minimize the total stall time.

Though quite different, those three problems were solved in a similar manner. Unlike previous works where the authors transform the solution of a relaxed integer linear program into an integer solution, we used a new technique which simplifies the linear programs, and allows us to get directly optimal integer solutions: our approach is based on the observation that only the structure of the solution matters in the objective function, jobs and pages don't appear namely. Therefore, we completely dissociate the resolution process into two phases. First a simplified linear program can be used to find an optimal *skeleton* for the solution, and it is only later that we need to worry about *assigning* jobs or pages to this skeleton: for scheduling problems, the skeleton is a sequence of slots, and the assignment maps jobs to slots; for the cache problem, the skeleton is a sequence of intervals and the assignment associates to every interval a page to evict at the beginning and a page to fetch at the end. Our skeletons are such that the *assignment* phase just comes down do running a greedy algorithm. Our contribution is that this strategy, where you don't compute the assignment in the linear program, leads to linear programs with very simple constraint matrices, which not only are totally unimodular, but are (the transpose of) directed vertex adjacency matrices.

This allows us to reduce our scheduling problems into a shortest path problem and to reduce the caching problem into a min cost flow problem. It is interesting to notice, that compared to the previous linear programs, our linear programs are not completely novel: they are in fact relaxations of the former ones. We will make this point clear in the next section. The tall/small job scheduling problem and the prefetch/caching problem can be solved in worst case time $O(n^3)$ improving over respectively $O(n^{10})$ and $O^*(n^{18})$. Implementations are available from the authors home-pages.

2 Scheduling Equal Length Jobs

We will first introduce our method on a basic scheduling problem. We have n jobs, each of the same length p. Every job $j \in [1, n]$ comes with an interval $[r_j, D_j]$ consisting of a release time and a strict deadline. The goal is to find a schedule on m parallel machines, such that each job is assigned to an execution slot consisting of a particular machine and a time interval $[s_j, s_j + p) \subseteq [r_j, D_j]$. In addition, all execution slots assigned to a particular machine must be disjoint. One possible application could be frequency allocation. A network operator has a link with m optical fiber strings. Users ask for allocations of a frequency band of fixed size, inside the large frequency band that the particular user devices can handle. The goal is to find an assignment which satisfies all users. In addition we want to find the solution (if it exists) that minimizes the total completion time of the jobs. In the standard Graham notation, this problem is called $P|r_j; p_j = p; D_j| \sum C_j$.

Simons [8] give a complicated *greedy-backtrack* algorithm running in time $O(n^3 \log\log n)$, and later improved to $O(mn^2)$ [9]. Recently Brucker and Kravchenko [5] gave another algorithm for it, using a completely different approach. While their algorithm has worse complexity it is interesting because of a generalization which permits to solve an open problem, namely minimizing the weighted total completion time, where jobs are given priority weights.

A generalization of the feasibility problem is to find a maximal set of jobs, which can all be scheduled between their release times and deadlines. This problem is still open. Even the more general problem, when jobs come with a weight, and the goal is to find a maximal weighted feasible job set, is not known to be NP-hard.

2.1 Previous Work

First we observe that without loss of generality we can restrict ourselves to schedules where each execution slot starts at some release time plus a multiple of p, simply by shifting each slot as much to the beginning as possible. Let $\mathcal{T} = \{r_i + (a-1)p : 1 \le i, a \le n\}$ be this set of time points. And finally for a fixed schedule, if we number the execution slots from left to right, we can always reassign the j-th slot to the machine $(j \bmod m) + 1$. This way we don't need to take care of which machines the slots are assigned to, as long as there are at most m slots starting in every time interval of size p, which ensures that slots don't overlap on a particular machine. The linear program of [5] has a variable x_{jt} for each job j and time $t \in \mathcal{T}$, with the meaning that $x_{jt} = 1$ if job j is executed in the slot $[t, t+p)$. Then the program is to minimize $\sum_{jt}(t+p)x_{jt}$ subject to

$$\forall j \in [1,n] : \sum_{t \in \mathcal{T}} x_{jt} = 1 : \qquad \textbf{(every job cmpl.)}$$

$$\forall j \in [1,n], \forall t \in \mathcal{T} \backslash [r_j, D_j - p] : x_{jt} = 0 \qquad \textbf{(allowed interval)}$$

$$\forall s \in \mathcal{T} : \sum_{s \le t < s+p} \sum_{j \in [1,n]} x_{jt} \le m \qquad \textbf{(no overlapping)}$$

It is quite clear that there is an integer solution to this linear program if and only if there is a feasible schedule. While this linear program is not totally unimodular, the authors of [5] were still able to round the fractional solution into an integer solution of the same cost.

2.2 Relaxing the Linear Program

The linear program above computes not only the time slots of the schedule, but also the assignment of jobs to slots. However once we are given the *skeleton* of a schedule, meaning a set of time slots, it is always possible to assign the jobs greedily in EDD fashion: assign to every slot the job with smallest deadline among the available jobs. We release the linear program from the job assignment, in order to obtain a simpler linear program which only computes a feasible skeleton.

We proceed in several steps. First we weaken equation (*every job completes*) into the inequality $\sum_{t\in\mathcal{T}} x_{jt} \geq 1$. Then combining this new constraint with (*allowed interval*) leads to

$$\forall j \in [1,n] : \sum_{t\in[r_j, D_j-p]} : x_{jt} \geq 1. \tag{1}$$

Now for every pair $s,t \in \mathcal{T}, s \leq t$ we sum (1) over all jobs j that have $[r_j, D_j - p] \subseteq [s,t]$, upper-bounding the left hand side we obtain

$$\forall s,t \in \mathcal{T}, s \leq t : \sum_{s'\in[s,t]} \sum_j x_{js'} \geq |\{i : [r_i, D_i - p] \subseteq [s,t]\}|. \tag{2}$$

The constraints are clearly necessary, and we will show later they are also sufficient to get the optimal solutions. We reduce the number of variables and group $\sum_j x_{jt}$ by setting $y_t := \sum_{s\leq t} \sum_j x_{jt}$. Now y_t represents the total number of slots up to time t. To simplify notations we introduce an additional time point $t_0 < \min \mathcal{T}$, and set $\mathcal{T}' = \mathcal{T} \cup \{t_0\}$. For any time $t > t_0$, we define the functions $\text{round}(t) := \max\{s \in \mathcal{T}' : s \leq t\}$ and $\text{prec}(t) := \max\{s \in \mathcal{T}' : s < t\}$.

minimize $\sum_{t\in\mathcal{T}} (t+p)(y_t - y_{\text{prec}(t)})$
subject to

$$y_{t_0} = 0,\ y_{max\mathcal{T}} - y_{t_0} \leq n$$

$$\forall t \in \mathcal{T}, s = \text{prec}(t) : y_s - y_t \leq 0 \qquad \textbf{(order)}$$

$$\forall s \in \mathcal{T}, t = \text{round}(s+p) : y_t - y_s \leq m \qquad \textbf{(load)}$$

$$\forall i,j \in [1,n], s = \text{prec}(r_i), t = \text{round}(D_j - p), s \leq t : y_t - y_s \geq c_{ij}, \qquad \textbf{(incl.)}$$

where $c_{ij} := |\{k : [r_k, D_k] \subseteq [r_i, D_j]\}|$ is the number of jobs which have to be executed in the interval $[r_i, D_j]$.

The two first inequalities force y_t at first and last time step. In fact they are not necessary, but simplify the proof. The *order* inequalities ensure that (y_t) is a non decreasing sequence. The *load* inequalities verify that there are never more than m slots overlapping, and the *inclusion* inequalities, are there to ensure that there is a feasible mapping from jobs to slots, as we show next. Execpt the equality $y_{t_0} = 0$, the linear program has in every constraint exactly two variables, and with the respective coefficients $+1$ and -1. So the dual of the constraint matrix is the incidence matrix of a directed graph, which means the constraint matrix is totally unimodular. Now this property is preserved when adding a row with a single $+1$ entry, corresponding to $y_{t_0} = 0$. Therefore our linear program's constraints are in the from $Ay \leq b$ with A totally unimodular and b integer. This means that if the linear program has a solution then there is an optimal integer solution.

Let (y_t) be an optimal integer solution to this linear program. It indeed defines the skeleton of a solution: at each time $t \in \mathcal{T}$ there will be $y_t - y_{prec(t)}$ slots available for scheduling. Assigning greedily jobs to these slots means scheduling

at each time on each slot the job with the smallest deadline among the *available* jobs, meaning jobs which are not yet scheduled and which release time, deadline intervals permits to be scheduled in that slot.

Lemma 1. *The greedy assignment produces a valid schedule.*

Proof. We can notice that according to the second condition and the *inclusion* condition on $[t_0, max\mathcal{T}]$, $y_{max\mathcal{T}} = n$. We define V to be the multiset of time slots, such that slot $[t, t+p]$ is contained $y_t - y_{prec(t)}$ times. Therefore $|V| = n$. As mentioned in the previous section, by the *load* inequality, the slots can be assigned to machines without overlapping. So, it only remains to show that there exist assignments of jobs to slots, which respect release times and deadlines, and then that the greedy assignment is one of them.

Let U be the set of n jobs, and $G(U, V, E)$ a bipartite graph where E contains all edges between a job j and a slot $[t, t+p]$ if $[t, t+p] \in [r_j, D_j]$. We have to show that this graph has an injection from U to V, and will use Hall's theorem for this.

For a set of jobs S, we denote the neighboring slots ∂S, as the set of all slots t such that there is a job $j \in S$ with $(j, t) \in E$. We need to show that for every set S, $|S| \leq |\partial S|$, which by Hall's theorem, characterizes the existence of an injection. Let S be a set of jobs. Suppose S can be partitioned into $S_1 \cup S_2$ such that for any jobs $i \in S_1$ and $j \in S_2$ the intervals $[r_i, D_i - p]$ and $[r_j, D_j - p]$ are disjoint. Then clearly ∂S is the disjoint union of ∂S_1 and ∂S_2. Therefore we can without loss of generality assume that $\bigcup_{j \in S}[r_j, D_j]$ is a unique interval $[r_i, D_j]$, for $i = \text{argmin}_{i \in S} r_i$ and $j = \text{argmax}_{j \in S} D_j$. Then $|S| \leq c_{ij}$. Also the number of slots in the interval $[r_i, D_j)$ is exactly $y_t - y_s$ for $s = \text{prec}(r_i), t = \text{round}(D_j - p)$. From the *inclusion* inequality we get the required inequality and we conclude that there exist a valid assignment. Now since $|V| = |U| = n$, the injection is in fact a bijection, and there exists at least one perfect matching from jobs to slots with respect to release times and deadlines.

Proving that you can permute jobs in any of these matching to get the greedy matching is a quite standard in scheduling: let be two jobs i, j with $D_i < D_j$, and i is scheduled at some time t, while j is scheduled at some time s with $r_i \leq s < t$. Then it is possible to exchange the jobs i, j in their execution slots $[s, s+p)$ and $[t, t+p)$. By the use of a potential function, decreasing at each exchange, it is possible to transform our schedule in a so called *earliest due date schedule*. We conclude that since there exists at least a valid assignment, the greedy assignment is valid as well. □

This means that an optimal integer solution can be found with a standard linear program solve. But our linear program describes in fact the dual of a minimum cost flow problem, with uncapacitated arcs, and a single supply node, which corresponds to a shortest path problem and can be solved in time $O(NM)$, where N is the number of variables and M the number of constraints [10, p.558].

Theorem 1. *Our algorithm solves* $P|r_j; p_j = p; D_j| \sum C_j$ *in worst case time* $O(n^4)$.

Proof. Given the instance $m, p, r_1, \ldots, r_n, D_1, \ldots, D_n$, we construct the set $\mathcal{T}$ of $O(n^2)$ time points. Then we compute for every pair of jobs i, j the number of jobs c_{ij} which need to be scheduled in $[r_i, D_j]$. A naive algorithm does it in time $O(n^3)$, which would be enough for us. However it can be solved in time $O(n^2)$ using the following recursive formula. We assume jobs are indexed in order of release times. For convenience we set $c_{n+1,j} = 0$. Then $c_{i,j} = c_{i+1,j}+1$ if $D_i \leq D_j$ and $c_{i,j} = c_{i+1,j}$ if $D_i > D_j$.

This permits to construct the graph G and find in time $O(n^4)$ the optimal solution to the linear program, if there is one. Finally we do an earliest due date assignment of the jobs to the slots defined by the solution to the linear program in time $O(n \log n)$ using a priority queue. □

Note that in this section we don't beat the best known algorithm for $P|r_j; p_j = p; D_j| \sum C_j$ which is $O(mn^2)$ [9]. However, it allows us to introduce our technique that will be used later on.

3 Scheduling Tall and Small Jobs

In a parallel machine environment, sometimes maintenance tasks are to be done which involve all machines at the same time. Think of business meetings or inventory. Formally we are given n jobs of unit length $p = 1$, each job j comes with an integer release time and a deadline interval $[r_j, D_j]$ in which it must be scheduled. We distinguish two kind of jobs. The first n_1 jobs are *small* jobs, in the sense that they must be scheduled on one of the m parallel machines, it does not matter which one. The $n_2 = n - n_1$ remaining jobs are *tall* jobs, in the sense that they must be scheduled on all the m machines at the same time.

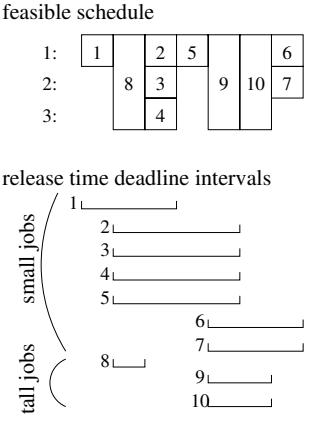


Fig. 1. Example for 3 machines

A time slot is an interval $[t, t + 1)$ for an integer boundary t. The goal is to find a *feasible* schedule, where each tall job is assigned to a different time slot, and each small job is assigned to a different (machine, time slot) pair for the remaining time slots. In addition the time slot to which some job j is assigned must be included in $[r_j, D_j]$.

This problem has been solved by Baptiste and Schieber [4], with a linear program using $O(n^2)$ variables and $O(n^2)$ constraints. The linear program is not totally unimodular, however they manage to show that for the particular objective function it always has an integer solution. We provide a linear program using only $O(n)$ variables but still $O(n^2)$ constraints, but whose constraint matrix is the incidence matrix of a directed graph, and can be solved in time $O(n^3)$ with a shortest path algorithm.

Baptiste and Schieber showed that we can assume that the time interval ranges only from 1 to n, otherwise the problem could easily be divided into two disjoint subproblems.

In a similar way than before, we will denote by x_t the total number of time slots assigned to tall jobs in $[1, t+1]$. For convenience we set $x_0 = 0$. The number of small jobs that must be scheduled in $[s,t]$ is $k_{s,t} = |\{j : j \leq n_1, [r_j, D_j] \subseteq [s,t]\}|$ and the same for tall jobs is $\ell_{s,t} = |\{j : j > n_1, [r_j, D_j] \subseteq [s,t]\}|$. Let be the following linear program, which does not have an objective value.

$$\forall t \in [1,n] : x_{t-1} \leq x_t \tag{3}$$

$$\forall t \in [1,n] : x_t - x_{t-1} \leq 1 \tag{4}$$

$$\forall s,t \in [1,n], s \leq t : x_{t-1} - x_{s-1} \geq \ell_{s,t} \tag{5}$$

$$\forall s,t \in [1,n], s \leq t : x_{t-1} - x_{s-1} \leq t - s - \lceil k_{s,t}/m \rceil. \tag{6}$$

Inequalities (3) make sure that (x_t) is a non decreasing sequence, (4) that only one tall job can be scheduled per unit-length interval, (5) that there are enough slots for the tall jobs and (6) that there are enough remaining slots for the small jobs.

Once again, the transpose of the constraint matrix is the adjacency matrix of an oriented graph, and the constant vector b is integer. As previously, it has optimal integer solutions.

Theorem 2. *Fix an instance of the tall/small scheduling problem. There is an integer solution to this linear program if and only if there is a feasible schedule.*

Proof. It is quite obvious that fixing (x_t) according to any feasible schedule will satisfy the constraints.

For the hard direction, let (x_t) be a solution to the linear program, we know it is integer. Then $x_t - x_{t-1}$ — which can be 0 or 1 — is the number of slots for tall jobs at time t. We will again use Hall's theorem to show that there is a valid assignment of the n_2 tall jobs to these slots. Inequality (5) for $[s,t] = [1,n]$ forces $x_n \geq n_2$. Now let be $G(U,V,E)$ the bipartite graph, where U are the n_2 tall jobs, and V the x_n slots. There is an edge between job j and time slot $[t, t+1]$ if it is included in $[r_j, D_j]$. We have to show that for every subset $S \subseteq U$, the number of neighboring slots in V is at least $|S|$. Let s be the smallest release time among S and t be the largest deadline among S. Again it is sufficient to show this claim for connected sets S in the sense that $\cup_{j \in S}[r_j, D_j] = [s,t]$. Now $|S| \leq \ell_{s,t} \leq x_{t-1} - x_{s-1}$, where the last expression is the number of slots in $[s,t]$. This completes the claim that there is a valid assignment from tall jobs to the slots.

For the small jobs, note that $a_{s,t} := (t-s) - (x_{t-1} - x_{s-1})$ is the number of remaining slots in $[s,t]$ which are not assigned to tall jobs, and $a_{s,t} \cdot m$ small jobs can fit in that interval. Again inequality (6) implies $k_{s,t} \leq m \cdot a_{s,t}$, and Hall's theorem shows that there is a valid assignment of small jobs to the remaining slots. □

In the original paper [4] the author gave a linear program which is solved in expected time $O(n^4)$ and worst case time $O(n^{10})$. Using the transformation into a shortest path problem allows us to improve this complexity.

Corollary 1. *The tall/small scheduling problem can be solved in worst case time* $O(n^3)$.

Proof. As in the second section, we have a linear program with $O(n)$ variables and $O(n^2)$ constraints which can be produced in time $O(n^2)$. We just take an arbitrary objective function in which all the variable coefficients are positive, and build the associated graph as in the previous section. Then we compute the all shortest paths from the source x_0, in time $O(n^3)$. If this computation detects a negative cycle, then the problem has no solution. Otherwise, we get the skeleton of a solution to the problem that minimize the total completion time of the tall jobs. Finally if there is a solution, the standard earliest due date assignment, first of tall jobs, then of small ones, produces a valid schedule in time $O(n \log n)$. □

Here again, a direction that we are still exploring is to find another shortest path algorithm inspired from [9], better fitted for these specific graphs, that could improve this complexity.

4 Prefetching

Caches are used to improve the memory access times. In this context the memory unit is called a *page*, and is stored on a slow disk. The cache can store up to k pages. Now if a page request arrives, and the page is already in the cache, it can be served immediately, otherwise it must first be fetched from the disk, and that introduces a stall time of F units. In the latter case the new page replaces some other page currently in the cache. The idea of *prefetching* is to fetch a page even before it is requested, so as to reduce the stall time: During a fetch which evicts some page y replacing it by some page z, other requests can be served for pages currently in the cache and different from y or z. In the single disk model we consider here, only a single fetch can occur at the same time. The goal is, knowing in advance the complete request sequence, to come up with a prefetch schedule, which minimizes total stall time.

While the real life problem is on-line, and has been extensively studied by Cao et al. [6], the offline problem has first been solved in 1998 [3], by the use of a linear program, for which it was shown that it always has an optimal integer solution, while not being totally unimodular. Later in 2000 [2], a polynomial time algorithm was given modeling the problem as a multi commodity flow with some postprocessing. Formally the problem can be defined as follows.

The Offline Prefetching *problem.* The input is a page request sequence $x_1, \ldots, x_n$, an initial cache set C_1, and a fetch duration F. Let $k = |C_1|$ be the cache size. A fetch is a tuple (s, y, e, z), where y, z are pages and $s, t \in [1, n]$ are time points with $s \leq e \leq s + F$. The meaning is that at time s, the page y leaves

the cache and at time e the page z enters the cache. Its cost, the induced stall time, is $F-(e-s)$. The goal is to come up with a fetch sequence minimizing the total stall time, such that two fetches intersect in at most one time point, and such that every request can be served, i.e. $\forall t \in [1,n] : x_t \in C_t$, where C_t is the cache at time t obtained from C_{t-1} by evicting/fetching all the pages that had to be evicted/fetched at time t. To simplify notation we assume that the request sequence contains at least k distinct pages, that C_1 consists of the first k distinct requests, and that at time 1, no page has left/entered the cache yet.

Albers, Garg and Leonardi defined a linear program with a characteristic variable for every fetch interval $[s,e]$, and two additional characteristic variables for every pair (y, $[s,e]$) indicating whether page y enters (resp. leaves) the cache at the beginning (resp. the end) of the fetch $[s,e]$. Finally they show that the linear program has always an integer solution for the considered objective function.

As observed in [3] without loss of generality the page to be evicted at time t from the cache C_{t-1} is the page, who's next request is furthest in the future or which is never requested again. Also without loss of generality the page to be fetched at time t is the page who's next request starting from t is nearest in the future. Therefore all the information about the fetches is in the time intervals, and we will write a linear program which produces only the time intervals in which evictions/fetches occur. The actual pages have to be assigned in a post processing, in greedy manner as just mentionned. Rather to have single variable for every interval and every page, we only count how many pages entered and how many left the cache in total since the beginning, which leaves us with $O(n)$ instead of $O(n^2F)$ variables. We denote by I_t (resp. O_t) the total number of pages which entered (resp. left) the cache up to time t included. We get the following linear program.

minimize $FO_n - FI_1 - \sum_{t=1}^{n}(O_t - I_t)$,
subject to

$$\forall t \in [2,n] : O_{t-1} \leq O_t \text{ and } I_{t-1} \leq I_t \tag{7}$$

$$\forall t \in [1,n] : O_t \geq I_t \tag{8}$$

$$\forall t \in [1,n] : O_t \leq I_t + 1 \tag{9}$$

$$\forall t \in [1,n] : I_{\min\{t+F,n\}} \geq O_t \tag{10}$$

$$\forall 1 \leq s \leq t \leq n : I_t - O_s \geq |\{x_s, x_{s+1}, \ldots, x_t\}| - k \tag{11}$$

Inequalities (7) make sure that (O_t) and (I_t) are non decreasing sequences, (8) that the cache cannot overflow, (9) that two fetches don't overlap in time, (10) that a fetch length is at most F and (11) that there are enough fetches to serve all requests.

The optimal solution of this linear program is always integer, since it is totally unimodular (for the same reason as in previous section: its constraint matrix the transposed incidence matrix of a directed graph.)

Theorem 3. *Let (I_t, O_t) be an optimal integer solution to the linear program. Then there is valid fetch sequence of the same cost, which can be built by greedy assignment.*

Proof. First we observe that the cost function makes sure that $O_n = I_n$, which ensures that all interval are eventually closed. The solution defines $m = O_n$ intervals as follows. For every $j = 1 \dots m$, let s_j be the smallest time such that $O_{s_j} \geq j$ and e_j the smallest time such that $I_{e_j} \geq j$. Then by (8) and (10) we have $s_j \leq e_j \leq s_j + F$. Which means that all intervals are well defined and of length smaller or equal than F. Now by (9), $e_j \leq s_{j+1}$ (otherwise, we wouldd have $I_{e_j} + 1 \geq O_{e_j} > O_{s_{j+1}}$ but $I_{e_j} = j$ and $O_{s_{j+1}} = j + 1$ by definition.), and this for all $j < m$, so the intervals do not overlap (but the ending point of one might be the starting point of another). Moreover the objective value of (I_t, O_t), equals the total stall time of these intervals, for at each time t, the difference O_t - I_t is 1 if an interval is currently opened and is 0 otherwise. It remains to prove that the greedy assignement of pages to evict/fetch to each interval is such that all requests are served, i.e. that the constraints (11) are sufficient. We denote by C_s the cache obtained at time s, after all entrances and evictions that occur at time s. We will show that the following invariant holds in a solution of our linear program for every time $s \in [1, n]$,

$$\forall t \in [s, n] : I_t - I_s \geq |\{x_s, \dots, x_t\} \backslash C_s|. \tag{12}$$

It means that if the number of pages requested in $[s, t]$ but not in the cache at time s is a, then at least a pages must enter the cache somewhere in $[s + 1, t]$. In particular it means for $t = s$, that the page requested at time s will be the in the cache at that moment. The proof is by induction on s.

Basis case $s = 1$. Let t_0 be the greatest request time such that x_{t_0} is not in C_1. Then by the assumption that initially the cache contains the first k distinct requests, we have that for $t < t_0$, $\{x_1, \dots, x_t\} \subseteq C_1$, so the right hand side of (12) is 0 and (12) holds by (7). For $t \geq t_0$, since the intersection of $\{x_1, \dots, x_t\}$ and C_1 is exactly k, the invariant holds by (11) and $O_1 = I_1 = 0$.

Induction case. Assume the invariant holds for some s. Let's show that it also holds for $s + 1$. Several things can happen at time $s + 1$, pages can leave the cache and pages can enter the cache. We will do these operations step by step, transform slowly I_s into I_{s+1} and C_s into C_{s+1}, and show that each step preserves the invariant (12).

By induction hypothesis $x_s \in C_s$, so $\{x_s, \dots, x_t\} \backslash C_s = \{x_{s+1}, \dots, x_t\} \backslash C_s$. Therefore, if nothing happens and no page enter or leave the cache, then $C_{s+1} = C_s$, $I_s = I_{s+1}$ and the invariant is preserved for $s + 1$.

Now we deal with the case when there is some page movement at time $s + 1$, that is $I_{s+1} > I_s$ or $O_{s+1} > O_s$ or both. We artificially decompose this page movement in as many times as needed, so that at each time there is only one operation happening: a fetch or an eviction. The page movements at those intermediary times are set so as to alternatively evict and enter pages, among the $O_{s+1} - O_s$ pages to evict and the $I_{s+1} - I_s$ pages to enter. Of course if a fetch is pending at time s, that is $|C_s| = k - 1$, then we start with entering a new page and otherwise if the cache is full, i.e. $|C_s| = k$, we start with evicting a page.

Since the number of total entrances and evictions up to some time can differ by at most one, it is possible to do so. Therefore, we need to do the induction case only in the case when a page is entering the cache or when one is leaving the cache but not both.

When page is entering the cache, we have $I_{s+1} = I_s + 1$. Let z be the page entering the cache, and let $t_0 \geq s+1$ be the next request time of z. Then if $t < t_0$, by the choice of z, all requests of $x_{s+1}, \ldots, x_t$ must be in C_{s+1}, so the right hand side of (12) at time $s+1$ is 0, and the inequality holds by (7). Now if $t \geq t_0$, since $z \in C_{s+1}$ but $z \notin C_s$, the left hand side of (12) at time $s+1$ has decreased by 1 compared to time s, but at the same time $I_{s+1} = I_s + 1$, so both sides of the invariant decrease by 1 and by induction the inequality is preserved at time $s+1$.

Now consider the case when a page leaves the cache. Let y be the leaving page. Then $I_s = I_{s+1}$ and $O_{s+1} = O_s + 1$. Let t_0 be the next request time of y or let $t_0 = n+1$ if y is never requested again. Then if $t < t_0$, removing y from C_{s+1} does not change the right hand side of (12) when replacing s by $s+1$. The left hand side does not change either since no page enters the cache, and the inequality is preserved. For $t \geq t_0$ however by the choice of the evicted page y, we have that $C_{s+1} \subseteq \{x_{s+1}, \ldots, x_t\}$. So the left hand side of (12) at time $s+1$ is $|\{x_{s+1}, \ldots, x_t\}| - (k-1)$, and $I_{s+1} = O_{s+1} - 1$ since we have just evicted a page. Therefore, (12) holds by (11). □

Theorem 4. *The offline prefetch problem can be solved in time $O(n^3)$ if $F = O(n)$ and in time $O(n^3 \log n)$ otherwise.*

Proof. The dual of the linear program is a min cost flow problem with uncapacitated arcs, where the supply/demand b_i of the nodes i are given by the coefficients in the cost function and where the arc costs c_{ij} are given by right hand sides of the inequalities. It could be solved in time $O(n^3 \log n)$ using [7]. To solve it in $O(n^3)$, when $F = O(n)$, we first explode the source of supply $F-1$ into $F-1$ vertices of supply $+1$ and do the same with the sink of demand $1-F$. The new graph has only sources of supply $+1$ and sinks of demand -1. Clearly there is a bijection between the min cost flows of the new and the original graph. Moreover a min cost flow matches sources to sinks such that the flow between a matched source/sink pair uses a shortest path (since the arcs have unbounded capacity) and such that the total distances are minimal. To obtain this flow we first compute the distances in the graph between all source/sink pairs, in time $O(n^3)$ using Floyd-Marshall's algorithm. Then we construct the bi-partite sources/sinks graph, where every edge is weighted with the source-sink distance in the original graph. Then a minimum weighted perfect matching can be computed in time $O(n^3)$ provided $F \in O(n)$, using Edmond's algorithm with adapted data-structures. The optimal flow then is obtained by adding a unit flow on the shortest path between source i and sink j, for every edge of the matching corresponding to source i and sink j.

Finally to get an optimal solution for the primal linear program, we use the standard technique of computing a shortest path tree in the residual graph obtained from the flow [1, chapter 9]. □

5 Conclusion

Further work would include trying to find other optimization problems where our technique may apply, and maybe generalize from it a general framework. We are also interested in improving the combinatorial algorithms that arise from the graph structures in the scheduling problems: indeed those graphs have, among others, the property that once the vertices drawn as points on a line, the arcs from left to right have positive weights and the ones from right to left negative. One idea for instance is to try and extract from Simons and Warmuth's algorithm a shortest path algorithm suitable for our class of graphs.

We wish to thank Arthur Charguéraud, Philippe Baptiste, Miki Hermann and Leo Liberti for helpful comments.

References

1. Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows : Theory, Algorithms and Applications*. Prentice Hall, 1993.
2. S. Albers and M. Büttner. Integrated prefetching and caching in single and parallel disk systems. *Information and Computation*, (198):24–39, 2005.
3. S. Albers, N. Garg, and S. Leonardi. Minimizing stall time in single and parallel disk systems. *Journal of the ACM*, (47):969–986, 2000.
4. Philippe Baptiste and Baruch Schieber. A note on scheduling tall/small multiprocessor tasks with unit processing time to minimize maximum tardiness. *Journal of Scheduling*, 6(4):395–404, 2003.
5. Peter Brucker and Svetlana Kravchenko. Scheduling jobs with equal processing times and time windows on identical parallel machines. Osnabrücker Schriften zur Mathematik H 257, Universität Osnabrück. Fachbereich Mathematik/Informatik, to appear in Journal of Scheduling, 2005.
6. P. Cao, E.W. Felten, A.R. Karlin, and K. Li. Implementation and performance of integrated application-controlled caching, prefetching and disk scheduling. *ACM Transaction of Computer Systems*, pages 188–196, 1995.
7. James B. Orlin. A faster strongly polynomial algorithm for the minimum cost flow problem. *Operations Research*, 41:338–350, 1993.
8. B. Simons. A fast algorithm for single processor scheduling. In *Proceedings IEEE 19th Annual Symposium on Foundations of Computer Science (FOCS'78)*, pages 246–252, 1978.
9. Barbara Simons and Manfred Warmuth. A fast algorithm for multiprocessor scheduling of unit-length jobs. *SIAM Journal on Computing*, 18(4):690–710, 1989.
10. Jan van Leeuwen. *HandBook of Theoretical Computer Science*, volume A: Algorithms and Complexity. Elsevier, 1990.

Preemptive Online Scheduling: Optimal Algorithms for All Speeds

Tomáš Ebenlendr[1], Wojciech Jawor[2], and Jiří Sgall[1]

[1] Mathematical Institute, AS CR, Žitná 25, CZ-11567 Praha 1, Czech Republic
{ebik, sgall}@math.cas.cz
[2] Department of Computer Science,University of California, Riverside, CA 92521
wojtek@cs.ucr.edu

Abstract. Our main result is an optimal online algorithm for preemptive scheduling on uniformly related machines with the objective to minimize makespan. The algorithm is deterministic, yet it is optimal even among all randomized algorithms. In addition, it is optimal for any fixed combination of speeds of the machines, and thus our results subsume all the previous work on various special cases. Together with a new lower bound it follows that the overall competitive ratio of this optimal algorithm is between 2.054 and $e \approx 2.718$.

1 Introduction

We study an online version of the classical problem of preemptive scheduling on uniformly related machines. We are given m machines with speeds $s_1 \geq s_2 \geq \ldots \geq s_m$ and a sequence of jobs, each described by its processing time (length). The time needed to process a job with length p on a machine with speed s is p/s. In the preemptive version, each job may be divided into several pieces, which can be assigned to different machines in disjoint time slots. (A job may be scheduled in several time slots on the same machine, and there may be times when a partially processed job is not running at all.) The objective is to find a schedule of all jobs in which the maximal completion time (makespan) is minimized.

In the online problem, jobs arrive one-by-one and we need to assign each incoming job to some time slots on some machines, without any knowledge of the jobs that arrive later. This problem, known as list scheduling, was first studied in Graham's seminal paper [11] for identical machines (i.e., $s_1 = \ldots = s_m = 1$). In the preemptive version, upon arrival of a job its complete assignment at all times must be given and we are not allowed to change this assignment later. In other words, the online nature of the problem is in the order in the input sequence and it is not related to possible preemptions and the time in the schedule.

The offline scheduling with makespan objective is well understood, and results for uniformly related machines were usually obtained using similar methods as for identical machines. Exact solutions can be computed with preemptions and approximation schemes exist for the non-preemptive version, which is NP-hard.

In the online version of non-preemptive scheduling, tight results are known only for two related or three identical machines. For deterministic algorithms on

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 327–339, 2006.

m identical machines, there still remains a small gap between the lower bound of 1.880 [13] and the upper bound of 1.923 [9], and a much larger gap for uniformly related machines, where the current bounds are 2.438 and 5.828 [2]. For randomized non-preemptive scheduling even less is known, the bounds are 1.581 and 1.916 for identical machines [3,14,1] and 2 and 4.311 for related machines [8,2]. It is still open whether randomized algorithms are better than deterministic.

The study of preemptive scheduling was partially motivated by studying the power of randomization in the non-preemptive setting. All previous lower bounds for randomized non-preemptive scheduling, except for [15], use speed sequences where tight bounds for preemptive scheduling are known and the same lower bound (but not the algorithm) works also in randomized non-preemptive setting.

All previous online algorithms that work for arbitrary speeds, preemptive or not, were obtained by a doubling approach. This means that a competitive algorithm is designed for the case when the optimum is approximately known in advance, and then, without this knowledge, it is used in phases with geometrically increasing guesses of the optimum. Such an approach probably cannot lead to an optimal algorithm. Instead, our algorithm computes exactly the current optimum at each step of the sequence and takes full advantage of this knowledge.

In all the previously known optimal algorithms for special cases, the optimal algorithms try to maintain certain fixed ratio of loads on the machines, generally with largest part of each job scheduled on the fast machine. These algorithms create no "holes" in the schedules, i.e., each machine is always busy from time 0 until some time t and idle afterwards. In contrast, our algorithm attempts to schedule the whole job on as slow machine as possible without violating the desired competitive ratio. This is done even at the cost of creating "holes" in the schedule, and using these holes efficiently is the key issue.

Previous results for preemptive online scheduling. The oldest result is by Chen *et al.* [4] for m identical machines. They gave an optimal algorithm with the competitive ratio $1/(1-(1-1/m)^m)$, which is $4/3$ for $m = 2$ and approaches $e/(e-1) \approx 1.582$ when $m \to \infty$. An optimal online algorithm for the special case of two related machines was given by Wen and Du [16], and by Epstein *et al.* [7]. The optimal competitive ratio in this case is $1 + s_1 s_2/(s_1^2 + s_1 s_2 + s_2^2)$. A special case with non-decreasing speed ratios, i.e., $s_{i-1}/s_i \leq s_i/s_{i+1}$ for $i = 2, \ldots, m-1$, was studied by Epstein [6]; note that this subsumes both identical and two related machines. Epstein gave an optimal competitive ratio for each sequence of speeds in this class; see Theorem 6.1 for its value. All these algorithms are deterministic and matching lower bounds are known to hold also for randomized algorithms.

For the general case, Ebenlendr and Sgall [5] obtained a 4-competitive deterministic algorithm and an e competitive randomized algorithm, where $e \approx 2.718$.

Epstein and Sgall [8] gave lower bounds on the competitive ratio for the worst case combination of speeds for any fixed m. These bounds approach 2 when $m \to \infty$ and all hold for randomized algorithms.

Our results. Our main result is an optimal online algorithm for preemptive scheduling on uniformly related machines. The algorithm achieves the best pos-

sible competitive ratio not only in the general case, but also for any number of machines and any particular combination of machine speeds. Our algorithm is deterministic, but its competitive ratio matches the best randomized algorithm. This proves that, similarly to the case of identical machines and other special cases studied before, randomization does not help for preemptive scheduling.

For any fixed set of speeds the competitive ratio of our algorithm can be computed by solving a linear program. We do not know, however, what is its worst case value over all speed combinations. Nevertheless, using the fact that there exists e-competitive randomized algorithm [5], we conclude that our (deterministic) algorithm is also e-competitive.

In Section 3 we present the linear program that gives the optimal competitive ratio and the lower bound, in Section 4 we describe the algorithm and its analysis.

In Section 5 we prove that no algorithm can be better than 2.054-competitive, by providing an explicit numerical instance on 100 machines. This improves the lower bound of 2 of Epstein and Sgall [8].

In Section 6 we analyze the linear program for computing the optimal competitive ratio for certain cases of speed sequences. We show that the formula given by Epstein [6] for non-decreasing speed ratios gives an upper bound on the competitive ratio for all possible speed combinations, and we extend the region where it is proven to be optimal. For $m = 3$ we give an exact formula for the competitive ratio for any speed combination.

2 Preliminaries

Let M_i, $i = 1, 2, \ldots, m$ denote the m machines, and let s_i be the speed of M_i. Without loss of generality we assume that the machines are sorted by decreasing speeds, i.e., $s_1 \geq s_2 \geq \ldots \geq s_m$. To avoid degenerate cases, we assume that $s_1 > 0$. Let $\mathcal{J} = (p_j)_{j=1}^{n}$ denote the input sequence of jobs, where n is the number of jobs and p_j is the length, or processing time, of jth job. Given $\mathcal{J}$, let $\mathcal{J}_j$ denote a sequence that is obtained from $\mathcal{J}$ by removing the last $j-1$ jobs.

The time needed to process a job with length p on machine with speed s is equal p/s; each machine can process at most one job at any time. Preemption is allowed, which means that each job may be divided into several pieces, which can be assigned to different machines, but any two time slots to which a single job is assigned must be disjoint (no parallel processing of a job); there is no additional cost for preemptions. The objective is to find a schedule of all jobs in which the maximal completion time (makespan) is minimized. In Graham's three-field notation the problem is denoted $Q|\text{pmtn}|C_{\max}$. For an algorithm A, let $C^{\mathsf{A}}_{\max}[\mathcal{J}]$ denote the makespan of the schedule of $\mathcal{J}$, produced by A. By $C^{*}_{\max}[\mathcal{J}]$ we denote the makespan of the optimal offline schedule of $\mathcal{J}$.

In the online version of this problem, denoted $Q|\text{online-list,pmtn}|C_{\max}$, jobs arrive one-by-one and we need to assign each incoming job to some time slots on some machines, without the knowledge of the jobs that arrive later. Upon release of each job a complete assignment of this job at all times must be given. An online algorithm A is called R-competitive if for every input $\mathcal{J}$, the makespan is at most R times the optimal makespan, i.e., $C^{\mathsf{A}}_{\max}[\mathcal{J}] \leq R \cdot C^{*}_{\max}[\mathcal{J}]$. In case

of a randomized algorithm, the same must hold for every input for the expected makespan of the online algorithm, $\mathbb{E}[C^{\mathsf{A}}_{\max}[\mathcal{J}]] \leq R \cdot C^*_{\max}[\mathcal{J}]$, where the expectation is taken over the random choices of the algorithm.

There are two easy lower bounds on $C^*_{\max}[\mathcal{J}]$. First, $C^*_{\max}[\mathcal{J}]$ can be bounded by the total work done on all machines. Second, the makespan of the optimal schedule is at least the makespan of the optimal schedule of any ℓ jobs. For $\ell < m$ this latter schedule uses only ℓ fastest machines, so the work of any ℓ jobs must fit on these machines. Formally, the bounds are:

$$C^*_{\max}[\mathcal{J}] \geq \frac{\sum_{j=1}^{n} p_j}{\sum_{i=1}^{m} s_i} \quad \text{and} \quad C^*_{\max}[\mathcal{J}] \geq \frac{P_\ell}{\sum_{i=1}^{\ell} s_i} \quad \text{for } \ell = 1, \ldots, m-1, \tag{1}$$

where P_ℓ denotes the sum of ℓ largest processing times in $\mathcal{J}$. It is known that $C^*_{\max}[\mathcal{J}]$ is the minimal value that satisfies (1), see [12,10,5].

The proof of the following lemma is very helpful in understanding our results.

Lemma 2.1 ([8]). *For any randomized R-competitive on-line algorithm A for preempted scheduling on m machines, and for any input sequence $\mathcal{J}$ we have $\sum_{j=1}^{n} p_j \leq R \cdot \sum_{i=1}^{m} s_i C^*_{\max}[\mathcal{J}_i]$. For non-preemptive scheduling, the same holds if $C^*_{\max}$ refers to the non-preemptive optimal makespan.*

Proof. Fix a sequence of random bits used by A. Let T_i denote the last time when at most i machines are running and set $T_{m+1} = 0$. First observe that $\sum_{j=1}^{n} p_j \leq \sum_{i=1}^{m} s_i T_i$: During the time interval $(T_{i+1}, T_i]$ at most i machines are busy, and their total speed is at most $s_1 + s_2 + \ldots + s_i$. Thus the maximum possible work done in this interval is $(T_i - T_{i+1})(s_1 + s_2 + \ldots + s_i)$. Summing over all i, we obtain $\sum_{i=1}^{m} s_i T_i$. In any valid schedule all the jobs are completed, so the observation follows.

Since the algorithm is online, the schedule for $\mathcal{J}_i$ is obtained from the schedule for $\mathcal{J}$ by removing the last $i-1$ jobs. At time T_i there are at least i jobs running, thus after removing $i-1$ jobs at least one machine is busy at T_i. So we have $T_i \leq C^{\mathsf{A}}_{\max}[\mathcal{J}_i]$ for any fixed random bits. Averaging over random bits of the algorithm and using $\sum_{j=1}^{n} p_j \leq \sum_{i=1}^{m} s_i T_i$, we have $\sum_{j=1}^{n} p_j \leq \mathbb{E}\left[\sum_{i=1}^{m} s_i C^{\mathsf{A}}_{\max}[\mathcal{J}_i]\right] = \sum_{i=1}^{m} s_i \mathbb{E}\left[C^{\mathsf{A}}_{\max}[\mathcal{J}_i]\right]$. Since A is R-competitive, i.e., $\mathbb{E}[C^{\mathsf{A}}_{\max}[\mathcal{J}_i]] \leq R \cdot C^*_{\max}[\mathcal{J}_i]$, the lemma follows. □

3 The Optimal Competitive Ratio and the Lower Bound

The optimal competitive ratio for given speeds $s_1, \ldots, s_m$ turns out to be equal to the best lower bound obtained by Lemma 2.1. In this section we formalize this bound using a linear program and prove the lower bound.

For each input sequence $\mathcal{J} = (p_j)_{j=1}^{n}$, Lemma 2.1 shows that the competitive ratio is at least $\sum_{j=1}^{n} p_j / (\sum_{i=1}^{m} s_i C^*_{\max}[\mathcal{J}_i])$. The set of input sequences can be restricted in two ways. First, we may assume that the processing times of jobs are non-decreasing: Sorting the jobs can only decrease the values of $C^*_{\max}[\mathcal{J}_i]$, as in each of these partial instances some jobs are possibly replaced by smaller jobs;

thus the bound on R can only increase. Second, the bound is invariant under scaling, so we may assume that $\sum_{i=1}^{m} s_i C^*_{\max}[\mathcal{J}_i] = 1$. The lower bound is then simply the sum of all processing times.

Now it is easy to give a linear program to compute the optimal lower bound, given the parameters $s_1 \geq \cdots \geq s_m$. The linear program has variables q_1, q_2, ..., q_m, O_1, O_2, ..., O_m. Variable q_1 corresponds to the sum of all processing times in $\mathcal{J}_m$, variables $q_2, \ldots, q_m$ to the processing times of the last $m-1$ jobs, and variables O_k correspond to $C^*_{\max}[\mathcal{J}_{m-k+1}]$.

Definition 3.1. *Let $r(s_1, \ldots, s_m)$ denote the value of the objective function of the optimal solution of the following linear program:*

$$\begin{aligned}
&\textbf{maximize} && r(s_1,\ldots,s_m) = q_1 + q_2 + q_3 + \ldots + q_m \\
&\textbf{subject to} && \\
q_1 + \ldots + q_k &\leq (s_1 + s_2 + \ldots + s_m)O_k && \text{for } k = 1,\ldots,m \\
q_j + q_{j+1} + \ldots + q_k &\leq (s_1 + s_2 + \ldots + s_{k-j+1})O_k && \text{for } 2 \leq j \leq k \leq m \\
1 &= s_1 O_m + s_2 O_{m-1} + \ldots + s_m O_1 && \\
q_j &\leq q_{j+1} && \text{for } j = 2,\ldots,m-1 \\
0 &\leq q_1 && \\
0 &\leq q_2 &&
\end{aligned} \tag{2}$$

The linear program has a feasible solution with the only non-zero variable $O_m = 1/s_1$. It is also easy to see that the objective function is bounded, the constraints imply that $q_1 + q_2 + \ldots + q_m \leq (s_1 + s_2 + \ldots + s_m)O_m \leq m \cdot s_1 O_m \leq m$. Thus the value $r(s_1, \ldots, s_m)$ is well-defined. Finally, the linear program has a quadratic number of constraints, and thus it can be solved efficiently.

Theorem 3.2. *Any randomized online algorithm for m machines with speeds $s_1 \geq s_2 \geq \cdots \geq s_m$ has competitive ratio at least $r(s_1, \ldots, s_m)$.*

Proof. There exist values $q_1^*, q_2^*, \ldots, q_m^*, O_1^*, O_2^*, \ldots, O_m^*$ of variables $q_1, q_2, \ldots, q_m, O_1, O_2, \ldots, O_m$, which satisfy all the constraints of the linear program (2) and $r(s_1, \ldots, s_m) = q_1^* + q_2^* + \ldots + q_m^*$. Create instance $\mathcal{I}$ as follows: The first m jobs have processing times $p_1 = \cdots = p_m = q_1^*/m$. The remaining $m-1$ jobs have processing times $p_{m+1} = q_2^*, p_{m+2} = q_3^*, \ldots, p_{2m-1} = q_m^*$.

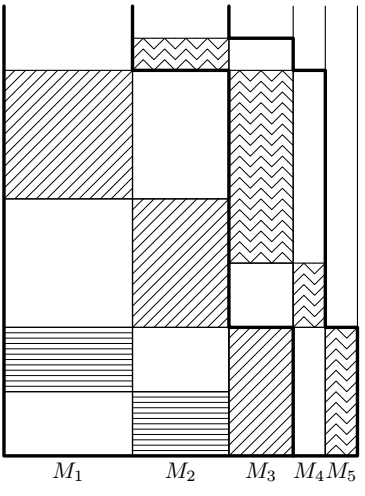
We claim that the first two families of constraints of (2) guarantee that the values O_k^* satisfy (1) for $C^*_{\max}[\mathcal{I}_{m-k+1}]$. If the set of ℓ largest jobs includes a job with processing time p_1 then it is easy to see that this bound in (1) is dominated either by the second bound for $\ell' < \ell$ such that ℓ' largest jobs do not contain any job of processing time p_1, or by the first bound which includes all m jobs with processing time p_1. In the remaining cases, the constraints of (2) imply the corresponding bounds in (1). Thus $C^*_{\max}[\mathcal{I}_{m-k+1}] \leq O_k^*$ for $k = 1, 2, \ldots, m$. Finally, Lemma 2.1 implies that the competitive ratio of any algorithm is at least

$$\frac{q_1^* + q_2^* + \ldots + q_m^*}{\sum_{i=1}^{m} s_i C^*_{\max}[\mathcal{J}_i]} \geq \frac{q_1^* + q_2^* + \ldots + q_m^*}{s_1 O_m^* + \ldots + s_m O_1^*} = r(s_1, \ldots, s_m),$$

using the constraint $s_1 O_m^* + \ldots + s_m O_1^* = 1$ in the last step. □

4 The Optimal Algorithm

In this section we present the $r(s_1, \ldots, s_m)$-competitive algorithm RatioStretch for all combinations of speeds. The idea of the algorithm is fairly natural. First we compute the desired competitive ratio for the given speeds, $r = r(s_1, \ldots, s_m)$. Next, for each arriving job, we compute the optimal makespan for jobs that have arrived so far and run the incoming job as slow as possible so that it finishes at r times the computed optimal makespan. There are many ways of creating such a schedule given the flexibility of preemptions. We choose a particular one based on the notion of a *virtual machine* from [5].



The ith *virtual machine*, denoted V_i, at each time τ contains the ith fastest machine among those real machines M_1, M_2, ..., M_m that are idle at time τ. When we schedule (a part of) a job on a virtual machine during some interval, we actually schedule it on the corresponding real machines that are uniquely defined at each time; this is always possible to achieve using preemptions. To simplify the description of the algorithm, we assume that there are infinitely many real machines of speed zero, i.e., $s_i = 0$ for any $i > m$. Scheduling a job on one of these zero-speed machines means that we do not schedule the job at the given time at all. Initially, each virtual machine V_i corresponds to the real machine M_i; as the incoming jobs are scheduled, the assignment of the real machines to the virtual machines changes.

The figure above illustrates an example of a schedule of three jobs produced by RatioStretch. Similarly shaded regions correspond to scheduled pieces of the same job. The bold lines mark the first two virtual machines.

In our algorithm, upon arrival of a job j we compute a value T_j defined as r times the current optimal makespan. Then we find two adjacent virtual machines V_k and V_{k+1}, and time t_j, such that if we schedule j on V_{k+1} in the time interval $(0, t_j]$ and on V_k from t_j on, then j finishes exactly at time T_j. It is essential that each job is stretched over the whole interval $(0, T_j]$, which is the maximal time interval which it can use without violating the desired competitive ratio. Next we update the virtual machines, which means that in the interval $(0, T_j]$ we merge V_k and V_{k+1} into V_k and shift machines V_{i+1}, $i > k$, to V_i. Then we continue with the next job. This gives a complete informal description of the algorithm sufficient for its implementation.

To prove that our algorithm works, it is sufficient to show that each job j scheduled on V_1, completes by time T_j; this is equivalent to the fact that we can schedule j as described above. We show that this is true due to our choice of r.

To facilitate the proof, we maintain an assignment of scheduled jobs (and consequently busy machines) to the set of virtual machines, i.e., for each virtual

machine V_i we compute a set $\mathcal{S}_i$ of jobs assigned to V_i. Although the incoming job j is split between two different virtual machines, at the end of each iteration each scheduled job belongs to exactly one set $\mathcal{S}_i$, since right after j is scheduled the virtual machines executing this job are merged (during the execution of j). We stress that the sets $\mathcal{S}_i$ serve only as means of bookkeeping for the purpose of the proof, and their computation is not an integral part of the algorithm.

At each time τ, machine $M_{i'}$ belongs to V_i if it is the ith fastest idle machine at time τ, or if it is running a job $j \in \mathcal{S}_i$ at time τ. At each time τ the real machines belonging to V_i form a set of adjacent real machines, i.e., all machines $M_{i'}, M_{i'+1}, \ldots, M_{i''}$ for some $i' \leq i''$. This relies on the fact that we always schedule a job on two adjacent virtual machines which are then merged into a single virtual machine during the times when the job is running, and on the fact that these time intervals $(0, T_j]$ increase with j, as adding new jobs cannot decrease the optimal makespan.

Let $v_i(t)$ denote the speed of the virtual machine V_i at time t, which is the speed of the unique idle real machine that belongs to V_i. Let $W_i(t) = \int_0^t v_i(\tau)d\tau$ be the total work which can be done on machine V_i in the time interval $(0, t]$. By definition we have $v_i(t) \geq v_{i+1}(t)$ and thus also $W_i(t) \geq W_{i+1}(t)$ for all i and t. Note also that $W_{m+1}(t) = v_{m+1}(t) = 0$ for all t.

Algorithm RatioStretch**.** First solve the linear program (2) for a fixed sequence of speeds $s_1 \geq s_2 \geq \cdots \geq s_m$ given on input. Let $r = r(s_1, \ldots, s_m)$ be the optimal objective value. Also initialize $T_0 := 0$, $\mathcal{S}_i := \emptyset$, $v_i(\tau) := s_i$, and $v_{m+1}(\tau) := 0$ for all $i = 1, 2, \ldots, m$ and $\tau \geq 0$.
For each arriving job j, compute the output schedule as follows:

(1) Let $T_j := r \cdot C^*_{\max}[(p_i)_{i=1}^j]$.
(2) Find the smallest k such that $W_k(T_j) \geq p_j \geq W_{k+1}(T_j)$. If such k does not exist, then output "failed" and stop. Otherwise find time $t_j \in [0, T_j]$ such that $W_{k+1}(t_j) + W_k(T_j) - W_k(t_j) = p_j$.
(3) Schedule job j on V_{k+1} in time interval $(0, t_j]$ and on V_k in $(t_j, T_j]$.
(4) Set $v_k(\tau) := v_{k+1}(\tau)$ for $\tau \in (t_j, T_j]$, and $v_i(\tau) := v_{i+1}(\tau)$ for $i = k+1, \ldots, m$ and $\tau \in (0, T_j]$. Also set $\mathcal{S}_k := \mathcal{S}_k \cup \mathcal{S}_{k+1} \cup \{j\}$, and $\mathcal{S}_i := \mathcal{S}_{i+1}$ for $i = k+1, \ldots, m$.

We leave out implementation details. We only note that job j can be preempted only at times $T_{j'}$ for $j' < j$ or at times $t_{j'}$ for $j' \leq j$, i.e., at most $2j-1$ times. The total number of preemptions is at most $n(m+1)$, since at most $m-1$ jobs are preempted at each time T_j and at most two jobs are preempted at each time t_j. This also implies that the functions v_i and W_i are piecewise linear with at most $2n$ parts. Thus it is possible to represent and process them efficiently. The computation of r and T_j is efficient as well.

Theorem 4.1. RatioStretch *is $r = r(s_1, \ldots, s_m)$ competitive for online preemptive scheduling on m uniformly related machines with speeds $s_1 \geq s_2 \geq \cdots \geq s_m$.*

Proof. If RatioStretch schedules a job, it is always completed at time $T_j \leq r \cdot C^*_{\max}[(p_i)_{i=1}^n]$. Thus to prove the theorem, it is sufficient to guarantee that the

algorithm does not fail to find machines V_k and V_{k+1} for the incoming job j. This holds if there is always enough space on V_1, i.e., that $p_j \leq W_1(T_j)$ in the iteration when j is to be scheduled. Since $W_{m+1} \equiv 0$, this is sufficient to guarantee that required k exists. Given the choice of k, it is always possible to find time t_j as the expression $W_{k+1}(t_j)+W_k(T_j)-W_k(t_j)$ continuously decreases from $W_k(T_j) \geq p_j$ for $t_j = 0$ to $W_{k+1}(T_j) \leq p_j$ for $t_j = T_j$.

To avoid cases, we assume that the input sequence starts by m jobs with processing time 0. In RatioStretch, they are assigned to V_1, but they are actually never running and thus do not affect the schedule produced by RatioStretch.

Let $j_1, j_2, \ldots, j_{m-1}$ denote the last $m-1$ jobs in $\mathcal{S}_1$, ordered as they appear on input. Let $\mathcal{I}$ be the sequence of the remaining jobs in $\mathcal{S}_1$, and let P be the total processing time of jobs in $\mathcal{I}$. Finally, let $j_m = j$ be the incoming job.

Consider any $i = 1, \ldots, m$ and any time $\tau \in (0, T_{j_i}]$. Using the fact that the times T_j are non-decreasing in j and that the algorithm stretches each job j over the whole interval $(0, T_j]$, there are at least $m-i$ jobs from $\mathcal{S}_1$ running at τ, namely jobs $j_i, j_{i+1}, \ldots, j_{m-1}$. Including the idle machine, there are at least $m+1-i$ real machines belonging to V_1. Since V_1 is the first virtual machine and the real machines are adjacent, they must include the fastest real machines M_1, $\ldots$, M_{m+1-i}. It follows that the total work that can be processed on the real machines belonging to V_1 during the interval $(0, T_{j_m}]$ is at least $s_1 T_{j_m} + s_2 T_{j_{m-1}} + \cdots + s_m T_{j_1}$. The total processing time of jobs in $\mathcal{S}_i$ is $P + p_{j_1} + p_{j_2} + \cdots + p_{j_{m-1}}$. Thus to prove that j_m can be scheduled on V_1 we need to verify that

$$p_{j_m} \leq s_1 T_{j_m} + s_2 T_{j_{m-1}} + \cdots + s_m T_{j_1} - (P + p_{j_1} + p_{j_2} + \cdots + p_{j_{m-1}}). \quad (3)$$

Let $\nu_1, \nu_2, \ldots, \nu_m$ be the sequence of jobs $j_1, j_2, \ldots j_m$ ordered so that the processing times p_{ν_i} are non-decreasing, i.e., $p_{\nu_i} \leq p_{\nu_{i+1}}$ for $i = 1, \ldots, m-1$. We claim that for each $i = 1, \ldots, m$,

$$T_{j_i} \;\geq\; r \cdot C^*_{\max}[(\mathcal{I}, p_{j_1}, p_{j_2}, \ldots, p_{j_i})] \geq r \cdot C^*_{\max}[(\mathcal{I}, p_{\nu_1}, p_{\nu_2}, \ldots, p_{\nu_i})].$$

The first inequality follows since removing some jobs from the input sequence cannot increase $C^*_{\max}$. The second one can be thought as replacing some of the jobs by smaller ones and then permuting them; this also cannot increase $C^*_{\max}$.

The inequality (4) and the fact that $p_{j_1} + p_{j_2} + \cdots + p_{j_m} = p_{\nu_1} + p_{\nu_2} + \cdots + p_{\nu_m}$ together imply that to prove (3), it is sufficient to prove

$$P + p_{\nu_1} + p_{\nu_2} + \cdots + p_{\nu_m} \leq r \cdot \sum_{i=1}^{m} s_i C^*_{\max}[(\mathcal{I}, p_{\nu_1}, p_{\nu_2}, \ldots, p_{\nu_{m-i+1}})]. \quad (4)$$

Let $\sigma = \sum_{i=1}^{m} s_i C^*_{\max}[(\mathcal{I}, p_{\nu_1}, p_{\nu_2}, \ldots, p_{\nu_i})]$. Let $q_1 = (P + p_{\nu_1})/\sigma$, $q_j = p_{\nu_j}/\sigma$, for $j = 2, 3, \ldots, m$, and let $O_k = C^*_{\max}[(\mathcal{I}, p_{\nu_1}, p_{\nu_2}, \ldots, p_{\nu_k})]/\sigma$ for $k = 1, \ldots, m$. These values satisfy all constraints of (2), as follows by using inequalities (1) for instances $(\mathcal{I}, p_{\nu_1}, p_{\nu_2}, \ldots, p_{\nu_k})$. Thus $(P + p_{\nu_1} + p_{\nu_2} + \cdots + p_{\nu_m})/\sigma = q_1 + q_2 + \cdots + q_m \leq r$. This proves (4) and thus also (3) and correctness of the algorithm. □

Theorems 3.2 and 4.1 show that RatioStretch is as good as any randomized algorithm. Together with e-competitive randomized algorithm from [5] we get:

Corollary 4.2. RatioStretch *is e-competitive for online preemptive scheduling on uniformly related machines with arbitrary speeds, where* $e \approx 2.718$.

5 Numerical Lower Bounds

We have the optimal algorithm for arbitrary speeds, but we do not know the numerical value of its competitive ratio. The competitive ratio for m machines is equal to the solution of a quadratic program obtained from (2) by considering s_i to be variables (in addition to q_j and O_k). However, this quadratic program is not convex and we do not know how to solve it efficiently. We have obtained some lower bounds numerically using mathematical software Maple.

A lower bound is simply a feasible solution of (2). Once the values of s_i are given, verification only involves solving a linear program. Once also the optimal values q_j^* are given, it is trivial to compute values O_k^* and verify (2). A complete file with our solutions for $m = 3, \ldots, 70$ and $m = 100$ in a format suitable for computer verification is available at `http://math.cas.cz/sgall/ps/optrel/`.

We obtain lower bounds improving the bounds from [8] for $m \geq 8$. For $m \geq 58$ the bounds are above 2, and for $m = 100$ we obtain a speed combination with optimal competitive ratio above 2.054, see Figure 1. Thus we have:

Theorem 5.1. *For any online algorithm for preemptive scheduling on uniformly related machines, the competitive ratio is at least* 2.054.

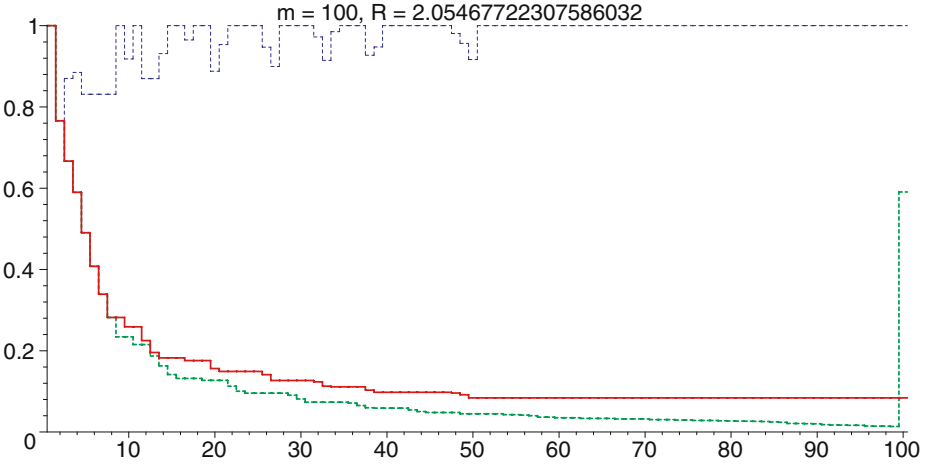


Fig. 1. An instance giving a lower bound of 2.054. The solid curve shows speeds s_i, the dashed curve (bottom) shows reversed scaled job variables, and the thin dashed curve (top) shows inverses of the speed ratios, i.e., s_{i-1}/s_i. The values of q_j are printed in reverse order, i.e., column i shows q_{m-i+1}, and scaled so that $q_m = 1 = s_1$. This ordering and scaling emphasizes the relations between job sizes and speeds.

6 Special Cases

One approach to analyze the optimal competitive ratio is to give a symbolic solution to the linear program (2). A feasible primal solution gives a lower bound (which can be easily turned into an sequence of jobs, as we have seen before). A feasible solution of a dual linear program gives an upper bound on the competitive ratio. A dual solution actually means that we form a positive linear combination of some of the linear constraints so that the resulting inequality bounds the objective function by the desired competitive ratio.

A basic solution of the linear program is described by giving a subset of constraints where equality holds. If for some range of speeds this subset does not change in the optimal solution, the optimal competitive ratio is given by a rational function of the speeds. However, in general, for different speed sequences we need to use different subsets of the constraints.

We give two cases where we can provide analysis along these lines. First we generalize the case of non-decreasing speed ratios from [6]. We prove that the same formula is a general upper bound on the competitive ratio and that it is actually optimal for a slightly larger region of speed sequences. Then we give a complete analysis of the case $m = 3$.

We denote $\quad S = \sum_{i=1}^{m} s_i, \quad \alpha = 1 - \frac{s_1}{S} = \frac{s_2 + \cdots + s_m}{s_1 + s_2 + \cdots + s_m}.$

Theorem 6.1. *Let* $R = \left(\sum_{i=1}^{m} \alpha^{i-1} s_i/S\right)^{-1}$. *Then* $r(s_1, \ldots, s_m) \le R$ *for any speeds* $s_1 \ge \cdots \ge s_m$ *and thus* RatioStretch *is* R*-competitive. Furthermore* $r(s_1, \ldots, s_m) = R$ *whenever*

$$s_1 \left(1 + \alpha + \ldots + \alpha^{i-1}\right) \le s_1 + \ldots + s_i \qquad \text{for all } i = 2, \ldots, m-1. \tag{5}$$

Proof. **The upper bound.** As described in the outline above, we form a positive linear combination of some constraints of the linear program (2). We add up

$$q_k \le s_1 O_k \quad \text{for } 2 \le k \le m, \quad \text{times } x_k = \sum_{i=2}^{k} s_{m-k+i} \alpha^{i-2} \tag{6}$$

$$q_1 + \ldots + q_k \le S O_k \quad \text{for } 1 \le k \le m, \quad \text{times } y_k = s_{m-k+1} - \frac{s_1}{S} x_k$$

We need to show that $y_k \ge 0$ and simplify the resulting inequality. For all $k = 1, \ldots, m$ we have

$$x_k + y_k = s_{m-k+1} + \left(1 - \frac{s_1}{S}\right) \sum_{i=2}^{k} s_{m-k+i} \alpha^{i-2} = \sum_{i=2}^{k+1} s_{m-k+i-1} \alpha^{i-2}, \tag{7}$$

$$\text{so} \quad y_k = \sum_{i=2}^{k+1} s_{m-k+i-1} \alpha^{i-2} - x_k \ge \sum_{i=2}^{k} (s_{m-k+i-1} - s_{m-k+i}) \alpha^{i-2} \ge 0.$$

In addition, (7) implies that $x_k + y_k = x_{k+1}$ for $k < m$, and $x_m + y_m = \sum_{i=1}^{m} s_i \alpha^{i-1} = S/R$. Using also the fact that $y_1 = s_m = x_2$, the left-hand side of the linear combination given by (6) is equal to

$$\begin{aligned} & q_1(y_1 + y_2 \ldots + y_m) + q_2(x_2 + y_2 + \ldots + y_m) + \cdots + q_m(x_m + y_m) \\ & \quad = (q_1 + \ldots + q_m)S/R. \end{aligned}$$

The right-hand side of the linear combination given by (6) is equal to

$$y_1 SO_1 + (x_2 s_1 + y_2 S)O_2 + \ldots + (x_m s_1 + y_m S)O_m = S(s_1 O_m + \ldots + s_m O_1) = S,$$

using the definitions of x_k and y_k and the third constraint of (2). We conclude that any feasible solution of (2) satisfies the linear combination given by (6), which simplifies to $(q_1 + \ldots + q_m)S/R \leq S$, and thus $r(s_1, \ldots, s_m) \leq R$.

The lower bound. Now we give a primal solution of (2) with the value of objective R, assuming (5). Naturally, this exactly corresponds to the lower bound from [6]. The solution is:

$$\begin{aligned} O_k &= R\frac{\alpha^{m-i}}{S} \qquad \text{for } k = 1, \ldots, m \\ q_1 &= R\alpha^{m-1} \\ q_k &= s_1 O_k = R\frac{s_1}{S}\alpha^{m-k} \qquad \text{for } i = 2, \ldots, m. \end{aligned}$$

The verification of the feasibility is straighforward; it uses the assumption (5) for the second constraint of (2). Details are omitted due to space constraints. □

We remark that the upper bound in the previous theorem is not bounded by any constant in the region where the condition (5) does not hold.

Epstein [6] in Claim 1 exactly proves that (5) is satisfied by speed sequences with non-decreasing speed ratios. In her notation, speeds are listed in reversed order and normalized so that $s_m = 1$; her x is our $1/\alpha$. However, (5) is satisfied for a slightly wider range of speeds. One example is $s_1 = 2$, $s_2 = s_3 = 1$.

Theorem 6.2. *For $m = 3$ and any speeds $s_1 \geq s_2 \geq s_3$,*

$$r(s_1, s_2, s_3) = \begin{cases} \left(\frac{s_1}{S} + (1 - \frac{s_1}{S})\frac{s_2}{S} + (1 - \frac{s_1}{S})^2\frac{s_3}{S}\right)^{-1} & \text{if } \frac{s_1}{s_2} \leq \frac{s_2}{s_3} + 1 \\ \dfrac{S^2}{s_1^2 + s_2^2 + s_3^2 + s_1 s_2 + s_1 s_3 + s_2 s_3} & \text{if } \frac{s_1}{s_2} \geq \frac{s_2}{s_3} + 1 \end{cases}$$

The function $r(s_1, s_2, s_3)$ has maximal value $\frac{37+7\sqrt{7}}{38} \approx 1.461$, and thus this is also the optimal competitive ratio for $m = 3$ over all speeds.

Proof. The first case follows directly from Theorem 6.1 and the observation that the case condition is equivalent to (5). For the second case, let $R = S^2/(s_1^2 + s_2^2 + s_3^2 + s_1 s_2 + s_1 s_3 + s_2 s_3)$. We omit the simple verification of solutions.

The upper bound. We again form a positive linear combination of some constraints of the linear program (2). We add up

$$\begin{array}{rrrrll}
q_1 & \le SO_1 & & & \text{times} & s_3 \\
q_1 + q_2 & \le & SO_2 & & \text{times} & s_2 \\
q_1 + q_2 + q_3 & \le & & SO_3 & \text{times} & (s_1^2 - s_2 s_3)/S \\
q_2 + q_3 & \le & & (s_1 + s_2)O_3 & \text{times} & s_3 \\
q_3 & \le & & s_1 O_3 & \text{times} & s_2.
\end{array}$$

The lower bound. We put $q_k = s_{4-k}R/S$ and $O_k = (s_{4-k} + \cdots + s_3)R/S^2$. □

Conclusions. The main open problem is to find better bounds on the overall competitive ratio, and perhaps to find an explicit formula for further special cases. With the knowledge of the optimal algorithm, this "only" involves analyzing the linear program better. In general, the formula for optimal competitive ratio may need to have many cases. Still, it is plausible that a good overall bound can be proved with only a few upper bounds similar to the ones in Section 6.

Another question concerns the idle times in the schedule. Our algorithm relies on creating idle periods on some machines during the schedule. However, we are not able prove that idle periods are really necessary to achieve the optimal competitive ratio, even for any particular combination of speeds.

Acknowledgments. We are grateful to referees for many useful comments. T. Ebenlendr and J. Sgall supported by Institutional Research Plan No. AV0Z10190503, by Inst. for Theor. Comp. Sci., Prague (project 1M0545 of MŠMT ČR), and grant 201/05/0124 of GA ČR. W. Jawor supported by NSF grants CCF-0208856 and OISE-0340752.

References

1. S. Albers. On randomized online scheduling. In *Proc. 34th Symp. Theory of Computing (STOC)*, pages 134–143. ACM, 2002.
2. P. Berman, M. Charikar, and M. Karpinski. On-line load balancing for related machines. *J. Algorithms*, 35:108–121, 2000.
3. B. Chen, A. van Vliet, and G. J. Woeginger. Lower bounds for randomized online scheduling. *Inform. Process. Lett.*, 51:219–222, 1994.
4. B. Chen, A. van Vliet, and G. J. Woeginger. An optimal algorithm for preemptive on-line scheduling. *Oper. Res. Lett.*, 18:127–131, 1995.
5. T. Ebenlendr and J. Sgall. Optimal and online preemptive scheduling on uniformly related machines. In *Proc. 21st Symp. on Theoretical Aspects of Computer Science (STACS)*, LNCS 2996, pages 199–210. Springer, 2004.
6. L. Epstein. Optimal preemptive scheduling on uniform processors with non-decreasing speed ratios. *Oper. Res. Lett.*, 29:93–98, 2001.
7. L. Epstein, J. Noga, S. S. Seiden, J. Sgall, and G. J. Woeginger. Randomized on-line scheduling for two related machines. *J. Sched.*, 4:71–92, 2001.
8. L. Epstein and J. Sgall. A lower bound for on-line scheduling on uniformly related machines. *Oper. Res. Lett.*, 26(1):17–22, 2000.

9. R. Fleischer and M. Wahl. On-line scheduling revisited. *J. Sched.*, 3:343–353, 2000.
10. T. F. Gonzales and S. Sahni. Preemptive scheduling of uniform processor systems. *J. ACM*, 25:92–101, 1978.
11. R. L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical J.*, 45:1563–1581, 1966.
12. E. Horwath, E. C. Lam, and R. Sethi. A level algorithm for preemptive scheduling. *J. ACM*, 24:32–43, 1977.
13. J. F. Rudin III. *Improved Bound for the Online Scheduling Problem.* PhD thesis, The University of Texas at Dallas, 2001.
14. J. Sgall. A lower bound for randomized on-line multiprocessor scheduling. *Inform. Process. Lett.*, 63:51–55, 1997.
15. T. Tichý. Randomized on-line scheduling on 3 processors. *Oper. Res. Lett.*, 32: 152–158, 2004.
16. J. Wen and D. Du. Preemptive on-line scheduling for two uniform processors. *Oper. Res. Lett.*, 23:113–116, 1998.

On the Complexity of the Multiplication Method for Monotone CNF/DNF Dualization

Khaled M. Elbassioni

Max-Planck-Institut für Informatik, Saarbrücken, Germany
elbassio@mpi-sb.mpg.de

Abstract. Given the irredundant CNF representation ϕ of a monotone Boolean function $f : \{0,1\}^n \mapsto \{0,1\}$, the dualization problem calls for finding the corresponding unique irredundant DNF representation ψ of f. The (generalized) multiplication method works by repeatedly dividing the clauses of ϕ into (not necessarily disjoint) groups, multiplying-out the clauses in each group, and then reducing the result by applying the absorption law. We present the first non-trivial upper-bounds on the complexity of this multiplication method. Precisely, we show that if the grouping of the clauses is done in an output-independent way, then multiplication can be performed in sub-exponential time $(n|\psi|)^{O(\sqrt{|\phi|})}|\phi|^{O(\log n)}$. On the other hand, multiplication can be carried-out in quasi-polynomial time $\text{poly}(n,|\psi|)\cdot|\phi|^{o(\log|\psi|)}$, provided that the grouping is done depending on the intermediate outputs produced during the multiplication process.

1 Introduction

Let $f : \{0,1\}^n \mapsto \{0,1\}$ be a *monotone* Boolean function, defined by its *irredundant conjunctive normal form* (CNF)

$$\phi(x) = \bigwedge_{C\in\mathcal{C}} \bigvee_{i\in C} x_i,$$

where $\mathcal{C}$ is the set of clauses (*prime implicates*) of ϕ, each represented by the indices of the variables it contains. Such a CNF representation exists and is uniquely defined for a monotone Boolean function. The well-known *monotone Boolean dualization problem* is to find the corresponding *irredundant disjunctive normal form* (DNF) representation of f:

$$\psi(x) = \phi^*(x) \stackrel{\text{def}}{=} \bigvee_{D\in\mathcal{D}} \bigwedge_{i\in D} x_i.$$

Equivalently, the problem is to find, for an explicitly given hypergraph $\mathcal{H} \subseteq 2^V$, the transversal hypergraph, consisting of all minimal subsets of V hitting every hyperedge of $\mathcal{H}$.

This problem has received considerable attention in the literature (see e.g. [3,12,13,25,27]), since it is known to be polynomially equivalent with many other

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 340–351, 2006.

problems appearing in various areas, such as artificial intelligence (e.g. [12,20]), database theory (e.g. [26]), distributed systems (e.g. [16,18]), machine learning and data mining (e.g. [1,17]), mathematical programming (e.g. [5,21]), matroid theory (e.g. [22]), and reliability theory (e.g. [8,29]).

Clearly, the size of ψ can be exponential in n and the size (the number of clauses) of ϕ, and hence one can only hope for an algorithm whose efficiency is measured in terms of these parameters n, $|\phi|$ and $|\psi|$. Fredman and Khachiyan (1996) established the remarkable result that the monotone dualization problem can be solved in quasi-polynomial time $O(nN) + N^{o(\log N)}$, where $N = |\phi| + |\psi|$, thus putting the problem somewhere between polynomiality and NP-completeness [15]. They achieved this by presenting a quasi-polynomial time algorithm for the decision-version of the problem: given two monotone Boolean formulae ϕ and ψ in CNF and DNF forms respectively, is $\phi \equiv \psi$? Furthermore, for several special classes of monotone formulae ϕ, the problem is known to be solvable in polynomial time, e.g. when every clause has bounded-size [7,9,12,19], when every variable has bounded degree [10,13,28], when clauses have bounded intersection-size [4], for read-once formulae [11], etc.

An elementary folklore method, sometimes known as *multiplication* (see e.g. [2, Page 52]), works in its simplest form by traversing the clauses in some order, say $i = 1, \ldots m = |\phi|$, multiplying-out clause C_i with the result obtained for $C_1 \wedge \ldots \wedge C_{i-1}$, and simplifying using *the absorption law* whenever possible (that is using the identity $x \vee (x \wedge y) = x$ valid for all Boolean x, y). It is not difficult to come up with examples for which this method exhibits an *exponential blow-up* in the input-output size, e.g. the intermediate outputs have exponential size, while the final output is polynomially-bounded. Consider for instance, the CNF $\phi = \bigwedge_{1 \leq i,j \leq n} (x_i \vee y_j)$ on the set of $2n$ variables $\{x_1, \ldots, x_n, y_1, \ldots, y_n\}$. One can easily check that the corresponding DNF is $(x_1 \wedge \ldots \wedge x_n) \vee (y_1 \wedge \ldots \wedge y_n)$. On the other hand, if we start by multiplying the clauses $(x_1 \vee y_1), \ldots, (x_n \vee y_n)$, then we get 2^n clauses, which will be canceled-out later in the process. More interestingly, Takata [30] gave an example for which the multiplication method exhibits a *superpolynomial* blow-up, under *any ordering* of the clauses of the input CNF.

In a more general setting, one can perform multiplication as follows (see e.g. [30]). Write $\phi = \phi_1 \wedge \ldots \wedge \phi_k$, where each ϕ_i is a conjunction of a subset of clauses of ϕ, then multiply independently the individual ϕ_i's to obtain DNF's $\psi_1, \ldots \psi_k$. Finally, multiply-out $\psi_1, \ldots, \psi_k$ to obtain the final result ψ. More formally, given the input CNF formula ϕ, we build a tree $\mathbf{T}$ each node w of which is associated with an equivalent pair of a monotone CNF $\phi(w)$ and a monotone DNF $\psi(w)$ defined as follows:

(I) if w is a leaf then $\phi(w) = \psi(w)$ is an individual clause of ϕ and every clause of ϕ appears in *at least* one leaf of $\mathbf{T}$;

(II) if w is an internal node with children $w_1, \ldots, w_k$, then $\phi(w) = \phi(w_1) \wedge \ldots \wedge \phi(w_k)$ is the conjunction of the subset of clauses of ϕ appearing in the leaves of the subtree of $\mathbf{T}$ rooted at w, and $\psi(w) = \phi^*(w)$ is the DNF representation of $\psi(w_1) \wedge \ldots \wedge \psi(w_k)$.

In particular, at the root r of $\mathbf{T}$, $\psi = \psi(r)$ will contain the required DNF representation of ϕ. For this method to be efficient, one further requires at each node w of $\mathbf{T}$ that:

(III) the size of the intermediate output at w is not too large: $|\psi(w)| \leq \rho(n, |\phi|, |\psi|)$, for some polynomial ρ;

(IV) the DNF $\psi(w) = \psi(w_1) \wedge \ldots \wedge \psi(w_k)$ can be computed by a "trivial" procedure[1], in time $\tau(w)$, polynomial (or sub-exponential) in n, $|\phi|$ and $|\psi|$, given the individual outputs $\psi(w_1), \ldots, \psi(w_k)$ at the children of w.

For any such tree $\mathbf{T}$, denote by $N(\mathbf{T})$ the the number of nodes in $\mathbf{T}$, by $d(\mathbf{T})$ the depth of $\mathbf{T}$, and by $\tau(\mathbf{T})$ the maximum of $\tau(w)$ over all nodes w of $\mathbf{T}$. In what follows, we shall distinguish between two types of multiplication trees. An *output-independent* tree is one in which the selection of the sub-formulae $\phi(w_1), \ldots, \phi(w_k)$, at any node w with children $w_1, \ldots w_k$, is performed independent of the intermediate outputs $\psi(w_1), \ldots, \psi(w_k)$, computed at that node, i.e. the tree is constructed completely from the input formula ϕ, regardless of the intermediate outputs. In an *output-sensitive* tree, on the other hand, the intermediate output at each node affects the decision on how to split the clauses among the children of that node. To measure the dependence between the children of a node w of $\mathbf{T}$ in this case, we let $\mu(w) = \sum_{w'} |\phi(w')|/|\phi(w)|$, where the sum is over all the children w' of w which are dependent. We will denote by $\mu(\mathbf{T})$ the maximum of $\mu(w)$ over all the nodes w of $\mathbf{T}$. This distinction will appear important when we discuss the parallel and space complexity of the multiplication method. With the above notation, we have the following statements (the space bound follows by techniques similar to that of [31]; see [14] for more details).

Proposition 1. *Let ϕ be a monotone CNF formula on n variables:*

(i) If $\mathbf{T}$ is a multiplication tree for ϕ then the corresponding DNF $\psi = \phi^*$ can be computed in time $O(N(\mathbf{T})\tau(\mathbf{T}))$, using $O(d(\mathbf{T})\mu(\mathbf{T})^{d(\mathbf{T})}|\phi|n)$ space.

(ii) If $\mathbf{T}$ is an output-independent multiplication tree for ϕ then the corresponding DNF $\psi = \phi^*$ can be computed in parallel time $O(d(\mathbf{T})\Delta(\mathbf{T}))$ using $O(N(\mathbf{T})\Pi(\mathbf{T}))$ processors, where $\Delta(\mathbf{T})$ and $\Pi(\mathbf{T})$ are respectively the maximum parallel time and number of processors required by the trivial multiplication procedure in (IV) above at any node of $\mathbf{T}$.

The main results of this paper are the following.

Theorem 1. *Let ϕ be a monotone CNF formula on n variables:*

(i) There exists an output-sensitive multiplication tree $\mathbf{T}$ satisfying properties (I)-(IV) above, with $N(\mathbf{T}) = |\phi|^{o(\log|\psi|)}$, $d(\mathbf{T}) = o(\log|\phi|\log|\psi|)$, $\tau(\mathbf{T}) = \mathrm{poly}(n, |\phi|, |\psi|)$, and $\mu(\mathbf{T}) \leq 1$.

(ii) There exists an output-independent multiplication tree $\mathbf{T}$ satisfying properties (I)-(IV) above, with $N(\mathbf{T}) = n^{O(\sqrt{|\phi|}\log|\phi|)}|\phi|^{O(\log n)}$, and $d(\mathbf{T}) = O(\sqrt{|\phi|}\log|\phi| + \log n)$.

[1] i.e. one that uses only left-to-right multiplication, followed by absorption.

As we shall see later, these trees can be found efficiently, and moreover in (ii) we will have $\Delta(\mathbf{T}) = \text{polylog}(n, |\phi|, |\psi|)$ and $\Pi(\mathbf{T}) = |\psi|^{O(\sqrt{|\phi|})}\,\text{poly}(n, |\phi|)$.

Corollary 1. *Given a monotone CNF formula ϕ on n variables, the corresponding DNF $\psi = \phi^*$ can be computed*

(i) in time poly$(n, |\phi|)|\phi|^{o(\log|\psi|)}$, using polynomial space $\text{poly}(n, |\phi|)$*;*

(ii)in parallel time $\sqrt{|\phi|}\,\text{polylog}(n, |\phi|, |\psi|)$, using $(n|\psi|)^{O(\sqrt{|\phi|}\log|\phi|)}|\phi|^{O(\log n)}$ processors.

Remark. All the above bounds remain valid, if we require, instead of outputting the whole DNF ϕ^*, a sub-formula of a prescribed size ψ of ϕ^*.

2 Notation and an Outline of the Approach

Let $\phi = \phi(x_1, \ldots, x_n)$ be a monotone CNF (DNF) formula. We denote by $V(\phi)$ the set of variables appearing in ϕ and by $\mathcal{S}(\phi) \subseteq 2^{V(\phi)}$ its set of clauses (terms), where we identify each clause (term) $C \in \mathcal{C}$ of ϕ with the index set $C \subseteq V(\phi)$ of the variables that it contains. We shall assume that the given CNF ϕ is *irredundant*, i.e. for all $C, C' \in \mathcal{S}(\phi)$, $C \subseteq C'$ implies that $C = C'$. If ϕ is a monotone CNF formula, we denote by ϕ^* the irredundant DNF formula representing the same monotone Boolean function as ϕ.

For a subset $S \subseteq [n]$ of variables, denote by ϕ_S the CNF formula obtained form ϕ by fixing $x_i = 1$ for all $i \in \bar{S} \stackrel{\text{def}}{=} [n] \setminus S$. Equivalently, $\phi_S = \bigwedge_{C \in \mathcal{S}(\phi),\ C \subseteq S} \bigvee_{i \in C} x_i$. For $i \in [n]$, we let $\deg_\phi(i) = |\{C \in \mathcal{S}(\phi)\ :\ i \in C\}|$ be the degree of x_i in ϕ, and for a positive number $\epsilon \in (0,1)$, we let $L = L(\phi, \epsilon) \stackrel{\text{def}}{=} \{i \in [n] :\ \deg_\phi(i) > \epsilon|\phi|\}$, correspond to the subset of "high" degree variables with respect to ϕ.

Given $\epsilon', \epsilon'' \in (0,1)$, let us call any subset of variables $S \subseteq [n]$, such that $\epsilon'|\phi| \le |\phi_S| \le \epsilon''|\phi|$, an (ϵ', ϵ'')-*balanced set* with respect to ϕ.

Proposition 2. *Let $\epsilon_1, \epsilon_2 \in (0,1)$ be two given numbers such that, $\epsilon_1 < \epsilon_2$ and $L = L(\phi, \epsilon_1)$ satisfies $|\phi_L| \le (1-\epsilon_2)|\phi|$. Then there exists a $(1-\epsilon_2, 1-(\epsilon_2-\epsilon_1))$-balanced set $L' \supseteq L$.*

Proof. Such a set L' can be found as follows. Write $\bar{L} = \{i_1, \ldots, i_l\}$ and find the index $j \in [l-1]$, such that

$$|\phi_{[n]\setminus\{i_1,\ldots,i_j\}}| > (1-\epsilon_2)|\phi| \text{ and } |\phi_{[n]\setminus\{i_1,\ldots,i_{j+1}\}}| \le (1-\epsilon_2)|\phi|. \tag{1}$$

The existence of such j is guaranteed by the facts that $\deg_\phi(i_1) \le \epsilon_1|\phi| < \epsilon_2|\phi| \le |\phi| - |\phi_L|$. Finally, we let $L' = [n] \setminus \{i_1, \ldots, i_j\}$. Since $\deg_\phi(i_{j+1}) \le \epsilon_1|\phi|$, it follows from (1) that $|\phi_{L'}| < (\epsilon_1 + 1 - \epsilon_2)|\phi|$, implying that L' is indeed a balanced superset of L. □

We use the following general framework for building the multiplication trees with the properties stated above. Let L be the set of "large" degree variables of the

input CNF ϕ. If L contains at least one clause of ϕ, then we can decompose the problem by picking each variable x_i in L, and solving the subproblem on the sub-formula of ϕ avoiding x_i, which is of reasonably small size, since x_i has large degree. Otherwise, L admits a balanced superset $L' \supseteq L$, and we can decompose ϕ into two sub-formulas on L' and $\bar{L}'$, each of considerably smaller size, because L' is balanced. This will essentially lead to a multiplication tree with the properties given in Theorem 1-(i). To get part (ii) of the theorem, we apply recursion only until we get a set L of large degree variables for which $|\phi_L| = 0$. At this point, we observe that, since the degrees of the variables become sufficiently small, we can switch to the multiplication algorithm of [6], which works for bounded degree CNF's.

3 Output-Sensitive Multiplication Algorithm

In the sequel we let ϕ be a monotone CNF formula on n variables. When referring to a DNF expression, it is assumed implicitly that the expression is irredundant, in the sense that, the absorption law has been applied whenever possible. Following [23], we call a family of subsets $\{S_1, \ldots, S_r\} \subseteq 2^{V(\phi)}$ *complete* for ϕ, if for every clause C of ϕ, there exists an index $i \in \{1, \ldots, r\}$ such that $S_i \supseteq C$.

Proposition 3 ([23]). *Let $\{S_1, \ldots, S_r\} \subseteq 2^{V(\phi)}$ be a complete family of subsets for ϕ. Then, $\phi^* = \bigwedge_{i=1}^{r} \phi^*_{S_i}$.* □

Proposition 4 ([24]). *For any $S \subseteq [n]$, $|\phi^*_S| \leq |\phi^*|$.*

Let ϵ_1, ϵ_2 be constants in $(0, 1)$ to be selected later. Consider the multiplication function DNF-S shown in Figure 1. It is clear that the multiplication tree constructed by this procedure is exactly the recursion tree traversed by the procedure, and thus the depth and the number of nodes of this tree correspond respectively to the depth of the recursion and the total number of recursive calls plus the number of leaves.

Proposition 5. *Function DNF-S is correct: when run to termination on a monotone irredundant input CNF ϕ, it returns all the terms of the corresponding DNF ϕ^* without repetition.*

Proof. The statement follows by induction on the size of ϕ. If $|\phi| = 1$, the DNF representation of ϕ is ϕ itself and is returned in Step 1. Otherwise, let L be the set of variables computed in Step 2. Assume first that the function exists at Step 4. Note that the family $\{L\} \cup \{V(\phi) \setminus i \ : \ i \in L\}$ is complete for ϕ (since ϕ is irredundant and $|\phi_L| \geq 1$). Thus Proposition 3 implies in this case that the output returned at Step 4 is correct. Finally, assume that the function exits at Step 8. Then we need to show that the family $\{L\} \cup \{\bar{Y} \ : \ Y \in \mathcal{S}(\phi^*_L)\}$ is a complete set for ϕ. Let $C \in \mathcal{S}(\phi)$ be a clause of ϕ. If $C \cap Y \neq \emptyset$ for all $Y \in \mathcal{S}(\phi^*_L)$, then C must contain some clause $C' \in \mathcal{S}(\phi_L)$. But since ϕ is irredundant and $C, C' \in \mathcal{S}(\phi)$, we conclude that $C = C'$, i.e. $C \subseteq L$. □

Function $(\phi)^*$**:**
Input: A monotone CNF ϕ, *Output:* the corresponding DNF ϕ^*.

1. **if** $|\phi| = 1$, **then return** ϕ.
2. $L{:=}\ L(\phi, \epsilon_1)$.
3. If $|\phi_L| \geq 1$, **then**
4. **return** $\left(\bigvee_{i \in L} x_i\right) \wedge \left(\bigwedge_{i \in L} \phi^*_{V(\phi) \setminus i}\right)$.
5. **else**
6. $L{:=}$ a $(1 - \epsilon_2, 1 - (\epsilon_2 - \epsilon_1))$-balanced superset of L; (c.f. Proposition 2)
7. $\psi{:=}\ \phi^*_L$;
8. **return** $\psi \wedge \left(\bigwedge_{Y \in \mathcal{S}(\psi)} \phi^*_Y\right)$.

Fig. 1. Function DNF-S

Next we show that the multiplication tree traversed by function DNF-S satisfies properties (III) and (IV), stated in the introduction. We shall make use of the following statement.

Proposition 6. *Let $\mathcal{H} \subseteq 2^{[n]}$ be a family of subsets of variables, and ϕ be a monotone CNF, such that*

$$\phi(x) \leq \bigvee_{H \in \mathcal{H}} \bigwedge_{i \in H} x_i \quad \textit{for all } x \in \{0,1\}^n. \tag{2}$$

Then

$$\phi(x) \equiv \psi(x) \stackrel{\text{def}}{=} \bigvee_{H \in \mathcal{H}} \left(\left(\bigwedge_{i \in H} x_i \right) \wedge \phi^*_{\bar{H}}(x) \right).$$

Proof. If $x \in \{0,1\}^n$ satisfies $\psi(x) = 1$ then there exists $H \in \mathcal{H}$ such that $x_i = 1$ for all $i \in H$ and $\phi_{\bar{H}}(x) = \phi^*_{\bar{H}}(x) = 1$. This readily implies $\phi(x) = 1$. On the other hand, if $\phi(x) = 1$, then by (2) there exists an $H \in \mathcal{H}$ such that $\bigwedge_{i \in H} x_i = 1$. Since $\phi(x) = 1$ implies $\phi_{\bar{H}}(x) = 1$, we get that $\psi(x) = 1$. □

Proposition 7. *Let w be a node of the multiplication tree $\mathbf{T}$ constructed by function DNF-S on input ϕ, let $w_1, \ldots, w_k$ be the children of w in $\mathbf{T}$, and let $\psi(v)$ be the corresponding DNF produced at node v of $\mathbf{T}$. Then (i) $|\psi(w)| \leq |\phi^*|$ and (ii) the product $\psi(w) = \psi(w_1) \wedge \ldots \wedge \psi(w_k)$ can be computed in time* $\mathrm{poly}(|\phi|, |\psi(w)|, n)$.

Proof. Part (i) follows immediately from Proposition 4 since at each node w, the input is the set of *all* clauses of ϕ contained in some subset $S \subseteq [n]$. For part (ii), we consider two cases:

(a) If the function returns at Step 4: the product can be computed by applying Proposition 6 with $\mathcal{H} = \{\{i\} : i \in L\}$ (which satisfies (2) since $|\phi_L| \geq 1$). This

means that the output in Step 4 is identically equal to the irredundant DNF representation of

$$\bigvee_{i\in L}\left(x_i \wedge \phi^*_{V(\phi)\setminus i}\right),$$

which can be computed in time $(\sum_{i\in L}|\phi^*_{V(\phi)\setminus i}|)^2 \le (n|\phi^*|)^2$.

(b) If the function returns at Step 8: the product can be computed by applying Proposition 6 with $\mathcal{H} = \mathcal{S}(\phi^*_L)$ (which again satisfies (2)), yielding the equivalent formula

$$\bigvee_{Y\in\mathcal{S}(\phi^*_L)}\left(\left(\bigwedge_{i\in Y} x_i\right)\wedge \phi^*_{\bar{Y}}(x)\right),$$

the irredundant DNF of which can be evaluated in time $(\sum_{Y\in\mathcal{S}(\phi^*_L)}|\phi^*_{\bar{Y}}|)^2 \le |\phi^*|^4$. □

For $a, b > 1$, let $c = c(a, b)$ be the unique positive root of the equation

$$2^{c+1}\left(a^{c/\log b} - 1\right) = 1. \tag{3}$$

Given $n, k \in \mathbb{R}_+$, we define $\epsilon = \epsilon(n, k)$ and $\Lambda = \Lambda(n, k)$, as

$$\epsilon = \frac{1}{2}\left[\frac{1}{2} + \left(\frac{2k}{n}\right)^{\frac{1}{\Lambda-1}}\right]^{-1}, \quad \Lambda = \frac{\log(2k/n)}{c(2n, 2k/n)} + 1, \tag{4}$$

if $2k > n$, and we set

$$\epsilon = \frac{1}{3}, \quad \Lambda = \frac{\log(2n)}{\log(3/2)} + 1, \tag{5}$$

otherwise. Note that $\Lambda > 1$ for $n \ge 1$ and hence $\epsilon \le 1/3$.

In function DNF-S, we use the thresholds $\epsilon_2 = 2\epsilon_1 = 2\epsilon$, where ϵ is given by (4) and (5). Note that the values of the thresholds are computed with respect to the *initial* values of $n = |V_\phi|$ and $m = |\phi|$. The value of $k = |\phi^*|$ can be estimated within a factor of 2, using standard *exponential search* (we run the procedure for each value of $k' = 1, 2, 4, \ldots$, and for each such value, if the procedure did not stop within the bounds stated in Lemma 1, we know that $k > k'$, and we consider the next k', etc).

Analysis of DNF-S. Let respectively $N(n, m, k)$ and $d(n, m, k)$ denote the number of nodes and depth of the multiplication tree constructed by DNF-S on an input instance of size $|\phi| = m$ whose corresponding output is of size $|\phi^*| = k$.

Step 4: Note that, for all $i \in L$, $\deg_\phi(i) > \epsilon_1|\phi|$. Thus $|\phi_{V(\phi)\setminus i}| \le (1-\epsilon_1)|\phi|$, yielding the recurrences

$$N(n, m, k) \le 1 + n \cdot N(n-1, (1-\epsilon_1)m, k),$$
$$d(n, m, k) \le 1 + d(n-1, (1-\epsilon_1)m, k).$$

Step 8: Note that, since L is balanced, we have $|\phi_L| \leq (1-(\epsilon_2-\epsilon_1))|\phi|$, and $|\phi_{\bar{Y}}| \leq |\phi| - |\phi_L| \leq \epsilon_2|\phi|$, for all $Y \in \mathcal{S}(\phi_L^*)$ (since any $C \in \mathcal{S}(\phi_{\bar{Y}})$ also satisfies $C \notin \mathcal{S}(\phi_L)$). Hence, we get the recurrences

$$N(n,m,k) \leq 1 + N(n-1,(1-(\epsilon_2-\epsilon_1))m,k) + k \cdot N(n-1,\epsilon_2 m,k),$$
$$d(n,m,k) \leq 1 + \max\{d(n-1,(1-(\epsilon_2-\epsilon_1))m,k), d(n-1,\epsilon_2 m,k)\}.$$

We write $N(m) = N(n,m,k)$, $d(m) = d(n,m,k)$, and observe that the above recurrences, together with our settings for ϵ_1 and ϵ_2, imply that

$$N(m) \leq 1 + n \cdot N((1-\epsilon)m) + k \cdot N(2\epsilon m), \tag{6}$$
$$d(m) \leq 1 + \max\{d((1-\epsilon)m), d(2\epsilon m)\}. \tag{7}$$

Lemma 1. *For $m \geq 1$, $N(m) \leq m^{\Lambda(n,k)}$ and $d(m) \leq \Lambda(n,k)\log m/\log(2n)+1$.*

Proof. We prove the statement by induction on $m \geq 1$, with the base case $m = 1$ being trivial. Clearly it is enough to prove this for (6) and (7).

Case 1. $n < 2k$: First we may verify that our setting (4) together with (3) imply that

$$n(1-\epsilon)^\Lambda + k(2\epsilon)^\Lambda = n(1-\epsilon)^{\Lambda-1} = 2k(2\epsilon)^{\Lambda-1} = \frac{1}{2}. \tag{8}$$

Now consider (6). We apply induction and use (8) to get

$$N(m) \leq 1 + \left[n(1-\epsilon)^\Lambda + k(2\epsilon)^\Lambda\right] m^\Lambda = 1 + \frac{1}{2}m^\Lambda \leq m^\Lambda,$$

for $m \geq 2$. (In fact ϵ was selected to minimize $n(1-\epsilon)^\Lambda + k(2\epsilon)^\Lambda$.)

Next consider (7): we get by induction and (8) that

$$\begin{aligned} d(m) &\leq 1 + \frac{\Lambda \log m}{\log(2n)} + 1 + \Lambda \cdot \max\left\{\frac{\log(1-\epsilon)}{\log(2n)}, \frac{\log(2\epsilon)}{\log(2n)}\right\} \\ &= 1 + \frac{\Lambda \log m}{\log(2n)} + 1 + \Lambda \cdot \max\cdot\left\{\frac{-1}{\Lambda-1}, \frac{-1}{\Lambda-1}\left(\frac{\log(4k)}{\log(2n)}\right)\right\} \\ &= 1 + \frac{\Lambda \log m}{\log(2n)} + 1 - \frac{\Lambda}{\Lambda-1} < \frac{\Lambda \log m}{\log(2n)} + 1. \end{aligned}$$

This gives the required bound on $d(m)$.

Case 2. $n \geq 2k$: First consider (6). Form (5), we get that $(3/2)^{\Lambda-1} = 2n$. Using $\epsilon = 1/3$, $k \leq n/2$, and induction, we get for $m \geq 2$

$$N(m) \leq 1 + \frac{3n}{2}N(\frac{2}{3}m) \leq 1 + \frac{3n}{2}\left(\frac{2}{3}\right)^\Lambda m^\Lambda = 1 + \frac{1}{2}m^\Lambda \leq m^\Lambda.$$

Now consider (7). Both terms of the maximum in (7) are equal to $d(\frac{2}{3}m)$ and hence we get by induction

$$d(m) \leq 1 + \frac{\Lambda \log m}{\log(2n)} + 1 + \frac{\Lambda \log(2/3)}{\log(2n)} = 1 + \frac{\Lambda \log m}{\log(2n)} + 1 - \frac{\Lambda}{\Lambda - 1} < \frac{\Lambda \log m}{\log(2n)} + 1.$$

□

Note that if $k > n/2$, then $\Lambda(n,k) \sim \log(2k/n)/\log\left(\frac{\log(2k/n)}{\log(2n)}\right)$. Furthermore, for any node w of the multiplication tree $\mathbf{T}$ constructed by function DNF-S, a subset of dependent children consists of two children w_1 and w_2 with $\phi(w_1) = \phi_L$ and $\phi(w_2) = \phi_{\bar{Y}}$, for a subset $L \subseteq V(\phi)$ and some set $Y \in \mathcal{S}(\phi^*_L)$. Noting that the clauses of ϕ_L and $\phi_{\bar{Y}}$ are disjoint (since $Y \cap C \neq \emptyset$ for all $C \in \mathcal{S}(\phi_L)$), we conclude that the total number of clauses allocated to two dependent children of Y is at most $|\phi|$, i.e. $\mu(\mathbf{T}) \leq 1$. This establishes the bounds stated in Theorem 1-(i).

Remark. If we replace in DNF-S, $\epsilon_1 < \epsilon_2$ by arbitrary constants in $(0,1)$, say $\epsilon_1 = 1/3$ and $\epsilon_2 = 2/3$, then we get $N(n,m,k) \leq (n+k)^{O(\log m)}$ and $d(n,m,k) = O(\log m)$.

4 Output-Independent Multiplication Algorithm

We first recall the following theorem from [6].

Theorem 2 ([6]). *Let $\phi : \{0,1\}^n \mapsto \{0,1\}$ be a monotone CNF, $h : [n] \mapsto \mathbb{Z}_+$ an integer-valued monotone* sub-additive *function*[2]*, and $\delta > 0$ be a given number, such that for all $X \subseteq [n]$,*

(i) $h(X) = 0$ implies that $|\phi_X| = 0$, and

(ii) $\frac{1}{h(X)} \sum_{C \in \mathcal{S}(\phi_X)} h(C) \leq \delta$.

Fix a constant $0 < \eta < 1$, and let $r = 1 + 2\delta/(1-\eta)$, and $\alpha = h([n])$. Then there exists an output-independent multiplication tree $\mathbf{T}$ for ϕ with depth $d(\mathbf{T}) \leq (\log \alpha)/\eta$, maximum degree at most r, and thus of size $N(\mathbf{T}) \leq r^{(\log \alpha)/\eta + 1}$.

Figure 2 gives an output-independent multiplication procedure. We use $\epsilon_1 = 1/\sqrt{|\phi|}$. As in DNF-S, whenever the set L of large degree variables contains a clause of ϕ, we recurse. However, if $|\phi_L|$ becomes empty we call DNF-D, which builds the multiplication tree guaranteed by Theorem 2. To apply the theorem, we use $h(X) = |X \cap \bar{L}|$ and $\delta = \epsilon_1 |\phi|$. It easy to check that both conditions (i) and (ii) in the statement of the theorem are satisfied. Consequently, we obtain the following statement, regarding the depth $d(m)$ and the number of nodes $N(m)$ of the tree constructed by DNF-I, where $m = |\phi|$.

Lemma 2. *For $m \geq 1$, $N(m) = n^{O(\sqrt{m} \log m)} m^{O(\log n)}$ and $d(m) = O(\sqrt{m} \log m + \log(n))$.*

[2] i.e. one for which $h(X \cup Y) \leq h(X) + h(Y)$ holds for all subsets $X, Y \subseteq [n]$.

Function $(\phi)^*$**:**
Input: A monotone CNF ϕ, *Output:* the corresponding DNF ϕ^*.

1. **if** $|\phi| = 1$, **then return** ϕ.
2. $L := L(\phi, \epsilon_1)$.
3. If $|\phi_L| \geq 1$, **then**
4. **return** $\left(\bigvee_{i \in L} x_i\right) \wedge \left(\bigwedge_{i \in L} \phi^*_{V(\phi) \setminus i}\right)$.
5. **else**
6. **return** DNF-D(ϕ).

Fig. 2. Function DNF-I

Proof. As long as $|\phi_L| \geq 1$ (in Step 3), the function recurses on $|L|$ formulae of size at most $(1 - \epsilon_1)|\phi|$ each. This gives the recurrences:

$$N(m) \leq 1 + n \cdot N((1 - \epsilon_1)m),$$
$$d(m) \leq 1 + d((1 - \epsilon_1)m).$$

Thus after at most $O(\log m/\epsilon_1) = O(\sqrt{m} \log m)$ steps, we either terminate or call the function DNF-D. By Theorem 2, the depth and number of nodes of each sub-tree constructed after that are respectively $O(\log n)$ and $m^{O(\log n)}$. The lemma follows. □

To conclude the proof of Theorem 1, part (ii), we observe that the product in Step 4 of function DNF-S can be computed in polynomial time (and also efficiently in parallel), see Proposition 7. Furthermore, since each node of the multiplication sub-trees constructed by DNF-D has degree at most $r = O(\sqrt{m})$, the product at such node can be done by trivial left-to-right multiplication in any order in time $k^{O(\sqrt{m})}$ (or in parallel time polylog(n, m, k) using $k^{O(\sqrt{m})}$ processors), where $k = |\phi^*|$.

References

1. M. Anthony and N. Biggs, *Computational Learning Theory*, Cambridge Univ. Press, 1992.
2. C. Berge, *Hypergraphs*, North Holland Mathematical Library, Vol. 445, 1989.
3. J. C. Bioch and T. Ibaraki, Complexity of identification and dualization of positive Boolean functions, *Information and Computation* 123 (1995), pp. 50–63.
4. E. Boros, K. Elbassioni, V. Gurvich and L. Khachiyan, Generating Maximal Independent Sets for Hypergraphs with Bounded Edge-Intersections, in *Proc. the 6th Latin American Theoretical Informatics Conference (LATIN 2004)*, *LNCS* 2976, pp. 488–498.
5. E. Boros, K. Elbassioni, V. Gurvich, L. Khachiyan and K. Makino, Dual-bounded generating problems: All minimal integer solutions for a monotone system of linear inequalities, *SIAM J. Comput.*, **31** (5) (2002) pp. 1624–1643.
6. E. Boros, K. Elbassioni, V. Gurvich and L. Khachiyan, Computing Many Maximal Independent Sets for Hypergraphs in Parallel, *DIMACS technical report 2004-44*, Rutgers University, (http://dimacs.rutgers.edu/TechnicalReports/2004.html).

7. E. Boros, V. Gurvich, and P.L. Hammer, Dual subimplicants of positive Boolean functions, *Optimization Methods and Software*, 10 (1998) pp. 147–156.
8. C. J. Colbourn, *The combinatorics of network reliability*, Oxford Univ. Press, 1987.
9. E. Dahlhaus and M. Karpinski, A fast parallel algorithm for computing all maximal cliques in a graph and the related problems, *Proc. 1st Scandinavian Workshop on Algorithm Theory (SWAT)* , Sweden, July 5-8, 1988, LNCS 318, pp. 139–144.
10. C. Domingo, N. Mishra and L. Pitt, Efficient read-restricted monotone CNF/DNF dualization by learning with membership queries, *Machine learning* 37 (1999) pp. 89–110.
11. T. Eiter, Exact Transversal Hypergraphs and Application to Boolean μ-Functions, *J. Symb. Comput.* 17(3) (1994) pp. 215–225.
12. T. Eiter and G. Gottlob, Identifying the minimal transversals of a hypergraph and related problems, *SIAM J. Comput.*, 24 (1995) pp. 1278–1304.
13. T. Eiter, G. Gottlob and K. Makino, New results on monotone dualization and generating hypergraph transversals, SIAM J. Comput. 32(2)(2003) pp. 514–537.
14. K. Elbassioni, On the complexity of monotone Boolean duality testing, *DIMACS Technical Report 2006-1*, Rutgers University (http://dimacs.rutgers.edu/TechnicalReports/2006.html).
15. M. L. Fredman and L. Khachiyan, On the complexity of dualization of monotone disjunctive normal forms. *J. Algorithms*, 21 (1996) pp. 618–628.
16. H. Garcia-Molina and D. Barbara, How to assign votes in a distributed system, *Journal of the ACM*, 32 (1985) pp. 841–860.
17. D. Gunopulos, R. Khardon, H. Mannila, and H. Toivonen, Data mining, hypergraph transversals and machine learning, in *Proc. the 16th ACM-SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 1997)*, pp. 12–15.
18. T. Ibaraki and T. Kameda, A theory of coteries: Mutual exclusion in distributed systems, *IEEE Transactions on Parallel and Distributed Systems*, 4 (1993) pp. 779–794.
19. D. S. Johnson, M. Yannakakis and C. H. Papadimitriou, On generating all maximal independent sets, *Info. Process. Lett.*, 27 (1988) pp. 119–123.
20. D. J. Kavvadias, C. Papadimitriou, and M. Sideri, On Horn envelopes and hypergraph transversals, in *Proceedings of the 4th International Symposium on Algorithms and Computation (ISAAC-93)*, LNCS 762, W. Ng, ed., Springer-Verlag, New York (1993) pp. 399–405.
21. L. Khachiyan, Transversal hypergraphs and families of polyhedral cones, in N. Hadjisavvas and P. Pardalos (Eds.), *Advances in Convex Analysis and Global Optimization, Honoring the memory of K. Carathéodory*, pp. 105–118, Kluwer Academic Publishers, Dordrecht/Boston/London, 2000.
22. L. Khachiyan, E. Boros, K. Elbassioni, V. Gurvich, and K. Makino, On the Complexity of Some Enumeration Problems for Matroids, SIAM J. Discrete Math. 19(4) (2005) pp. 966–984.
23. L. Khachiyan, E. Boros, K. Elbassioni, V. Gurvich, A New Algorithm for the Hypergraph Transversal Problem, in *the Proceedings of the Computing and Combinatorics, 11th Annual International Conference (COCOON 2005)*, Kunming, China, August 16-29, 2005, LNCS 3595, pp. 767–776.
24. E. Lawler, J. K. Lenstra and A. H. G. Rinnooy Kan, Generating all maximal independent sets: NP-hardness and polynomial-time algorithms, *SIAM J. Comput.*, 9 (1980) pp. 558–565.
25. L. Lovász, Combinatorial optimization: some problems and trends, DIMACS Technical Report 92-53, Rutgers University, 1992.

26. H. Mannila and K. J. Räihä, Design by example: An application of Armstrong relations, *Journal of Computer and System Science* 22 (1986) pp. 126–141.
27. C. Papadimitriou, NP-completeness: A retrospective, in Proceedings of the 24th International Colloquium on Automata, Languages, and Programming (ICALP '97), LNCS 1256, Springer-Verlag, New York, 1997, pp. 2–6.
28. N. Mishra and L. Pitt, Generating all maximal independent sets of bounded-degree hypergraphs, in Proceedings of the 10th Annual Conference on Computational Learning Theory (COLT), Nashville, TN, 1997, pp. 211–217.
29. K. G. Ramamurthy, *Coherent Structures and Simple Games*, Kluwer Academic Publishers, 1990.
30. K. Takata, On the sequential method for listing minimal hitting sets, in *Proc. SIAM Workshop on Discrete Mathematics and Data Mining (DM & DM)*, Arlington, VA, April 2002, pp. 109–120.
31. H. Tamaki, Space-efficient enumeration of minimal transversals of a hypergraph, IPSJ-AL **75** (2000) 29–36.

Lower and Upper Bounds on FIFO Buffer Management in QoS Switches*

Matthias Englert and Matthias Westermann

Department of Computer Science
RWTH Aachen, D-52056 Aachen, Germany
{englert, marsu}@cs.rwth-aachen.de

Abstract. We consider FIFO buffer management for switches providing differentiated services. In each time step, an arbitrary number of packets arrive, and only one packet can be sent. The buffer can store a limited number of packets, and, due to the FIFO property, the sequence of sent packets has to be a subsequence of the arriving packets. The differentiated service model is abstracted by attributing each packet with a value according to its service level. A buffer management strategy can drop packets. The goal is to maximize the sum of values of sent packets.

For only two different packet values, we introduce the account strategy and prove that this strategy achieves an optimal competitive ratio of ≈ 1.282, if the buffer size tends to infinity, and an optimal competitive ratio of $(\sqrt{13}-1)/2 \approx 1.303$, for arbitrary buffer sizes. For general packet values, the simple preemptive greedy strategy (PG) is studied. We show that PG achieves a competitive ratio of $\sqrt{3} \approx 1.732$ which is the best known upper bound on the competitive ratio of this problem. In addition, we give a lower bound of $1 + 1/\sqrt{2} \approx 1.707$ on the competitive ratio of PG which improves the previously known lower bound. As a consequence, the competitive ratio of PG cannot be further improved significantly.

1 Introduction

Quality of Service (QoS) guarantees for network services allow service providers to address the service requirements of customers by providing different levels of service. In the network setting, where traffic volumes may exceed network capacity, effective management of packets at buffers in switches is a key to achieving QoS guarantees. By differentiating service levels, packets of different types may be treated according to the level of service they require.

1.1 The Model

Time is slotted in time steps. In each time step, an arbitrary number of packets arrive, and, at the end of each time step, only one packet can be sent. Packets

* Supported by the DFG grant WE 2842/1.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 352–363, 2006.

that are not sent can be stored in a *FIFO buffer* with a limited storage capacity for b packets. Initially, the FIFO buffer is empty. Due to the FIFO property, the sequence of sent packets has to be a subsequence of the arriving packets, i.e., if a packet p is sent before a packet p', then p has arrived before p'.

The differentiated service model is abstracted by attributing each packet p with the *value* $v(p)$ according to its service level. A buffer management strategy can *drop* arriving packets, i.e., these packets are never stored in the buffer, or can *drop* packets stored in the buffer, i.e., these packets are deleted from the buffer and not sent. The goal of the buffer management strategy is to maximize the sum of the values of sent packets.

The notion of an online strategy is intended to formalize the realistic scenario where the strategy does not have knowledge about the whole input sequence of arriving packets in advance. The online strategy gets to know this sequence packet by packet, and has to react without knowledge about the future. Online strategies are typically evaluated in a competitive analysis. In this kind of analysis the total value produced by the online strategy is compared with the total value produced by an optimal offline strategy.

For a given input sequence σ of arriving packets, let $V(\sigma)$ denote the total value produced by an optimal offline strategy. An online strategy is denoted as *c-competitive*, if it produces total value at least $V(\sigma)/c - \kappa$, for each input sequence σ of arriving packets, where κ is a term that does not depend on σ. The value c is also called the *competitive ratio* of the online strategy.

1.2 Previous Work

Aiello et al. [1] introduce the model of differentiated services for FIFO buffers without preemption. Mansour, Patt-Shamir, and Lapid [11] add preemption and general packet values to this model. Kesselman and Mansour [8] study the value of the lost packets instead of the value of the sent packets.

Kesselman et al. [7] show that the greedy strategy achieves a competitive ratio of 2. Kesselman, Mansour, and van Stee [9] introduce the preemptive greedy strategy and prove that this strategy achieves a competitive ratio of ≈ 1.983. In addition, they give the previously best known lower bound of $(1+\sqrt{5})/2 \approx 1.618$ on the competitive ratio of the preemptive greedy strategy. Bansal et al. [5] study a modification of the preemptive greedy strategy and show that this strategy achieves a competitive ratio of $7/4$ which is the previously best known upper bound on the competitive ratio of this problem. Note that this modification does not improve the overall performance of the strategy [6]. The best known lower bound on the competitive ratio of this problem is ≈ 1.419 [9].

The following results refer to the case where only two different packet values are considered. Lotker and Patt-Shamir [10] present a strategy that achieves a competitive ratio of ≈ 1.30448. Kesselman et al. [7] show a lower bound of ≈ 1.281 on the competitive ratio. Andelman [2] presents a randomized strategy that achieves a competitive ratio of $5/4$. Further, he gives a lower bound of ≈ 1.197 on the competitive ratio of any randomized strategy.

Azar and Richter [4] extend the buffer management problem to multi-queues, i.e., several incoming queues have to be served by delivering packets that arrive at these queues through one output port, one packet per time step. They present a generic technique that transforms a strategy for a single queue to a strategy for several queues. They show that the competitive ratio of the constructed strategy is at most twice the competitive ratio of the single queue strategy.

1.3 Our Contributions

In Section 2, only two packet values are considered. We introduce the account strategy and prove that this strategy achieves an optimal competitive ratio of ≈ 1.282, if the buffer size tends to infinity, and an optimal competitive ratio of $(\sqrt{13}-1)/2 \approx 1.303$, for arbitrary buffer sizes. Note that this is the first non-trivial optimal result in this area.

In Section 3, general packet values are considered. We study the preemptive greedy strategy (PG) introduced in [9]. This is a simple strategy that can be implemented efficiently. We show that PG achieves a competitive ratio of $\sqrt{3} \approx 1.732$ which is the best known upper bound on the competitive ratio of this problem. In addition, we give a lower bound of $1 + 1/\sqrt{2} \approx 1.707$ on the competitive ratio of PG which improves the previously known lower bound of $(1+\sqrt{5})/2 \approx 1.618$. Hence, the gap between upper and lower bound for PG narrows to approximately $1/40$. We conjecture that the lower bound is tight. As a consequence, new approaches are needed, since the competitive ratio of PG cannot be further improved significantly. Based on our lower bound for PG and our optimal account strategy for two packet values, we propose an approach to tackle the problems of PG.

2 Two Packet Values

In this section, only two packet values 1 and $\alpha > 1$ are considered. A packet of value 1 is denotes as 1-packet, and a packet of value α is denotes as α-packet. Define

$$r := \frac{\sqrt{13}-1}{2} \approx 1.303 \quad \text{and}$$

$$r_\infty := \frac{46 + 32\sqrt{2} + 7\sqrt{2}\sqrt{10+8\sqrt{2}} + 10\sqrt{10+8\sqrt{2}}}{36 + 25\sqrt{2} + 6\sqrt{2}\sqrt{10+8\sqrt{2}} + 7\sqrt{10+8\sqrt{2}}} \approx 1.282 \ .$$

The following theorem states two lower bounds on the competitive ratio of any deterministic strategy. The proof for the first statement of this theorem can be found, e.g., in [3], and the proof for the second statement of this theorem can be found, e.g., in [7].

Theorem 1. *Consider only two packet values 1 and $\alpha > 1$.*

1. *The competitive ratio of any deterministic strategy is at least r, if the buffer size is 2.*

2. *The competitive ratio of any deterministic strategy is at least r_∞, if the buffer size tends to infinity.*

The *account strategy (ACC)* tries to preempt 1-packets from the buffer in order to avoid losing too many α-packets in case of a buffer overflow. The number of preempted 1-packets has to be chosen carefully. Obviously, the total number of preempted 1-packets should not exceed $(x-1)$ times the total value of sent packets, if a competitive ratio of x should be achieved. Hence, one basic idea of ACC is to preempt at most $(x-1)\cdot\alpha$ 1-packets for each α-packet entering the buffer and at most $(x-1)$ 1-packets for each sent 1-packet. ACC tries to preempt as much 1-packets as possible without violating this constraint.

We define ACC(x) with one parameter $x \geq 1$ which is the competitive ratio we aim for and which is therefore used to determine how aggressive the strategy is w.r.t. preemption. ACC(x) uses an *account* a which is initially set to 0. Basically, each packet sent by ACC(x) increases the account by $(x-1)$ times its own value, and each preempted 1-packet decreases the account by 1. More precisely, for each time step, ACC(x) does the following.

1. For each arriving packet p, do the following.
   (a) If there is an unoccupied location in the buffer, store p in the buffer. Otherwise, if p is an α-packet, drop the first (i.e., closest to the front of the buffer) packet p' with the smallest value among the packets in the buffer, and store p in the buffer.
   (b) If p is an α-packet (observe that arriving α-packets are always stored in the buffer) and has not ejected another α-packet, the account a is increased by $(x-1)\cdot\alpha$. The first $\lfloor a \rfloor$ 1-packets in the buffer are dropped (if there are less, all 1-packets are dropped), and the account a is decreased by the number of dropped 1-packets.
   (c) If the buffer is completely filled with α-packets, the account a is reset to 0.
2. After all packets have arrived and a packet p has been sent, do the following. If p is a 1-packet, the account a is increased by $(x-1)$. The first $\lfloor a \rfloor$ 1-packets in the buffer are dropped (if there are less, all 1-packets are dropped), and the account a is decreased by the number of dropped 1-packets.

Note that ACC(x) does not require any knowledge about the two packet values in advance.

The following theorem shows that ACC(x) achieves optimal competitive ratios. The proof of this theorem is omitted due to space limitations.

Theorem 2. *Consider only two packet values 1 and $\alpha > 1$.*

1. *ACC(r) achieves a competitive ratio of r, for arbitrary buffer sizes.*
2. *ACC(r_∞) achieves a competitive ratio of r_∞, if the buffer size tends to infinity.*

3 The Preemptive Greedy Strategy

Kesselman, Mansour, and van Stee [9] introduce the preemptive greedy strategy (PG) with the parameter $\beta > 1$. When a packet p arrives, PG does the following.

1. Find the first (i.e., closest to the front of the buffer) packet p', with $v(p') \leq v(p)/\beta$. If such a packet p' exists, drop it (p' is denoted as *preempted* by p).
2. If there is an unoccupied location in the buffer, store p in the buffer.
3. Otherwise, drop a packet p' with the smallest value among the packets in the buffer, if $v(p') < v(p)$ (p' is denoted as *ejected* by p). If a packet is dropped, store p in the buffer.

Bansal et al. [5] study a modified version of PG. The only difference is that step 1 of PG is substituted by the following.

1. Find the first (i.e., closest to the front of the buffer) packet p', with $v(p') \leq v(p)/\beta$ and $v(p')$ is not larger than the value of the packet stored after p' in the buffer. If such a packet exists, drop it.

Note that this modification does not improve the overall performance of the strategy [6].

New approaches are needed, since, due to the following lower and upper bound, the competitive ratio of PG cannot be further improved significantly. A basic concept of PG is that, for each arriving packet p, the first packet whose value is at most $v(p)/\beta$ is preempted. At first sight, it seems more reasonable that, instead, the packet with the smallest value is preempted. But in fact, the preemption of the first packet whose value is suitable small enough is a crucial property to achieve a competitive ratio smaller than 2. However, this can turn out to be a great disadvantage as the first input sequence in the following lower bound shows. This disadvantage diminishes with increasing β. On the other hand, too few packets are preempted for larger β as the second input sequence in the following lower bound shows. An approach to tackle this problem might be the following: If, for large β, the value of a single packet does not suffice to preempt another packet, the values of more than one packet are combined for preemption. Note that, in the case of only two packet values, we achieve with this idea an optimal strategy.

3.1 Lower Bound

The following theorem gives an lower bound on the competitive ratio of PG.

Theorem 3. *The competitive ratio of PG is at least* $1 + 1/\sqrt{2} \approx 1.707$.

Proof. Fix an even buffer size b. Depending on β, we distinguish the following two cases.

– Suppose that $\beta \leq 2 + \sqrt{2}$.
The input sequence consists of n consecutive phases defined as follows.

- Phase $1 \leq i < n$ consists of $b/2$ time steps. In time step 1, at first b packets of value ε and finally $b/2$ packet of value β^i arrive. In the remaining $b/2 - 1$ time steps, new packets do not arrive.
- Phase n consists of one time step. In this time step, b packets of value β^{n-1} arrive.

For this input sequence, PG produces value

$$\lim_{\varepsilon \to 0} \sum_{i=1}^{n-1} \left(\frac{b}{2} \cdot \varepsilon \right) + b \cdot \beta^{n-1} = b \cdot \beta^{n-1} \ ,$$

and the optimal value is

$$\sum_{i=1}^{n-1} \left(\frac{b}{2} \cdot \beta^i \right) + b \cdot \beta^{n-1} = b \cdot \frac{3\beta^n - 2\beta^{n-1} - \beta}{2(\beta - 1)} \ .$$

Hence, the competitive ratio is

$$\lim_{n \to \infty} \frac{3\beta^n - 2\beta^{n-1} - \beta}{2(\beta^n - \beta^{n-1})} = 1 + \frac{\beta}{2(\beta - 1)} \geq 1 + \frac{1}{\sqrt{2}} \ .$$

– Suppose that $\beta > 2 + \sqrt{2}$.
The input sequence consists of n consecutive phases defined as follows.

- Phase 1 consists of $b - 1$ time steps. In time step 1, at first $b - 1$ packets of value 1 and finally one packet of value $\alpha < \beta$ arrive. In each of the remaining $b - 2$ time steps, one packet of value α arrives.
- Phase $1 < i < n$ consists of $b - 1$ time steps. In each of these time steps, one packet of value α^i arrives.
- Phase n consists of one time step. In this time step, b packets of value α^{n-1} arrive.

For this input sequence, PG produces value

$$\sum_{i=0}^{n-2} ((b-1) \cdot \alpha^i) + b \cdot \alpha^{n-1} = b \cdot \frac{\alpha^n - \frac{1}{b} \cdot \alpha^{n-1} - \frac{b-1}{b}}{\alpha - 1} \ ,$$

and the optimal value is

$$\sum_{i=1}^{n-1} ((b-1) \cdot \alpha^i) + b \cdot \alpha^{n-1} = b \cdot \frac{\left(2 - \frac{1}{b}\right) \cdot \alpha^n - \alpha^{n-1} - \frac{b-1}{b} \cdot \alpha}{\alpha - 1} \ .$$

Hence, the competitive ratio is

$$\lim_{\alpha \to \beta} \lim_{n \to \infty} \lim_{b \to \infty} \frac{\left(2 - \frac{1}{b}\right) \cdot \alpha^n - \alpha^{n-1} - \frac{b-1}{b} \cdot \alpha}{\alpha^n - \frac{1}{b} \cdot \alpha^{n-1} - \frac{b-1}{b}}$$
$$= \lim_{\alpha \to \beta} \lim_{n \to \infty} \frac{2\alpha^n - \alpha^{n-1} - \alpha}{\alpha^n - 1} = \lim_{\alpha \to \beta} \frac{2\alpha - 1}{\alpha} = 1 + \frac{\beta - 1}{\beta} \geq 1 + \frac{1}{\sqrt{2}} \ .$$

This concludes the proof of the theorem. □

3.2 Upper Bound

The following theorem gives an upper bound on the competitive ratio of PG.

Theorem 4. *PG achieves a competitive ratio of* $\sqrt{3} \approx 1.732$ *for* $\beta = 2 + \sqrt{3}$.

Proof. Let OPT denote an optimal offline strategy. W.l.o.g. we assume that, at the arrival of each packet, the buffer of PG is completely filled with packets. If there are unoccupied locations in the buffer of PG, it is assumed that dummy packets of value 0 are stored at these locations which are always at the end of the buffer. Hence, each arriving packet either preempts another packet, causes the ejection of another packet, or is not stored in the buffer of PG.

Fix an input sequence of arriving packets. This input sequence can also be regarded as a sequence $\sigma = \sigma_1\sigma_2\cdots$ of arrival and send events, where each arrival of a new packet corresponds to an arrival event and each sending of a packet corresponds to a send event. The event sequence σ is partitioned into time steps, where the first time step starts with the first event and a new time step starts right after each send event.

Let S_t^{pg} (S_t^{opt}) denote the set of packets sent by PG (OPT) by the end of event σ_t. Let B_t^{pg} (B_t^{opt}) denote the set of packets stored in the buffer of PG (OPT) at the end of σ_t. For a packet $p \in B_t^{\text{pg}}$, we call $c_t(p)$ the *charge* of p at the end of σ_t. Further, we call $D_t \subseteq B_t^{\text{opt}} \setminus B_t^{\text{pg}}$ the set of packets with a *deposit* at the end of σ_t. Initially, $D_0 := \emptyset$. The goal is to choose $c_t(p)$ and D_t in such a way that, for each event σ_t, the main inequality

$$\sum_{p \in S_t^{\text{pg}}} r \cdot v(p) + \sum_{p \in B_t^{\text{pg}}} c_t(p) \geq \sum_{p \in S_t^{\text{opt}} \cup D_t} v(p)$$

is true, with $r := \sqrt{3}$. As a consequence, this yields the theorem.

Let Δ_t^{pg} (Δ_t^{opt}) denote the alterations of the left (right) side of the main inequality at the event σ_t, i.e.,

$$\Delta_t^{\text{pg}} := \sum_{p \in S_t^{\text{pg}} \setminus S_{t-1}^{\text{pg}}} r \cdot v(p) + \sum_{p \in B_t^{\text{pg}}} c_t(p) - \sum_{p \in B_{t-1}^{\text{pg}}} c_{t-1}(p) \quad \text{and}$$

$$\Delta_t^{\text{opt}} := \sum_{p \in S_t^{\text{opt}} \setminus S_{t-1}^{\text{opt}}} v(p) + \sum_{p \in D_t} v(p) - \sum_{p \in D_{t-1}} v(p) \ .$$

Obviously, the main inequality is true before the first event, since packets have not been sent so far and the buffers and the set of packets with a deposit are empty. Hence, it is sufficient to show, for each event σ_t, $\Delta_t^{\text{pg}} \geq \Delta_t^{\text{opt}}$, since this yields the main inequality.

First, an intuition for the basic ideas of the proof is given. Then, the formal proof is presented. The basic idea for the set $D_t \subseteq B_t^{\text{opt}} \setminus B_t^{\text{pg}}$ is simple. Packets stored exclusively in the buffer of OPT at the end of event σ_t, especially packets already sent by PG, could be a problem, if PG cannot sent a packet, i.e., the buffer of PG is empty, when those packets are sent by OPT. The left

$s_t(p)$	$c_t(p)$	comment
BC	$(r-2)\cdot v(p)$	buddy with credit
B	0	buddy
U	$(r/\beta)\cdot v(p) + (2-r)\cdot v_t^{\min}(p)$	unproblematic
E	$(r-1)\cdot v(p)$	exclusively in B_t^{pg}, i.e., not in B_t^{opt}
EB	$2(r-1)\cdot v(p)$	exclusively in B_t^{pg} with buddy

Fig. 1. Definition of the charge $c_t(p)$ of a packet $p \in B_t^{\mathrm{pg}}$ at the end of event σ_t

side of the main inequality is not increased at these events, and it is crucial for the proof that the same is true for the right side of the main inequality. Hence, these packets have to be contained in D_t. Intuitively, PG has already gained enough value to pay these packets in advance, i.e., before they are sent by OPT.

The basic idea for $c_t(p)$ is the following. In case of a send event σ_t in which OPT sends a much more valuable packets than PG that is not in D_{t-1}, the right side of the main inequality is increased by a large amount and we have to compensate this by increasing the charge of packets stored in the buffer of PG. It is fairly unproblematic to charge a packet up to $(r-1)$ times its own value, since, if such a packet is sent by PG and OPT in the same send event, the left side of the main inequality is still increased by the same amount as the right side of the main inequality. In any case, larger charges are only allowed for packets that are exclusively in the buffer of PG.

In case of a buffer overflow in the buffer of PG in which a charged packet is ejected, this charge has to be transferred to another packet in the buffer of PG. This is problematic for an ejected packet that is charged by more than $(r-1)$ times its own value, since, after this charge is transferred to another packet in the buffer of PG, there might be a packet charged by more than $(r-1)$ times its own value that is not exclusively in the buffer of PG. Therefore we introduce the concept of buddies. A packet stored exclusively in the buffer of PG might be charged by $2(r-1)$ times its own value, only if there is another packet in the buffer of PG that is not charged at all. We call the packet with no charge buddy for the packet with the high charge.

Unfortunately, the precise definition of charges is slightly more complicated. Before we define the charges in detail, we need some preliminaries. For each two packets p and p', we write $p \prec p'$, if p arrives before p' in the input sequence. Further, for each packet p and the undefined symbol $\perp$, $p \prec \perp$, $\perp \prec p$, and $\perp \prec \perp$. Each $p \in B_t^{\mathrm{pg}}$ can have assigned another $p' \in B_t^{\mathrm{pg}}$ as *buddy*, if $p \prec p'$. But each $p' \in B_t^{\mathrm{pg}}$ is assigned as buddy for at most one other packet. If $p \in B_t^{\mathrm{pg}}$ has assigned another $p' \in B_t^{\mathrm{pg}}$ as buddy at the end of event σ_t, define $b_t(p) := p'$, otherwise, define $b_t(p) := \perp$. Further, for each $p \notin B_t^{\mathrm{pg}}$, $b_t(p) := \perp$. Finally, for each $p \in B_t^{\mathrm{pg}}$, define $v_t^{\min}(p) := \min\{v(p') | B_t^{\mathrm{pg}} \ni p' \preceq p\}$.

Each $p \in B_t^{\mathrm{pg}}$ is in one of the five states BC, B, U, E, and EB. Let $s_t(p)$ denote the state of p at the end of event σ_t, and define $s_t(\perp) := \perp$. Let BC_t, B_t, U_t, E_t,

and EB_t denote the set of packets that are in state BC, B, U, E, and EB, respectively, at the end of event σ_t. The initial state of each packet is B, and dummy packets of value 0 are always in state B. The charge $c_t(p)$ of a packet p at the end of event σ_t is defined in Fig. 1. Note that the charge of a packet, except for packets in state U, does not change as long as this packet stays in the same state. The charge of a packet in state U can only increase, since $v_t^{\min}(p) \leq v_{t+1}^{\min}(p)$.

Let P_t denote the set of packets that are preempted by PG by the end of event σ_t. For each packet p, if p preempts another packet p', define $d(p) := p'$, otherwise, define $d(p) := \perp$. A packet p transitively preempts another packet p', if either $d(p) = p'$, p preempts a packet that transitively preempts p', or p ejects a packet that transitively preempts p'. For each $p' \in P_t$, if p' is transitively preempted by a packet $p \in B_t^{\mathrm{pg}}$, define $\hat{d}_t(p') := p$, otherwise, define $\hat{d}_t(p') := \perp$. For each $p' \notin P_t$, define $\hat{d}_t(p') := \perp$.

In order to prove the theorem, we show the following 5 invariants by induction over the event sequence σ.

I1: $\Delta_t^{\mathrm{pg}} \geq \Delta_t^{\mathrm{opt}}$.
I2: If $p \in \mathrm{E}_t \cup \mathrm{EB}_t$, then $p \notin B_t^{\mathrm{opt}}$.
I3: If $p \in \mathrm{EB}_t$, then $b_t(p) \in \mathrm{BC}_t \cup \mathrm{B}_t$.
I4: If $p \in X_t := (P_t \cup S_t^{\mathrm{pg}}) \cap (B_t^{\mathrm{opt}} \setminus D_t)$, then $\hat{d}_t(p) \in \mathrm{BC}_t \cup \mathrm{B}_t$.
I5: If $p \in B_t^{\mathrm{pg}} \setminus \mathrm{BC}_t$, then $b_t^{-1}(p) \prec d(p)$.

Observe that the invariants have only to be verified in the following cases.

I1: Always.
I2: For each packet $p \in (\mathrm{E}_t \cup \mathrm{EB}_t) \setminus (\mathrm{E}_{t-1} \cup \mathrm{EB}_{t-1})$.
I3: For each packet p with $p \in \mathrm{EB}_t \setminus \mathrm{EB}_{t-1}$, $b_{t-1}(p) \in (\mathrm{BC}_{t-1} \cup \mathrm{B}_{t-1}) \setminus (\mathrm{BC}_t \cup \mathrm{B}_t)$, or $b_{t-1}(p) \neq b_t(p)$.
I4: For each packet p with $p \in X_t \setminus X_{t-1}$, $\hat{d}_{t-1}(p) \in (\mathrm{BC}_{t-1} \cup \mathrm{B}_{t-1}) \setminus (\mathrm{BC}_t \cup \mathrm{B}_t)$, or $\hat{d}_{t-1}(p) \neq \hat{d}_t(p)$.
I5: For each packet p with $p \in (B_t^{\mathrm{pg}} \setminus \mathrm{BC}_t) \setminus (B_{t-1}^{\mathrm{pg}} \setminus \mathrm{BC}_{t-1})$, or $b_{t-1}^{-1}(p) \neq b_t^{-1}(p)$.

The following lemma restricts the set of packets stored in the buffer of OPT.

Lemma 5. *W.l.o.g.* $B_t^{\mathrm{opt}} \subseteq P_t \cup S_t^{\mathrm{pg}} \cup B_t^{\mathrm{pg}}$.

Proof. We show that w.l.o.g. each $p \in B_t^{\mathrm{opt}}$ is not ejected by PG by the event σ_t. By analogous arguments, w.l.o.g. each $p \in B_t^{\mathrm{opt}}$ is stored in the buffer of PG at its arrival, i.e., it is not rejected by PG. Combining both results yields the lemma.

Assume that a packet $p \in B_t^{\mathrm{opt}}$ is ejected by another packet p' at the event σ_t. Then, $p' \in B_t^{\mathrm{opt}}$, because otherwise OPT could increase its total value by dropping p instead of p'. Since $p, p' \in B_t^{\mathrm{opt}}$ and p is ejected by p' at the event σ_t, a $q \in B_t^{\mathrm{pg}} \setminus B_t^{\mathrm{opt}}$ exists with $v(q) \geq v(p)$. W.l.o.g. q is sent by OPT in a previous event $\sigma_{t'}$ with $t' < t$. Now assume that OPT sends p instead of q at the event $\sigma_{t'}$ and that $q \in B_t^{\mathrm{opt}}$ instead of $p \in B_t^{\mathrm{opt}}$. Note that p can arrive after the event $\sigma_{t'}$, however, OPT can just take advantage in this case.

Invariants I3, I4 and I5 are not effected, since changes are not made in the buffer of PG and $q \notin P_t \cup S_t^{\mathrm{pg}}$. If $s_t(q) \notin \{\mathrm{E}, \mathrm{EB}\}$, invariant I2 is not effected either. Otherwise, $s_t(q) := \mathrm{U}$ and, if q was in state EB, $s_t(b(q)) := \mathrm{U}$. Hence, invariants I2–I5 are still true.

Observe that Δ_t^{opt} is decreased by $v(q) - v(p)$. Hence, Δ_t^{pg} can be decreased by the same amount.

- If q was in state E and its state changed to U, Δ_t^{pg} is decreased by at most $(r-1) \cdot v(q) - ((r/\beta) \cdot v(q) + (2-r) \cdot v_t^{\min}(q)) = (2-r) \cdot (v(q) - v_t^{\min}(q)) \leq v(q) - v(p)$, since $r/\beta = 2r - 3$ and p is ejected at the event σ_t.
- If q was in state EB and its state changed to U, Δ_t^{pg} is decreased by at most $2(r-1) \cdot v(q) - ((r/\beta) \cdot v(q) + (2-r) \cdot v_t^{\min}(q)) = v(q) + (r-2) \cdot v_t^{\min}(q) \leq v(q) + (r-2) \cdot v(p)$. In this case, the state of $b(q)$ changed from BC or B to U. This increases Δ_t^{pg} by at least $(r/\beta) \cdot v(b(q)) + (2-r) \cdot v_t^{\min}(b(q)) \geq (r/\beta) \cdot v(p) + (2-r) \cdot v(p) \geq (r-1) \cdot v(p)$. Hence, in total Δ_t^{pg} is decreased by at most $v(q) + (r-2) \cdot v(p) - (r-1) \cdot v(p) = v(q) - v(p)$.

Altogether, invariant I1 is true. □

Fix an arrival event σ_t in which a packet p arrives. We distinguish the following cases. If not mentioned otherwise, everything remains unchanged at σ_t. The verification of the invariants I1–I5 is omitted due to space limitations. Observe that the only possible state transition at σ_t is that p is either in the state B or U at the end of σ_t.

- p preempts another packet q
  - $q \in \mathrm{B}_{t-1} \cup \mathrm{BC}_{t-1}$
    Changes: $s_t(p) := \mathrm{B}$ and $b_t(b_{t-1}^{-1}(q)) := p$
  - $q \in \mathrm{E}_{t-1} \cup \mathrm{EB}_{t-1}$
    Changes: $s_t(p) := \mathrm{U}$
  - $q \in \mathrm{U}_{t-1}$
    Changes: $s_t(p) := \mathrm{U}$ and $D_t := D_{t-1} \cup \{q\}$
- p ejects another packet q
  - $q \in \mathrm{B}_{t-1} \cup \mathrm{BC}_{t-1}$
    Changes: $s_t(p) := \mathrm{B}$ and $b_t(b_{t-1}^{-1}(q)) := p$
  - $q \in \mathrm{E}_{t-1} \cup \mathrm{U}_{t-1}$
    Changes: $s_t(p) := \mathrm{U}$
  - $q \in \mathrm{EB}_{t-1}$
    Changes: $s_t(p) := \mathrm{U}$ and $s_t(b_{t-1}(q)) := \mathrm{U}$
- p is not stored in the buffer of PG
  Changes: –
  (Due to lemma 5, p is also not stored in the buffer of OPT.)

Fix a send event σ_t in which PG sends packet p and OPT sends packet q. Note that due to lemma 5, $q \in P_{t-1} \cup S_{t-1}^{\mathrm{pg}} \cup B_{t-1}^{\mathrm{pg}}$. Since a new dummy packet of value 0 is stored in the buffer of PG after a packet is sent, a packet $u_{\mathrm{B}} \in B_t^{\mathrm{pg}} \setminus B_{t-1}^{\mathrm{pg}}$ exists with $s_t(u_{\mathrm{B}}) = \mathrm{B}$. We can assign u_{B} as buddy to another packet at this

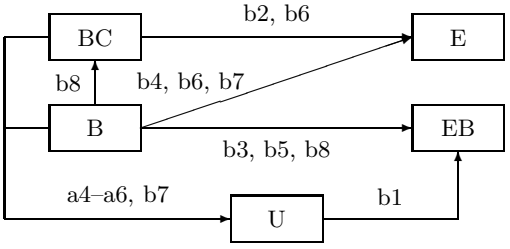


Fig. 2. Possible state transitions at a send event. The labels at the edges specify the cases in which the respective state transition could occur.

event, since $u_\mathrm{B} \notin B^{\mathrm{pg}}_{t-1}$. We distinguish the following cases. If not mentioned otherwise, everything remains unchanged at σ_t. The verification of the invariants I1–I5 is omitted due to space limitations. In Fig. 2, we depict the possible state transitions at σ_t.

- $q \in P_{t-1} \cup S^{\mathrm{pg}}_{t-1}$
  - a1: $q \in D_{t-1}$ and $p \in \mathrm{B}_{t-1} \cup \mathrm{BC}_{t-1}$
    Changes: $D_t := D_{t-1} \cup \{p\} \cup \{p' | \hat{d}_{t-1}(p') = p\}$
  - a2: $q \in D_{t-1}$ and $p \in B^{\mathrm{opt}}_t \setminus (\mathrm{B}_{t-1} \cup \mathrm{BC}_{t-1})$
    Changes: $D_t := D_{t-1} \cup \{p\}$
  - a3: $q \in D_{t-1}$ and $p \notin B^{\mathrm{opt}}_t \cup (\mathrm{B}_{t-1} \cup \mathrm{BC}_{t-1})$
    Changes: –
  - a4: $q \notin D_{t-1}$ and $p \in \mathrm{B}_{t-1} \cup \mathrm{BC}_{t-1}$
    Changes: $s_t(\hat{d}_{t-1}(q)) := \mathrm{U}$, $D_t := D_{t-1} \cup \{p\} \cup \{p' | \hat{d}_{t-1}(p') = p\} \cup \{q' \neq q | \hat{d}_{t-1}(q') = \hat{d}_{t-1}(q)\}$, $b_t(b^{-1}_{t-1}(\hat{d}_{t-1}(q))) := u_\mathrm{B}$
    (Due to I4, $\hat{d}_{t-1}(q) \in \mathrm{BC}_{t-1} \cup \mathrm{B}_{t-1}$.)
  - a5: $q \notin D_{t-1}$ and $p \in B^{\mathrm{opt}}_t \setminus (\mathrm{B}_{t-1} \cup \mathrm{BC}_{t-1})$
    Changes: $s_t(\hat{d}_{t-1}(q)) := \mathrm{U}$, $D_t := D_{t-1} \cup \{p\} \cup \{q' \neq q | \hat{d}_{t-1}(q') = \hat{d}_{t-1}(q)\}$, $b_t(b^{-1}_{t-1}(\hat{d}_{t-1}(q))) := u_\mathrm{B}$
  - a6: $q \notin D_{t-1}$ and $p \notin B^{\mathrm{opt}}_t \cup (\mathrm{B}_{t-1} \cup \mathrm{BC}_{t-1})$
    Changes: $s_t(\hat{d}_{t-1}(q)) := \mathrm{U}$, $D_t := D_{t-1} \cup \{q' \neq q | \hat{d}_{t-1}(q') = \hat{d}_{t-1}(q)\}$, $b_t(b^{-1}_{t-1}(\hat{d}_{t-1}(q))) := u_\mathrm{B}$
- $q = p$
  Changes: –
- $q \in B^{\mathrm{pg}}_{t-1} \setminus \{p\}$
  - b1: $q \in \mathrm{U}_{t-1}$
    Changes: $b_t(q) := u_\mathrm{B}$, $s_t(q) := \mathrm{EB}$
  - b2: $q \in \mathrm{BC}_{t-1}$
    Changes: $b_t(b^{-1}_{t-1}(q)) := u_\mathrm{B}$, $s_t(q) := \mathrm{E}$
  - b3: $q \in \mathrm{B}_{t-1}$ and $v(p) < v(q)/\beta$
    Changes: $b_t(q) := u_\mathrm{B}$, $s_t(q) := \mathrm{EB}$
    (Due to I5, $b^{-1}_{t-1}(q) \prec d(q)$, i.e., $b^{-1}_{t-1}(q) \notin B^{\mathrm{pg}}_{t-1}$, since $v(p) < v(q)/\beta$.)
  - b4: $q \in \mathrm{B}_{t-1}$ and $v(p) \geq v(q)/\beta$ and $p \notin \mathrm{EB}_{t-1}$
    Changes: $b_t(b^{-1}_{t-1}(q)) := u_\mathrm{B}$, $s_t(q) := \mathrm{E}$

b5: $q \in \mathrm{B}_{t-1}$ and $v(p) \geq v(q)/\beta$ and $p \in \mathrm{EB}_{t-1}$ and $b_{t-1}^{-1}(q) = \perp$
Changes: $b_t(q) := u_\mathrm{B}$, $s_t(q) := \mathrm{EB}$

b6: $q \in \mathrm{B}_{t-1}$ and $v(p) \geq v(q)/\beta$ and $p \in \mathrm{EB}_{t-1}$ and $b_{t-1}(p) \preceq b_{t-1}^{-1}(q) \neq \perp$
Changes: $s_t(b_{t-1}(p)) := \mathrm{E}$, $b_t(b_{t-1}^{-1}(q)) := u_\mathrm{B}$, $s_t(q) := \mathrm{E}$
(Due to I5, $b_{t-1}^{-1}(q) \prec d(q)$, i.e., $v(b_{t-1}(p)) \geq v(q)/\beta$.)

b7: $q \in \mathrm{B}_{t-1}$ and $v(p) \geq v(q)/\beta$ and $p \in \mathrm{EB}_{t-1}$ and $\perp \neq b_{t-1}^{-1}(q) \prec b_{t-1}(p)$ and $v(b_{t-1}(p)) > 2v(q)$
Changes: $s_t(b_{t-1}(p)) := \mathrm{U}$, $D_t := D_{t-1} \cup \{p' | \hat{d}_{t-1}(p') = b_{t-1}(p)\}$, $b_t(b_{t-1}^{-1}(q)) := u_\mathrm{B}$, $s_t(q) := \mathrm{E}$

b8: $q \in \mathrm{B}_{t-1}$ and $v(p) \geq v(q)/\beta$ and $p \in \mathrm{EB}_{t-1}$ and $\perp \neq b_{t-1}^{-1}(q) \prec b_{t-1}(p)$ and $v(b_{t-1}(p)) \leq 2v(q)$
Changes: $s_t(b_{t-1}(p)) := \mathrm{BC}$, $b_t(b_{t-1}^{-1}(q)) := b_{t-1}(p)$, $b_t(q) := u_\mathrm{B}$, $s_t(q) := \mathrm{EB}$

This concludes the proof of the theorem. □

References

1. W. Aiello, Y. Mansour, S. Rajagopolan, and A. Rosen. Competitive queue policies for differentiated services. *Journal of Algorithms*, 55(2):113–141, 2005.
2. N. Andelman. Randomized queue management for DiffServ. In *Proceedings of the 17th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 1–10, 2005.
3. N. Andelman, Y. Mansour, and A. Zhu. Competitive queueing policies for QoS switches. In *Proceedings of the 14th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 761–770, 2003.
4. Y. Azar and Y. Richter. Management of multi-queue switches in QoS networks. In *Proceedings of the 35th ACM Symposium on Theory of Computing (STOC)*, pages 82–89, 2003.
5. N. Bansal, L. Fleischer, T. Kimbrel, M. Mahdian, B. Schieber, and M. Sviridenko. Further improvements in competitive guarantees for QoS buffering. In *Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP)*, pages 196–207, 2004.
6. W. Jawor. Three dozen papers on online algorithms. *SIGACT News*, 36(1):71–85, 2005.
7. A. Kesselman, Z. Lotker, Y. Mansour, B. Patt-Shamir, B. Schieber, and M. Sviridenko. Buffer overflow management in QoS switches. *SIAM Journal on Computing*, 33(3):563–583, 2004.
8. A. Kesselman and Y. Mansour. Loss-bounded analysis for differentiated services. *Journal of Algorithms*, 46(1):79–95, 2003.
9. A. Kesselman, Y. Mansour, and R. van Stee. Improved competitive guarantees for QoS buffering. *Algorithmica*, 43(1–2):63–80, 2005.
10. Z. Lotker and B. Patt-Shamir. Nearly optimal FIFO buffer management for DiffServ. In *Proceedings of the 21st Symposium on Principles of Distributed Computing (PODC)*, pages 134–143, 2002.
11. Y. Mansour, B. Patt-Shamir, and O. Lapid. Optimal smoothing schedules for real-time streams. In *Proceedings of the 19th Symposium on Principles of Distributed Computing (PODC)*, pages 21–29, 2000.

Graph Coloring with Rejection

Leah Epstein[1], Asaf Levin[2], and Gerhard J. Woeginger[3]

[1] Department of Mathematics, University of Haifa, 31905 Haifa, Israel
lea@math.haifa.ac.il
[2] Department of Statistics, The Hebrew University, 91905 Jerusalem, Israel
levinas@mscc.huji.ac.il
[3] Department of Mathematics and Computer Science, TU Eindhoven P.O. Box 513, 5600 MB Eindhoven, The Netherlands
gwoegi@win.tue.nl

Abstract. We consider the following vertex coloring problem. We are given an undirected graph $G = (V, E)$, where each vertex v is associated with a penalty rejection cost r_v. We need to choose a subset of vertices, V', and to find a proper coloring of the induced subgraph of G over V'. We are interested in both the number of colors used to color the vertices of V', and in the total rejection cost of all other vertices. The goal is to minimize the sum of these two amounts. In this paper we consider both the online and the offline versions of this problem. In the online version, vertices arrive one at a time, revealing the rejection cost of the current vertex and the set of edges connecting it to previously revealed vertices. We also consider the classical online coloring problem on bounded degree graphs and on $(k+1)$-claw free graphs.

1 Introduction

Given an undirected graph $G = (V, E)$, a coloring of G is an assignment of colors to the vertices such that two adjacent vertices are assigned distinct colors. I.e., a coloring is a function $c : V \to Z^+$ such that if $(i, j) \in E$ then $c(i) \neq c(j)$. In the offline COLORING PROBLEM, the goal is to find a coloring of G where the number of used colors, $\max_{i \in V} c(i)$, is minimized. This problem is well-known to be NP-hard for general input graphs (see problem [GT4] in [3]), however it can be solved in polynomial time when the input graph is *perfect* (using the ellipsoid method, see [9]). The minimum number of colors that are necessary in order to color G, is called the *chromatic number of* G and it is denoted by $\chi(G)$. For a coloring c, denote by $n(c, G)$ the number of distinct colors that are used by c.

We define the COLORING WITH REJECTIONS problem, which is a new generalization of the standard coloring problem, as follows. We are given an undirected graph $G = (V, E)$, where each vertex $v \in V$ is associated with a non-negative penalty rejection cost r_v. The goal is to find a subset $V' \subseteq V$, and a coloring c of $G[V']$ that is the induced subgraph of G over V', so as to minimize the sum of the number of used colors (in the coloring of $G[V']$) and the total rejection cost of all the vertices in $V \setminus V'$. I.e., the goal function is $n(c, G[V']) + \sum_{v \in V \setminus V'} r_v$. If we can color every induced subgraph of G using a minimum number of colors, then our goal is to find a vertex set V' that minimizes $\chi(G[V']) + \sum_{v \in V \setminus V'} r_v$.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 364–375, 2006.

The use of a penalty function for failing to serve some clients is a common practice in combinatorial optimization problems. For example, this is the motivation behind the definition of the prize-collecting Steiner tree and the prize-collecting traveling salesperson problems (in both problems we pay a penalty for not connecting a vertex, where in the first problem the goal is to construct a tree that spans some vertices, and in the second problem the goal is to construct a cycle over some vertex set). For an earlier work on these problems see for example [4].

Since there does not exist a competitive online algorithm for the standard coloring problem even for very limited graph classes (such as trees), and as the coloring problem with rejections generalizes the standard coloring problem, we consider both problems on special classes of graphs, where one can hope to achieve a finite competitive ratio.

In an online setting, vertices arrive one by one, and we need to deal with an arriving vertex before seeing any future vertices. In the ONLINE COLORING WITH REJECTIONS problem, when a vertex arrives the following information is revealed. Its rejection cost, and the edges that connect this new vertex to all the previously introduced vertices (but no information regarding edges to the future vertices is given at this time). We need to decide whether we would like to accept the new vertex (i.e., to add it to the vertex set V') or to reject it. If the algorithm decides to accept the new vertex, then it needs to color it using one of the existing colors or using a new color (and in this case we say that the algorithm *opens* a new color). In the ONLINE COLORING problem we simply do not reveal the rejection cost of a vertex (as such a notion is undefined for the standard coloring problem, or we can alternatively assume infinite rejection costs for all vertices in this case).

The MAXIMUM WEIGHT k-COLORABLE SUBGRAPH problem is the following related problem. The input to this problem consists of an integer number k and an undirected graph $G = (V, E)$, where each vertex v has a non-negative weight w_v. The goal is to pick a subset $V' \subseteq V$, such that there exists a coloring c of $G[V']$ with k colors, and among all such subsets, the value $\sum_{v \in V'} w_v$ is maximized. Yannakakis and Gavril [11] showed that this problem is NP-hard on split graphs when k is part of the input, and it is polynomially solvable on interval graphs. This was done by formulating the problem as an integer Linear Program on the vertices versus cliques constraint matrix, which is totally unimodular. If all weights are equal, we obtain the MAXIMUM SIZE k-COLORABLE SUBGRAPH problem, that was shown to be polynomial for comparability and co-comparability graphs by Frank [2].

The online coloring problem is well-studied (see Kierstead [7] for a survey). Gyárfás and Lehel [5] showed that for every positive integer k and every online algorithm $\mathcal{A}$, there exists a tree T_k on 2^{k-1} vertices, with maximum degree $k-1$, such that $\mathcal{A}$ must use at least k colors when coloring T_k. They also showed that for trees, the First-Fit algorithm denoted as FF (the FF algorithm assigns for each vertex the minimum color that can be used to color the vertex) is best possible among all possible online algorithms (in terms of the competitive

ratio). Bar-Noy, Motwani and Naor [1] showed that First-Fit is a 2-competitive online algorithm for coloring line graphs, and that any online algorithm has a competitive ratio of at least two (and hence First-Fit is the best possible online algorithm for coloring line graphs).

For an algorithm $\mathcal{A}$, we denote its cost by $\mathcal{A}$ as well. The cost of an optimal offline algorithm that knows the complete sequence of vertices is denoted by OPT. In this paper we consider the absolute competitive ratio and the absolute approximation ratio criteria. For an online algorithm we use the term competitive ratio whereas for an offline algorithm we use the term approximation ratio. The competitive ratio of $\mathcal{A}$ is the infimum $\mathcal{R}$ such that for any input, $\mathcal{A} \leq \mathcal{R} \cdot \mathrm{OPT}$. If the competitive ratio of an online algorithm is at most $\mathcal{C}$ we say that it is $\mathcal{C}$ competitive. The approximation ratio of a polynomial time offline algorithm is defined similarly to be the infimum $\mathcal{R}$ such that for any input, $\mathcal{A} \leq \mathcal{R} \cdot \mathrm{OPT}$. If the approximation ratio of a polynomial time offline algorithm is at most $\mathcal{R}$ we say that it is a $\mathcal{R}$ approximation.

A *perfect graph* is a graph G such that for every induced subgraph G' of G, the chromatic number of G' equals its maximum clique size. A *split graph* is a graph whose vertices can be partitioned into two subsets I and K such that I is an independent set and K induces a complete graph. Note that the complement of a split graph is a split graph. A $(k+1)$-*claw free graph* is a graph that does not contain an induced subgraph that is a star with $k+1$ leaves, and a *claw-free graph* is a 3-claw free graph. A *line graph* G can be modeled by the edges of a second graph $H = (V_H, E_H)$ such that the vertex set of G is E_H and there is an edge between two vertices of G if the corresponding edges of H share a common endpoint.

Our results. In Section 2 we consider the offline coloring problem with rejections. Given a graph class $\mathcal{F}$ we show a close relation between the complexity of this problem on $\mathcal{F}$ to the complexity of the maximum weight k-colorable subgraph problem on $\mathcal{F}$. In fact we show that if the class $\mathcal{F}$ is closed under the operation of union with a disjoint finite clique, then one problem is polynomially solvable if and only if the other one is polynomially solvable. Among the sub-classes of the family of perfect graphs discussed in [6], the only graph class that is not closed under this operation is the class of split graphs. For this family it is known that the maximum weight k-colorable subgraph problem is NP-hard, and we show that the coloring problem with rejections on split graphs is NP-hard as well. Then, we turn our attention to approximation algorithms for this problem. We first consider split graphs, we show an approximation algorithm with an additive error guarantee of one from the optimum, and then we show how to derive a polynomial time approximation scheme for the coloring problem with rejections on split graphs. We conclude this section by presenting an $O(\log n)$ approximation algorithm for the coloring problem with rejections on perfect graphs that is based on the greedy algorithm. In Section 3 we focus on the classical online coloring problem, and analyze the competitive ratio of the First-Fit algorithm on two graph classes. We show that First-Fit is a $\frac{\Delta+1}{2}$ competitive algorithm for the class of graphs with a maximum vertex degree of

at most Δ, and we show that the First-Fit algorithm colors a $(k+1)$-claw free graph using at most $k\mathrm{OPT}-k+1$ colors. We also show that any online algorithm cannot perform better on these graph classes. In Section 4 we turn to deal with the online coloring problem with rejections. We first show an online algorithm whose competitive ratio is $\Delta+2$, for the class of graphs with maximum degree at most Δ. Thus we show that for bounded degree graphs, adding the notion of rejections to the problem makes it harder but still tractable. We then show that there is no online algorithm whose competitive ratio is smaller (in terms of Δ), even if the input graph is a collection of disjoint cliques. Since the class of $(k+1)$-claw free graphs contains all graphs which are collections of cliques, we get that unlike the bounded degree problem, adding rejections to this problem makes it much harder . Among the sub-classes of the family of perfect graphs discussed in [6], the only graph class that does not contain a disjoint union of cliques is the class of split graphs. Therefore, we conclude the paper by showing that even for split graphs there is no online algorithm with a finite competitive ratio.

2 Offline Coloring with Rejections

In this section we study the offline version of the coloring with rejection. We show a close connection between the tractability of this problem on graph classes $\mathcal{F}$ and the tractability of the problem of computing the maximum weight k-colorable subgraph of graphs that belong to $\mathcal{F}$.

Theorem 1. *Given a graph class $\mathcal{F}$ such that $\mathcal{F}$ is closed under the operation of union with a disjoint finite clique, the offline coloring problem with rejections on $\mathcal{F}$ is polynomially solvable if and only if the maximum weight k-colorable subgraph problem on graphs that belong to $\mathcal{F}$ is polynomially solvable.*

Proof. Assume that for $\mathcal{F}$ it is possible to solve the maximum weight k-colorable subgraph problem. Then, given an instance of the offline coloring with rejections $G=(V,E)$ where $G\in\mathcal{F}$ and rejection cost $r_v,\ v\in V$, we apply the following procedure. For each value of k we compute the maximum weight k-colorable subgraph of G where the weight w_v of a vertex v equals its rejection cost, and denote this subgraph by (V_k,E_k). For each value of k, we compute the total cost of k and the total rejection cost of the vertices from $V\setminus V_k$, and we pick the solution whose total cost is minimized. In order to analyze the performance of the resulting algorithm we fix an optimal solution OPT, and consider the iteration in which the algorithm uses the value of k that equals the correct number of colors that OPT uses. By the optimality of (V_k,E_k) to the maximum weight k-colorable subgraph problem, we conclude that the total rejection cost of the vertices in $V\setminus V_k$ is at most the total rejection cost that OPT pays. Therefore, the algorithm returns an optimal solution to the coloring with rejections problem.

Next assume that for the class $\mathcal{F}$, it is possible to compute in polynomial time an optimal solution to the coloring problem with rejection. Let $G=(V,E)$ and $w:V\rightarrow R^+$ be an input instance to the maximum weight k-colorable subgraph

problem such that $G \in \mathcal{F}$. Let $G' = (V', E')$ be the graph resulting from G by augmenting it with a set of k new vertices and a clique over them (so G' is a union of G and a clique of size k). Since $G \in \mathcal{F}$ and by the assumption on $\mathcal{F}$, we conclude that $G' \in \mathcal{F}$. Therefore, it is possible to compute in polynomial time the optimal solution to the coloring with rejections problem for every rejection cost function, r. Let $0 \leq \varepsilon \leq \frac{1}{1+\sum_{v \in V} w(v)}$ and define a rejection cost function r as follows: $r(v) = 1$ if $v \in V' \setminus V$ (i.e., if it belongs to the new clique), and otherwise $r(v) = \varepsilon \cdot w(v)$.

Consider an optimal solution OPT to the instance of the coloring problem with rejections. W.l.o.g. OPT does not reject a vertex $v \in V' \setminus V$ (this can be assumed as opening a new color and assigning v to the new color does not increase the total cost). Therefore, the number of colors that OPT uses is at least k. We next argue that OPT uses exactly k colors. To see this claim, assume otherwise that OPT uses at least $k+1$ colors. Then, there is a color in OPT such that each vertex that is assigned to this color is from V, and therefore by definition of ε, the total rejection cost of all the vertices that are assigned to this color is less than 1. Therefore, rejecting all these vertices rather than opening this color results in a solution of a strictly smaller cost, which contradicts the optimality of OPT.

Given that OPT uses exactly k colors, it assigns each one of the clique vertices, $V' \setminus V$, to one of these colors, and also assigns a maximum weight k-colorable subgraph of G to these colors (and the remaining vertices if there are such, are rejected). Since the rejection costs are simply the scaled original weights, OPT solves also the maximum weight k-colorable subgraph of G in polynomial time. □

In the full version of the paper we show that the offline coloring with rejection problem is NP-hard even when the input graph is restricted to be a split graph. Although computing the maximum weight k-colorable subgraph of a split graph is known to be NP-hard (when k is not fixed), the next result does not follow from Theorem 1 as split graphs are not closed under the operation of union with a disjoint clique. The following proposition can be proved by using a reduction from 3-set packing problem.

Proposition 1. *The offline coloring problem with rejections is NP-hard in the strong sense even when the input graph is restricted to be a split graph.*

We next show that for a split graph there is an approximation algorithm with an additive approximation guarantee. We do not require an input split graph to be introduced together with its realization. However, such a realization is easy to find in polynomial time.

Theorem 2. *If the input to the offline coloring with rejections problem is a split graph, and the optimal solution* OPT *has cost* OPT*, then it is possible to compute in polynomial time a feasible solution* SOL *whose cost is at most* $\mathrm{OPT} + 1$.

Proof. Assume that G is decomposed into an independent set $\mathcal{I}$ and a vertex set $\mathcal{K}$ such that the induced subgraph of G over $\mathcal{K}$ is a complete graph. Clearly such a decomposition can be found in polynomial time since G is a split graph.

The algorithm guesses the number of colors that OPT uses. Denote this number by $\mathcal{N}$. The term "guessing" means performing an exhaustive enumeration of all possibilities from the polynomial size set $\{0, 1, 2, \ldots, |V|\}$, and picking the cheapest solution among the resulting solutions (one solution for each possibility). In the analysis it suffices to consider the solution that the algorithm returns in the iteration in which it uses the correct value of $\mathcal{N}$.

SOL uses $\mathcal{N}+1$ colors. The first color is used to color the independent set $\mathcal{I}$ of G. The other $\mathcal{N}$ colors are used to color the $\mathcal{N}$ most expensive rejection cost vertices from $\mathcal{K}$ (one vertex per each of these $\mathcal{N}$ colors).

The total cost that SOL pays is the sum of $\mathcal{N}+1$ and the total rejection cost. It suffices to show that the total rejection cost that OPT pays is at least the total rejection cost that SOL pays. However, this is clear as each color in OPT can be used to color at most one vertex from $\mathcal{K}$ and therefore OPT pays the rejection cost of at least $|\mathcal{K}| - \mathcal{N}$ vertices from $\mathcal{K}$ as SOL does. We conclude the proof using the fact that SOL pays the rejection cost for the cheapest vertices in $\mathcal{K}$. □

We can actually show how to transform the algorithm of Theorem 2 into a polynomial time approximation scheme.

Corollary 1. *There is a PTAS for approximating the offline coloring problem with rejections on split graphs.*

We conclude this section by considering approximation algorithms for perfect graphs. For the maximum weight k-colorable subgraph problem on perfect graphs there is an easy $(1 - \frac{1}{e})$-approximation algorithm that is based on the greedy algorithm for the maximum coverage problem. The MAXIMUM COVERAGE problem is defined as follows: We are given a ground set $\mathcal{E}$ where each element $e \in \mathcal{E}$ has a weight w_e, a collection $\mathcal{S}$ of subsets of $\mathcal{E}$ and an integer number k. The goal is to find a sub-collection $\mathcal{S}'$ of $\mathcal{S}$ of (exactly) k subsets that covers a maximum total weight of the elements of $\mathcal{E}$ (where covering an element means that at least one of the subsets in $\mathcal{S}'$ contains the element). It is known (see [8]) that the *greedy algorithm* is an $(1 - \frac{1}{e})$-approximation algorithm where the greedy algorithm at each iteration adds the set from $\mathcal{S}$ that covers the most total weight of elements that were not covered prior to this iteration, to the collection $\mathcal{S}'$ until there are k subsets in $\mathcal{S}'$. Noting that we can formulate the maximum weight k-colorable subgraph problem as an instance of the maximum coverage problem, by letting $\mathcal{E}$ be the set of vertices of the input graph, and $\mathcal{S}$ be the collection of independent sets of the graph. Note that the greedy algorithm does not use a list of the sets in $\mathcal{S}$, but it only needs to compute a maximum weight independent set in a residual graph (the graph induced by the non-covered vertices), and this can be done in polynomial time for perfect graphs. So we can apply the greedy algorithm for the maximum coverage problem where each iteration is implemented in

polynomial time (where the time complexity is polynomial in the size of the input graph).

However, we are not aware of a constant approximation algorithm for the offline coloring with rejections problem when applied to perfect graphs. We next discuss an $O(\log n)$-approximation algorithm for this problem.

Theorem 3. *There is an $O(\log n)$-approximation algorithm for the offline coloring problem with rejections on perfect graphs.*

Proof. We denote the total rejection cost of vertices for which OPT does not pay the rejection cost (this is the sum of rejection costs of the vertices that OPT colors) by R and by $\overline{R}$ the total rejection cost that OPT pays. We also denote by C the number of colors that OPT uses. The PARTIAL COVER problem is defined as follows. We are given a ground set $\mathcal{E}$ where each element $e \in \mathcal{E}$ has a weight w_e, a collection $\mathcal{S}$ of subsets of $\mathcal{E}$ and a number W. The goal is to find a subcollection $\mathcal{S}'$ of $\mathcal{S}$ of minimum number of subsets that covers a subset of $\mathcal{E}$ whose total weight is at least W (where covering an element means that at least one of the subsets in $\mathcal{S}'$ contains the element). It is known (see [10]) that the *greedy algorithm* is an $O(\log n)$-approximation algorithm for the partial cover problem where the greedy algorithm at each iteration adds the set from $\mathcal{S}$ that covers the most total weight of elements that were not covered prior to this iteration, to the collection $\mathcal{S}'$ until the total weight of covered elements is at least W. Note that we can formulate the offline coloring with rejections problem as an instance of the partial cover problem by letting $\mathcal{E}$ be the set of vertices of the input graph, $\mathcal{S}$ be the collection of independent sets of the graph and $W = R$. Given this instance note that there is a solution to the partial cover instance that uses at most C subsets and therefore the greedy algorithm returns a solution with at most $O(C \log n)$ subsets, and the total cost of the solution of the offline coloring with rejections is at most $O(C \log n) + \overline{R} \leq O(\log n) \cdot (C + \overline{R}) = O(\log n) \cdot \mathrm{OPT}$. So if the exact value of R is known to us, then the greedy algorithm for the partial cover instance is an $O(\log n)$ approximation algorithm. To overcome the lack of this information, we consider the following algorithm.

Apply the greedy algorithm for the partial cover instance until all elements are covered, adding each of the intermediate solutions created at the end of each iteration of the greedy algorithm, to a solution set denoted by SOL. When the algorithm terminates, return the best solution from SOL. We note that SOL contains the approximated greedy solutions for the partial cover instances for all possible values of W. Therefore, picking the cheapest solution among SOL is superior to the solution created for the correct value of R, and the last solution is an $O(\log n)$-approximation algorithm. □

3 Online Coloring (Without Rejections)

In this section, we consider the classical online coloring problem. We consider two classes of graphs, $(k+1)$-claw free graphs and bounded degree graphs. We show that the First-Fit algorithm is a best possible online algorithm for both cases.

3.1 $(k+1)$-Claw Free Graphs

In this subsection we prove that for $(k+1)$-claw free graphs, the First-Fit algorithm is a best possible online algorithm. Our result extends the result of Bar-Noy, Motwani and Naor [1] for line graphs (that are 3-claw free graphs).

Theorem 4. *The solution returned by the First-Fit algorithm uses at most* $k\mathrm{OPT} - k + 1$ *colors when applied to* $(k+1)$*-claw free graphs. Moreover, no online algorithm can guarantee a smaller number of colors.*

Proof (sketch). We first show that the solution returned by First-Fit uses at most $k\mathrm{OPT} - k + 1$ colors. Assume that First-Fit uses $t+1$ colors, and let v be a vertex that is colored by First-Fit using the $t+1$-th color. Then, clearly v has at least t neighbors. Since the input graph is a $(k+1)$-claw free graph, the maximum size independent set in the induced subgraph over the neighbors of v, has size at most k. Therefore, OPT must use at least $\lceil \frac{t}{k} \rceil$ colors to color the neighbors of v and one additional color to color v. Therefore, if $FF = t+1$, then $\lceil \frac{t}{k} \rceil + 1 \leq \mathrm{OPT}$. Therefore,

$$FF = t+1 \leq k \cdot \left(\left\lceil \frac{t}{k} \right\rceil + 1 \right) - k + 1 \leq k \cdot \mathrm{OPT} - k + 1.$$

It remains to show the lower bound. We fix a value of $k = \ell$, and the cost of the optimal solution, which we denote by $\mathrm{OPT} = n+1$. Our lower bound makes use of the following recursive construction. We define structures of types $1, 2, \ldots$, where each such structure is a graph which has a subset of n designated vertices which are called the "top clique". All other vertices are also arranged in cliques and are associated with levels. A structure of type 1 is simply defined to be a clique over n vertices, all these vertices are defined to be the top clique. These vertices are also called the level 1 clique. We define a structure of type t recursively using structures of types $\ell = 1, 2, \ldots, t$. To obtain a structure of type t, we construct for every $1 \leq i \leq t-1$, n copies of a type i structure. All $n(t-1)$ structures are pairwise disjoint and constructed independently from each other. Denote the structures of type i by $S_{i,1}, \ldots, S_{i,n}$. To combine them all into a type t structure, we construct an additional clique on n vertices, denoted by $v_1, \ldots, v_n$. This will be the top clique of the structure. Vertex v_j is connected to all vertices of the top clique in $S_{i,j}$ for all $1 \leq i \leq t-1$. The level i cliques of the type t structure, for $1 \leq i \leq t-1$ are all level i cliques of all structures of types $1, \ldots, t-1$ (or actually of types $i, \ldots, t-1$, since structures of smaller types do not have level i cliques), which are used in the type t structure. The level t clique of a type t structure is defined to be its top clique.

Next, we define a "superior structure" of type t. This structure is constructed similarly to a regular type t structure, but instead of a top clique which consists of n vertices, we use a top clique with $n+1$ vertices. Furthermore, we use $n+1$ structures of each type $1, \ldots, t-1$ instead of n such structures. Note that structures used in a recursive construction are regular structures and a superior structure is never used as a building block for another structure.

Our lower bound proof is based on considering optimal coloring of a superior type t structure, and then showing how to force an online algorithm to use a large number of colors. □

3.2 Bounded Degree Graphs

In this section we restrict ourselves to input graphs with maximum degree at most Δ. We can prove that the First-Fit algorithm (FF) is a best possible online algorithm for this case.

Theorem 5. *The First-Fit algorithm is a $\frac{\Delta+1}{2}$-competitive algorithm for online coloring of graphs with maximum degree at most Δ, and no online algorithm can guarantee a better competitive ratio on this graph class.*

Remark 1. Another way to achieve the same lower bound (using a similar construction) for coloring bounded degree graphs using the First-Fit algorithm, is by adapting the lower bound construction in [5]. In that paper, a lower bound of $\Omega(\log n)$ on the competitive ratio of online coloring of trees is given. One can stop their construction when a new vertex has a degree of exactly Δ. At this step each vertex has a degree of at most Δ. Moreover, OPT uses only two colors whereas the online algorithm is forced to use at least $\Delta + 1$ colors. Therefore, no online algorithm can guarantee a better than $\frac{\Delta+1}{2}$ competitive ratio.

4 Online Coloring with Rejections

In this section we first show our positive result, that is, restricting ourselves to input graphs with maximum degree of at most Δ, then there is an online algorithm with competitive ratio of $\Delta + 2$. Our first lower bound proves that without this restriction there is no competitive online algorithm. This lower bound applies to a disjoint union of cliques (with arbitrary size, so these are not bounded degree graphs). The second lower bound applies to split graphs. Our first lower bound shows that for each fixed value of Δ, our $\Delta + 2$ competitive algorithm is best possible (for any online algorithm). Another consequence from our constructions is that for $(k+1)$-claw free graphs, there is no constant competitive algorithm for the coloring problem with rejections. We note this to show that despite the similar behavior of the two classes of graphs, $(k + 1)$-claw free graphs and bounded degree graphs, with respect to the standard online coloring problem, the online coloring problem with rejections separates these two classes.

Theorem 6. *There is a $\Delta + 2$ competitive algorithm for the online coloring problem with rejections on graphs with a maximum degree of at most Δ.*

Proof. Our algorithm rejects all arriving vertices as long as the total rejection costs does not exceed one. Then, it opens $\Delta + 1$ colors and uses the First-Fit algorithm to color all future vertices (starting from the vertex in which the total rejection costs exceeds one, and that vertex is the first one to be accepted). If

the algorithm reaches the point where it opens $\Delta + 1$ colors, then its total cost is at most $\Delta + 2$ (as the total rejection cost is at most one). In this case the cost of an optimal solution must be at least one, since if OPT uses at least one color, then its cost is at least one, and otherwise it rejects all vertices with total rejection cost at least one (as our algorithm decided to open the colors).

Otherwise, assume that the algorithm rejects all vertices with a total rejection cost of x, then $x \leq 1$, and an optimal solution also rejects all the vertices with total cost of x (if $x = 1$ then there may be a different optimal solution which uses a single color and has the same cost). In this case the competitive ratio is $1 < \Delta + 2$. □

We next show that our algorithm is best possible for graphs with maximum degree at most Δ.

Theorem 7. *For all $\delta > 0$, there is no online algorithm for the online coloring problem with rejections on graphs with a maximum degree of at most Δ, whose competitive ratio is at most $\Delta + 2 - \delta$.*

Proof. Assume that there is an online algorithm whose competitive ratio is at most $\Delta+2-\delta$. Let $\varepsilon > 0$ to be defined later. We define a sequence $\{\varepsilon_k\}_{k=1}^{\Delta+1}$ as the (unique) solution for the following equations $\varepsilon_1 = \varepsilon$ and for all $k = 1, 2, \ldots, \Delta$, the following equation $\varepsilon \cdot \varepsilon_k = \sum_{j=k+1}^{\Delta+1} \varepsilon_j$. The implicit values of the solution are $\varepsilon_i = \frac{\varepsilon^i}{(1+\varepsilon)^{i-1}}$, for $i \leq \Delta$, and $\varepsilon_{\Delta+1} = \frac{\varepsilon^{\Delta+1}}{(1+\varepsilon)^{\Delta-1}}$. Our construction will have two phases. In both phases we will present vertices of disjoint cliques one clique after the other. In the first phase vertex i of the current clique has rejection cost $\varepsilon_{\Delta+2-i}$, and in the second phase vertex i of the current clique has rejection cost $\frac{\varepsilon_{\Delta+2-i}}{\varepsilon}$. We stop presenting vertices in the current clique (and move to the next clique) after the first time that the online algorithm has rejected a vertex from the clique. If the algorithm has opened (at least) $\Delta + 1$ colors we stop the instance immediately. This means that no clique has more than $\Delta + 1$ vertices, since introducing more than $\Delta + 1$ vertices in the clique means that the online algorithm did not reject any of the vertices. However, in that case, it must use $\Delta + 1$ colors to color the clique, and thus the construction terminates at this time. Given an upper bound of $\Delta + 1$ on the size of cliques, we get that the largest degree ever used is at most Δ.

The first phase lasts as long as the total rejection cost of the instance is at most one. Thus the last clique in this phase is presented until the total rejection cost is about to become at least one, and not until the algorithm rejects some vertex. Afterwards, in the next clique that we present, we move to the second phase. If the total rejection cost that the online algorithm pays ever exceeds $3(\Delta + 2)$ at some time, we stop the instance immediately, even if the online algorithm did not open at least $\Delta + 1$ colors. Note that at least one of the two events must happen. In each clique of the second phase, where less than $\Delta + 1$ colors are used by the algorithm, the algorithm increases its rejection cost by at least $\frac{\varepsilon_{\Delta+1}}{\varepsilon}$. Thus within a finite number of cliques either the rejection cost becomes large enough, or at least $\Delta + 1$ colors are used.

If we stop the instance already in the first phase, then the optimal solution rejects all vertices. Otherwise, we get to the second phase, and the solution that we consider as an upper bound on the optimal cost has one color, and it accepts only the last vertex from each clique. Therefore, if we denote the total rejection cost of the optimal solution by R_{opt} and the total rejection cost of the online algorithm by R_{onl}, then we argue that $R_{opt} \leq \varepsilon^2 + \varepsilon R_{onl} \leq \varepsilon + \varepsilon R_{onl}$ is satisfied. To show this, consider first the very last clique of the first phase. Excluding the last vertex of this clique, for which OPT does not need to pay for its rejection cost (since it accepts and colors it), the clique has a total rejection cost of at most ε. As for all preceding cliques, OPT pays the rejection cost only for all vertices except for the last one, whereas the online algorithm pays only for the last vertex, and $\varepsilon \cdot \varepsilon_k = \sum_{j=k+1}^{\Delta+1} \varepsilon_j$ holds.

Assume that the instance stops while we are in the first phase. Let x denote the total rejection cost of all the vertices. Then, the optimal solution of the instance is to reject all the vertices with a total cost of x. The total rejection cost of all the vertices in the last clique is less than 2ε, and therefore the online algorithm has a total rejection cost of at least $\frac{x-2\varepsilon}{1+\varepsilon}$. Since we stop the instance in the first phase the online algorithm has opened $\Delta + 1$ colors, and therefore its total cost is $\Delta + 1 + \frac{x-2\varepsilon}{1+\varepsilon} \geq \Delta + 1 + x - 3\varepsilon$, where the inequality holds as $x \leq 1$. We next note that for all $x \leq 1$ and $\varepsilon \leq \frac{1}{3}$ the following holds $\frac{\Delta+1-3\varepsilon+x}{x} \geq \Delta + 2 - 3\varepsilon$. Therefore, if $\varepsilon \leq \frac{\delta}{3}$, and by the assumption that our online algorithm has a competitive ratio of at most $\Delta + 2 - \delta$, we get that the algorithm does not open the $\Delta + 1$-th color in the first phase. Hence the total rejection cost that the online solution pays for the vertices in the first phase is at least $\frac{1-2\varepsilon}{1+\varepsilon}$.

Recall that R_{onl} is the total rejection cost that the online algorithm pays. If $R_{onl} \geq 3(\Delta + 2)$, then $R_{onl} \leq 3(\Delta + 2) + \varepsilon$. Assume that $\varepsilon \leq \frac{1}{3(\Delta+2)+1}$, and therefore the competitive ratio of the resulting algorithm is at least $\frac{R_{onl}}{1+R_{opt}} \geq \frac{R_{onl}}{1+\varepsilon+\varepsilon R_{onl}} \geq \frac{R_{onl}}{1+\varepsilon+1}$, where the last inequality holds as $\varepsilon \cdot R_{onl} \leq \varepsilon \cdot (3(\Delta + 2) + 1) \leq 1$. Therefore, the competitive ratio of the algorithm is at least $\frac{R_{onl}}{3} \geq \frac{3(\Delta+2)}{3} = \Delta + 2$, and this contradicts the fact that the competitive ratio of the algorithm is at most $\Delta + 2 - \delta$. Therefore, the total rejection cost that the online algorithm pays satisfies $\frac{1-2\varepsilon}{1+\varepsilon} \leq R_{onl} \leq 3(\Delta + 2)$, and the algorithm pays an additional $\Delta + 1$ units of cost for opening at least $\Delta + 1$ colors.

Therefore, the total cost paid by the online algorithm is at least $\Delta + 1 + \frac{1-2\varepsilon}{1+\varepsilon} \geq \Delta + 2 - 3\varepsilon \geq \Delta + 2 - \frac{\delta}{2}$ (assuming $\varepsilon < \frac{\delta}{6}$), and the total cost paid by OPT is at most $1 + R_{opt} \leq 1 + \varepsilon + \varepsilon R_{onl} \leq 1 + \varepsilon \cdot (7 + 3\Delta)$. The competitive ratio of the online algorithm is at least $\frac{\Delta+2-\frac{\delta}{2}}{1+\varepsilon\cdot(7+3\Delta)}$. Note that this last quantity should be at most $\Delta + 2 - \delta$. However, for $\varepsilon < \frac{\delta}{2(7+3\Delta)(\Delta+2-\delta)}$, the above constraint does not hold, and we reach a contradiction. □

Corollary 2. *There is no competitive online algorithm for the online coloring problem with rejections, even if the input graph is a disjoint union of cliques.*

Proof. We use the proof of Theorem 7, and note that where the maximum degree can be arbitrarily large the lower bound is arbitrarily large. □

However, the conclusion of Corollary 2 does not apply to split graphs as a union of two (or more) disjoint cliques is not a split graph. We note that the offline problem on split graphs has a polynomial time approximation scheme (and thus is approximable very well), whereas the online problem cannot be tackled. We can prove the following theorem.

Theorem 8. *There is no competitive online algorithm for the online coloring problem with rejections on split graphs.*

References

1. A. Bar-Noy, R. Motwani, and J. Naor. The greedy algorithm is optimal for on-line edge coloring. *Information Processing Letters*, 44(5):251–253, 1992.
2. A. Frank. On chain and antichain families of a partially ordered set. *Journal of Combinatorial Theory Series B*, 29:176–184, 1980.
3. M. R. Garey and D. S. Johnson. *Computers and intractability.* W. H. Freeman and Company, New York, 1979.
4. M. X. Goemans and D. P. Williamson. A general approximation technique for constrained forest problems. *SIAM J. Comput.*, 24(2):296–317, 1995.
5. A. Gyárfás and J. Lehel. On-line and first-fit colorings of graphs. *Journal of Graph Theory*, 12:217–227, 1988.
6. D. S. Johnson. The NP-completeness column: an ongoing guide. *Journal of Algorithms*, 6(3):434–451, 1985.
7. H. A. Kierstead. Coloring graphs on-line. In A. Fiat and Gerhard J. Woeginger, editors, *Online Algorithms - The State of the Art*, chapter 13, pages 281–305. Springer-Verlag, Berlin, 1998.
8. G. L. Nemhauser and L. Wolsey. Maximizing submodular set functions: formulations and analysis of algorithms. In *Studies of graphs and discrete programming*, pages 279–301. North-Holland, Amsterdam, 1972.
9. A. Schrijver. *Combinatorial optimization polyhedra and efficiency.* Springer-Verlag, 2003.
10. P. Slavík. Improved performance for the greedy algorithm for partial cover. *Information Processing Letters*, 64(5):251–254, 1997.
11. M. Yannakakis and F. Gavril. The maximum k-colorable subgraph problem for chordal graphs. *Information Processing Letters*, 24(2):133–137, 1987.

A Doubling Dimension Threshold $\Theta(\log \log n)$ for Augmented Graph Navigability*

Pierre Fraigniaud[1,**], Emmanuelle Lebhar[2,***], and Zvi Lotker[3]

[1] CNRS, Laboratoire de Recherche en Informatique (LRI),
Université Paris-Sud, 91405 Orsay, France
pierre@lri.fr
[2] LIP, École Normale Supérieure de Lyon, 46 allée d'Italie,
69364 Lyon Cedex 07, France
Emmanuelle.Lebhar@ens-lyon.fr
[3] Centrum voor Wiskunde en Informatica (CWI), Kruislaan 413,
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
Z.Lotker@cwi.nl

Abstract. In his seminal work, Kleinberg showed how to augment meshes using random edges, so that they become navigable; that is, greedy routing computes paths of polylogarithmic expected length between any pairs of nodes. This yields the crucial question of determining wether such an augmentation is possible for all graphs. In this paper, we answer negatively to this question by exhibiting a threshold on the doubling dimension, above which an infinite family of graphs cannot be augmented to become navigable whatever the distribution of random edges is. Precisely, it was known that graphs of doubling dimension at most $O(\log \log n)$ are navigable. We show that for doubling dimension $\gg \log \log n$, an infinite family of graphs cannot be augmented to become navigable. Finally, we complete our result by studying the special case of square meshes, that we prove to always be augmentable to become navigable.

Keywords: doubling dimension, small world, greedy routing.

1 Introduction

The *doubling dimension* [4,13,15] appeared recently as a key parameter for measuring the ability of networks to support efficient algorithms [7,14] or to realize

* The results contained in this paper were partially obtained when the two first authors were visiting CWI (Centrum voor Wiskunde en Informatica) in the Netherlands. Additional support from CWI.

** Supported by the projects PairAPair of the ACI Masses de Données, and FRAGILE of the ACI Sécurité Informatique. Additional support from the INRIA project "Grand Large".

*** Supported by the CNRS Locoglobo grant and the project PairAPair of the ACI Masses de Données.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 376–386, 2006.

specific tasks efficiently [1,2,6,24]. Roughly speaking, the doubling dimension of a graph G is the smallest d such that, for any integer $r \geq 1$, and for any node $u \in V(G)$, the ball $B(u, 2r)$ centered at u and of radius $2r$ can be covered by at most 2^d balls $B(u_i, r)$ centered at nodes $u_i \in V(G)$. (This definition can be extended to any metric, and, for instance, $\mathbf{Z}^d$ with the ℓ_1 norm is of doubling dimension d). In particular, the doubling dimension has an impact on the analysis of the small world phenomenon [22], precisely on the expected performances of greedy routing in *augmented* graphs [17].

An augmented graph is a pair (G, φ) where G is an n-node graph, and φ is a collection of probability distributions $\{\varphi_u, u \in V(G)\}$. Every node $u \in V(G)$ is given an extra link pointing to some node v, called the *long range contact* of u. The link from a node to its long range contact is called a *long range link*. The original links of the graph are called *local links*. The long range contact of u is chosen at random according to φ_u as follows: $\Pr\{u \to v\} = \varphi_u(v)$. Greedy routing in (G, φ) is the oblivious routing protocol where the routing decision taken at the current node u for a message of destination t consists in (1) selecting a neighbor v of u that is the closest to t according to the distance in G (this choice is performed among all neighbors of u in G and the long range contact of u), and (2) forwarding the message to v. This process assumes that every node has a knowledge of the distances in G, or at least a good approximation of them. On the other hand, every node is unaware of the long range links added to G, but its own long range link. Hence the nodes have no notion of the distances in the augmented graph. Note that the knowledge of the distances in the underlying graph G is a reasonable assumption when, for instance, G is a network in which distances can be computed from the coordinates of the nodes (e.g., in meshes, as in [17]).

An infinite family of graphs $\mathcal{G} = \{G^{(i)}, i \in I\}$ is *navigable* if there exists a family $\Phi = \{\varphi^{(i)}, i \in I\}$ of collections of probability distributions, and a function $f(n) \in O(\text{polylog}(n))$ such that, for any $i \in I$, greedy routing in $(G^{(i)}, \varphi^{(i)})$ performs in at most $f(n^{(i)})$ expected number of steps where $n^{(i)}$ is the order of the graph $G^{(i)}$. More precisely, for any pair of nodes (s, t) of $G^{(i)}$, the expected number of steps $\mathbb{E}(\varphi^{(i)}, s, t)$ for traveling from s to t using greedy routing in $(G^{(i)}, \varphi^{(i)})$ is at most $f(n^{(i)})$. The *greedy diameter* of $(G^{(i)}, \varphi^{(i)})$ is defined as $\max_{s,t \in G^{(i)}} \mathbb{E}(\varphi^{(i)}, s, t)$.

In his seminal paper, Kleinberg [17] proved that, for any fixed integer $d \geq 1$, the family of d-dimensional meshes is navigable. Duchon et al [8] generalized this result by proving that any infinite family of graphs with bounded growth is navigable. Fraigniaud [11] proved that any infinite family of graphs with bounded treewidth is navigable. Finally, Slivkins [24] recently related navigability to doubling dimension by proving that any infinite family of graphs with doubling dimension at most $O(\log \log n)$ is navigable. All these results naturally lead to the question of whether all graphs are navigable.

Let $\delta : \mathbf{N} \mapsto \mathbf{N}$, let $\mathcal{G}_{n,\delta(n)}$ be the class of n-node graphs with doubling dimension at most $\delta(n)$, and let $\mathcal{G}_\delta = \cup_{n \geq 1} \mathcal{G}_{n,\delta(n)}$. By rephrasing Slivkins result [24],

we get that $\mathcal{G}_\delta$ is navigable for any function δ bounded from above by $c \log\log n$ for some constant $c > 0$. This however lets open the case of graphs of larger doubling dimensions, namely the cases of all families $\mathcal{G}_\delta$ where δ is satisfying $\delta(n) \gg \log\log n$.

1.1 Our Results

We prove a threshold of $\delta(n) = \Theta(\log\log n)$ for the navigability of $\mathcal{G}_\delta$: below a certain function δ, $\mathcal{G}_\delta$ is navigable, while above it $\mathcal{G}_\delta$ is not navigable. More precisely, we prove that, for any function δ satisfying $\lim_{n\to\infty}(\log\log n)/\delta(n) = 0$, $\mathcal{G}_\delta$ is not navigable. Hence, the result in [24] is essentially the best that can be achieved by considering only the doubling dimension of graphs.

Our negative result requires to prove that for an infinite family of graphs in $\mathcal{G}_\delta$, *any* distribution of the long range links leaves the expected number of steps of greedy routing above any polylogarithmic for some pairs of source and target. For this purpose, we exhibit graphs presenting a very high number of possible "directions" for a long range link to go. By a counting argument, we show that there exist pairs of source and target at distance greater than any polylogarithm, between which greedy routing does not use any long range link, whatever their distribution is. In other words, we exhibit an infinite family of graphs with non polylogarithmic greedy diameter for any augmentation. This negative results answers a question asked in [11,18].

We also prove a somehow counter intuitive result by showing that a supergraph of a navigable graph is not necessarily navigable. In particular, we show that all square meshes are navigable, for all dimensions. Specifically, we prove that although the family of non navigable graphs that we use to disprove the navigability of all graphs contains the standard square meshes of dimension δ as subgraphs, this latter family of graphs is navigable.

1.2 Related Works

Kleinberg showed that greedy routing performs in $O(\log^2 n)$ steps between any pair of nodes on d-dimensional meshes augmented by the d-harmonic distribution. I.e. the greedy diameter of these augmented meshes is $O(\log^2 n)$. Since then, several results have been developed to tighten the analysis of greedy routing on randomly augmented networks. Precisely, Barrière et al. [5] showed that the greedy diameter of the d-dimensional meshes augmented by the d-harmonic distribution is $\Theta(\log^2 n)$. In the special case of rings, Aspnes et al. [3] proved a lower bound on the greedy diameter of $\Omega(\log^2 n/\log\log n)$ for *any* augmentation. For paths, Flammini et al. [9] recently showed a lower bound on the greedy diameter of $\Omega(\log^2 n)$ in the case of symmetric and distance monotonic augmentations. Martel and Nguyen [21] showed however that the (standard) diameter of all these networks augmented by the harmonic distribution is $\Theta(\log n)$. In another perspective, several authors developed decentralized algorithms for the d-dimensional mesh augmented by the d-harmonic distribution.

Lebhar and Schabanel [19] presented a decentralized algorithm which performs in $O(\log n(\log\log n)^2)$ expected number of steps in this graph. The algorithm Neighbor-Of-Neighbor presented by Manku et al. [20] performs in $O(\frac{1}{k\log k}(\log n)^2)$ expected number of steps, where k is the number of long range links per node in the mesh. Assuming some extra knowledge on the long range links, Fraigniaud et al. [12] described an oblivious routing which performs in $O((\log n)^{1+1/d})$ expected number of steps. Finally, Martel and Nguyen [21] presented a non oblivious routing protocol achieving the same performances under the same assumption as in [12].

1.3 Organization of the Paper

The paper is organized as follows: Section 2 presents the main result of the paper by exhibiting the non navigability of graphs with doubling dimension $\gg \log\log n$. In Section 3, we study the special case of square meshes, and prove that they are all navigable.

2 Non Navigable Graphs

In this section, we prove that the result in [24] is essentially the best that can be achieved as far as doubling dimension is concerned.

Theorem 1. *Let* $\delta : \mathbf{N} \mapsto \mathbf{N}$ *be such that* $\lim_{n\to\infty} \frac{\log\log n}{\delta(n)} = 0$. *Then* $\mathcal{G}_\delta$ *is not navigable.*

Informally, the argument of the proof is that a doubling dimension $\gg \log\log n$ implies that the number of possible "directions" where a random link can go is greater than any polylogarithm of n. Therefore, for any trial of the long range links, there always exist a direction for which these long links do not help in the sense that there exist a source and a target between which greedy routing does not use any long range link.

Proof. We show that there exists an infinite family of graphs $\{G^{(n)}, n \geq 1\}$ indexed by their number of vertices, such that $G^{(n)} \in \mathcal{G}_{n,\delta(n)}$ and for any family $\Phi = \{\varphi^{(n)}, n \geq 1\}$ of collections of probability distributions, greedy routing in $(G^{(n)}, \varphi^{(n)})$ performs in an expected number of steps $t(n) \notin O(\text{polylog}(n))$ for some pairs of source and target.

Let $d : \mathbf{N} \mapsto \mathbf{N}$ be such that $d \leq \delta$, $\lim_{n\to\infty} \frac{\log\log n}{d(n)} = 0$, and $d(n) \leq \varepsilon\sqrt{\log n}$ for some $0 < \varepsilon < 1$. For the sake of simplicity, assume that $p = n^{1/d(n)}$ is integer. $G^{(n)}$ is the graph of n nodes consisting of $p^{d(n)}$ nodes labeled $(x_1, \ldots, x_{d(n)})$, $x_i \in \mathbf{Z}_p$. Node $(x_1, \ldots, x_{d(n)})$ is connected to all nodes $(x_1 + a_1, \ldots, x_{d(n)} + a_{d(n)})$ where $a_i \in \{-1, 0, 1\}$, $i = 1, \ldots, d(n)$, and all operations are taken modulo p (cf. Figure 1). Note that, by construction of $G^{(n)}$, the distance between two nodes $y = (y_1, \ldots, y_{d(n)})$ and $z = (z_1, \ldots, z_{d(n)})$ is $\max_{1\leq i\leq d(n)} \min(|y_i - z_i|, p - |y_i - z_i|)$. Hence, the diameter of $G^{(n)}$ is $\lfloor p/2 \rfloor$.

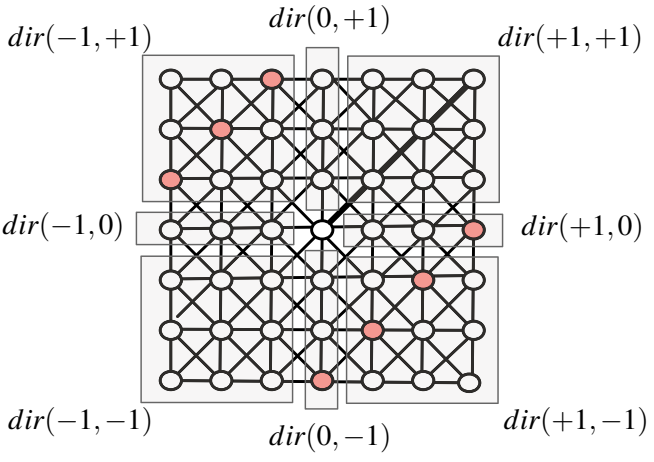


Fig. 1. Example of graph $G^{(n)}$ defined in proof of Theorem 1 with $d(n) = 2$. Grey areas represent the various *directions* for the central node. The bold line represents a *diagonal* for the central node. Colored nodes belongs to a *line*.

Claim. $G^{(n)} \in \mathcal{G}_{n,\delta(n)}$.

Clearly $G^{(n)}$ has n nodes. We prove that $G^{(n)}$ has doubling dimension $d(n)$, therefore at most $\delta(n)$. Let $\mathbf{0} = (0, \ldots, 0)$. The ball $B(\mathbf{0}, 2r)$ can be covered by $2^{d(n)}$ balls of radius r, centered at the $2^{d(n)}$ nodes $(x_1, \ldots, x_{d(n)})$, $x_i \in \{-r, +r\}$ for any $i = 1, \ldots, d(n)$. Hence the doubling dimension of $G^{(n)}$ is at most $d(n)$. On the other hand, $|B(\mathbf{0}, 2r)| = (4r+1)^{d(n)}$ and $|B(\mathbf{0}, r)| = (2r+1)^{d(n)}$. Thus at least $(4r+1)^{d(n)}/(2r+1)^{d(n)}$ balls are required to cover $B(\mathbf{0}, 2r)$, since in $G^{(n)}$, for any node u and radius r, $|B(u, r)| = |B(\mathbf{0}, r)|$. This ratio can be rewritten as $2^{d(n)}(1 - \frac{1}{2(2r+1)})^{d(n)}$. For $2r = \frac{n^{1/d(n)}}{5}$, since $d(n) \leq \sqrt{\log n}$, we get that $(2r+1) > \frac{2^{\sqrt{\log n}}}{5} > d(n)$ for $n \geq n_0$, $n_0 \geq 1$. Then, for $n \geq n_0$,

$$\begin{aligned}\left(1 - \frac{1}{2(2r+1)}\right)^{d(n)} &> \left(1 - \frac{1}{2d(n)}\right)^{d(n)} \\ &= 2^{d(n)\log\left(1-\frac{1}{2d(n)}\right)} \\ &\geq 2^{d(n)\left(-\frac{1}{2d(n)} - \frac{4}{4(d(n))^2}\right)} = 2^{-\frac{1}{2}-\frac{1}{d(n)}}\end{aligned}$$

There exists $n_1 \geq n_0$, such that $2^{-\frac{1}{2}-\frac{1}{d(n)}} > \frac{1}{2}$ for $n \geq n_1$. Then, for $n \geq n_1$, $|B(\mathbf{0}, 2r)|/|B(\mathbf{0}, r)| > 2^{d(n)-1}$. Thus the doubling dimension of $G^{(n)}$ is at least $d(n)$, which proves the claim. ◇

Definition 1. *For any node* $u = (u_1, \ldots, u_{d(n)})$*, and for any* $D = (\nu_1, \ldots, \nu_{d(n)}) \in \{-1, 0, +1\}^{d(n)}$*, we call* direction *the set of nodes*

$$dir_u(D) = \{v = (v_1, \ldots, v_{d(n)}) : v_i = (u_i + \nu_i \cdot x_i) \bmod p,\ 1 \leq x_i \leq \lfloor p/2 \rfloor\}.$$

Note that, for any u, the directions $\mathrm{dir}_u(D)$ for $D \in \{-1, 0, +1\}^{d(n)}$ partition the nodes of $G^{(n)}$ (see Figure 1). There are obviously $3^{d(n)}$ directions, and the $2^{d(n)}$ directions defined on $\{-1, +1\}^{d(n)}$ have all the same cardinality.

Definition 2. *For any node $u = (u_1, \ldots, u_{d(n)})$, and for any $D = (\nu_1, \ldots, \nu_{d(n)}) \in \{-1, +1\}^{d(n)}$, we call* diagonal *the set of nodes*

$$diag_u(D) = \{v = (v_1, \ldots, v_{d(n)}) : v_i = (u_i + \nu_i \cdot x) \bmod p,\ 1 \leq x \leq \lfloor p/2 \rfloor\}.$$

The next claim shows that long range links are useless for greedy routing along a diagonal if they are not going in the direction of the diagonal.

Claim. Let u be any node and let v be the long range contact of u for some distribution $\varphi^{(n)}$ of the long range links. Assume $v \in \mathrm{dir}_u(D)$ and let $t \in \mathrm{diag}_u(D')$ for $D, D' \in \{-1, +1\}^{d(n)}$, $D \neq D'$. Greedy routing from u to t does not use the long range link (u, v).

Let $u = (u_1, \ldots, u_{d(n)})$, $v = (v_1, \ldots, v_{d(n)})$ and $t = (t_1, \ldots, t_{d(n)})$. Since $t \in \mathrm{diag}_u(D')$, there exists $x \in \{1, \ldots, \lfloor p/2 \rfloor\}$ such that $|t_i - u_i| = x$ for all $1 \leq i \leq d(n)$. Since $D \neq D'$, there exists $j \in \{1, \ldots, d(n)\}$ such that:

$$t_j = u_j + \alpha \cdot x \tag{1}$$

$$v_j = u_j - \alpha \cdot y, \tag{2}$$

for some $\alpha \in \{-1, +1\}$ and $y \in \{1, \ldots, \lfloor p/2 \rfloor\}$. Then,

$$\mathrm{dist}(v, t) \geq |t_j - v_j| = x + y > x = \mathrm{dist}(u, t).$$

Therefore greedy routing from u to t does not use the long range link (u, v), which proves the claim. ◇

Consider now a distribution $\varphi^{(n)}$ of long range links that belongs to some given collection of probability distributions $\Phi = \{\varphi^{(n)}, n \geq 1\}$. We prove that routing on the diagonal is hard. More precisely, let s be a source node and t be a target node, $t \in \mathrm{diag}_s(D)$ for some D. If any node $u \in \mathrm{diag}_s(D)$ between s and t has its long range contact $v \in \mathrm{dir}_u(D_v)$ for some $D_v \neq D$, then, from the previous claim, greedy routing from s to t does not use any long range link and thus takes $\mathrm{dist}(s, t)$ steps. We prove that this phenomenon occurs for at least one pair (s, t) such that $\mathrm{dist}(s, t) \geq 2^{d(n)} - 3$.

Definition 3. *An* interval *$I = [a, b]$ is a connected subset of a diagonal. Precisely, $[a, b]$ is an interval of $diag_a(D)$ if $b \in diag_a(D)$ and*

$$I = \{c \in diag_a(D) \,|\, \mathrm{dist}(a, c) + \mathrm{dist}(c, b) = \mathrm{dist}(a, b)\}.$$

We say that an interval I of $\mathrm{diag}_u(D)$ is *good* if there exists $x \in I$ such that the long range contact y of x satisfies $y \in \mathrm{dir}_x(D)$.

Definition 4. *A* line L *of* $G^{(n)}$ *in direction* $D \in \{-1,+1\}^{d(n)}$ *is a maximal subset of* $V(G^{(n)})$ *such that for any two nodes* $u, v \in L$, *we have*

$$diag_u(D) \cap diag_v(D) \neq \emptyset.$$

The set of all the lines in the same direction D partitions $G^{(n)}$ into n/p lines of size p.

Let us partition each line into p/X disjoint intervals of same length X. This results into n/X intervals per direction, thus in total into a set S of $\frac{n}{X} \cdot 2^{d(n)}$ intervals of length X, since there are $2^{d(n)}$ directions defined in $\{-1,+1\}^{d(n)}$. We show that if X is too small, then there is at least one of all the intervals in S which is not good.

There is a one-to-one mapping between intervals and nodes in the following sense. Each good interval $I = [a,b] \in S$ must contain a node u whose long range contact v satisfies $v \in \mathrm{dir}_u(D)$. The node u is called the *certificate* for I. Node u cannot be the certificate of any other interval $J \in S$ with $J \neq I$, even for those such that $J \cap I \neq \emptyset$ when I and J are in two distinct directions.

We have $2^{d(n)} \cdot \frac{n}{X}$ intervals in S. Since a certificate certifies the goodness of exactly one interval, $2^{d(n)} \cdot \frac{n}{X}$ has to be at most n, that is: $X \geq 2^{d(n)}$. By the pigeonhole principle, if $X < 2^{d(n)}$, there is one interval $I = [s,t] \in S$ which is not good. From Claim 2, greedy routing from s to t takes $X - 1$ steps.

Since $d(n) \leq \varepsilon\sqrt{\log n}$, we have:

$$p = n^{1/d(n)} \geq 2^{\frac{1}{\varepsilon}\sqrt{\log n}} \geq 2^{\varepsilon\sqrt{\log n}} - 2 \geq 2^{d(n)} - 2.$$

Therefore, the value $X = 2^{d(n)} - 2$ can be considered for our partitioning. In this case, we obtain that greedy routing from s to t takes $2^{d(n)} - 3$ steps.

We complete the proof of the theorem by proving the following claim.

Claim. $2^{d(n)} \notin O(\mathrm{polylog}\, n)$.

Let $\alpha \geq 1$, we have:

$$\frac{\log^\alpha n}{2^{d(n)}} = 2^{\alpha \log\log n - d(n)} = 2^{\alpha\, d(n)\left(\frac{\log\log n}{d(n)} - \frac{1}{\alpha}\right)}.$$

Since $\lim_{n\to\infty} \frac{\log\log n}{d(n)} = 0$, there exists $n_1 \geq 1$ such that for any $n \geq n_1$, $\left(\frac{\log\log n}{d(n)} - \frac{1}{\alpha}\right) \leq -\frac{1}{2\alpha}$, and thus $\frac{\log^\alpha n}{2^{d(n)}} \leq 2^{-d(n)/2}$. Moreover, $d(n) \geq \log\log n$, then, for $n \geq n_1$,

$$\frac{\log^\alpha n}{2^{d(n)}} \leq 2^{-\frac{\log\log n}{2}} = o(1).$$

In other words, $2^{d(n)}$ is not a polylogarithm of n, which proves the claim. ◇

Remark. Note that the proof of Theorem 1 does not assume independent trials for the long range links.

3 Navigability of Meshes

The family of non navigable graphs defined in the proof of Theorem 1 contains the standard square meshes of dimension $d(n)$ as subgraphs, where $d(n) \gg \log \log n$. Nevertheless, and somehow counter intuitively, a supergraph of a navigable graph is not necessarily navigable. In this section, we illustrate this phenomenon by focusing on the special case of d-dimensional meshes, the first graphs that were considered for the analysis of navigable networks [17]. Precisely, we show that any d-dimensional torus $C_{n^{1/d}} \times \ldots C_{n^{1/d}}$ is navigable: either it has a polylogarithmic diameter, or it admits a distribution of links such that greedy routing computes paths of polylogarithmic length. This result has partially been proven in [9] for the case of constant dimensions. We give here a complete proof that holds for any dimension.

Theorem 2. *For any positive function $d(n)$, the n-node $d(n)$-dimensional torus is navigable.*

Proof. We construct a random link distribution φ as follows. Let $u = (u_1, \ldots, u_{d(n)})$ and $v = (v_1, \ldots, v_{d(n)})$ be two nodes. If they differ in more than one coordinate, then $\varphi_u(v) = 0$; otherwise, i.e. they differ in only one coordinate, say the ith, then:

$$\varphi_u(v) = \frac{1}{d(n)} \cdot \frac{1}{2H_k} \cdot \frac{1}{|u_i - v_i|},$$

where $k = \frac{n^{1/d}}{2}$ and $H_k = \sum_{j=1}^{k} \frac{1}{j}$ is the harmonic sum. Note that this distribution corresponds to:

- picking a dimension uniformly at random (probability $\frac{1}{d(n)}$ to pick dimension i)
- and to draw a long-range link on this axis according to the 1-harmonic distribution over distances ($\frac{1}{2H_k}$ is the normalizing coefficient for this distribution), which is the distribution given by Kleinberg to make the 1-dimensional torus navigable.

Let now $s = (s_1, \ldots, s_{d(n)})$ and $t = (t_1, \ldots, t_{d(n)})$ be a pair of source and target in the mesh. Assume that the current message holder during an execution of greedy routing is $x = (x_1, \ldots, x_{d(n)})$, at distance X from t. The probability that x has a long range link to some node $w = (x_1, \ldots, x_{i-1}, w_i, x_{i+1}, \ldots, x_{d(n)})$, $1 \leq i \leq d(n)$ such that $|t_i - w_i| \leq |t_i - x_i|/2$, is at least $\sum_{1 \leq i \leq d(n)} \frac{1}{3d(n)H_k} = \frac{1}{3H_k}$, along the same analysis as the analysis of Kleinberg one dimensional model, summing over the dimensions. If such a link is found, it is always preferred to the local contact of x that only reduces one of the coordinate by 1. Thus, after at most $3H_k$ steps on expectation, one of the coordinates has been divided by two. Note that since long range links only get to nodes that differs in a single coordinate from their origin, further steps cannot increase $|x_i - t_i|$ for any $1 \leq i \leq d(n)$, x being the current message holder. Repeating the analysis for all coordinates, we thus get that after $3d(n)H_k$ steps on expectation, all

the coordinates have been divided by at least two, and so the current distance to the target is at most $X/2$. Finally, the algorithm reaches t after at most $3d(n)H_k \log(dist(s,t))$ steps on expectation, which is $O\left((\log^2 n)/d(n)\right)$.

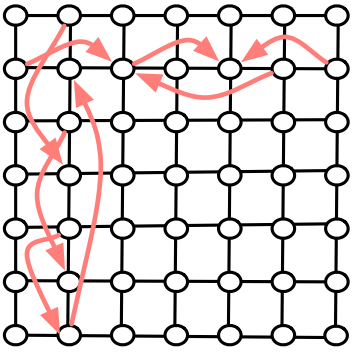


Fig. 2. Example of 2-dimensional mesh augmented as in proof of Theorem 2: 1-harmonic distribution of links on each axis. Bold links are long range links, they are not all represented.

Remark. Note that our example of non-navigability in Section 3 may appear somehow counter intuitive in contrast to our latter construction of long range links on meshes. Indeed, why not simply repeating such a construction on the graph $G^{(n)}$ defined in the proof Theorem 1? That is, why not selecting long range contacts on each "diagonal" using the 1-harmonic distribution, in which case greedy routing would perform efficiently between pairs (s,t) on the diagonals? This cannot be done however because, to cover all possible pairs (s,t) on the diagonals, $2^{d(n)}$ long range links per node would be required, which is larger than any polylogarithm of n when $d(n) \gg \log\log n$.

4 Conclusion

The increasing interest in graphs and metrics of bounded doubling dimension arises partially from the hypothesis that large real graphs do present a low doubling dimension (see, e.g., [10,16] for the Internet). Under such an hypothesis, efficient compact routing schemes and efficient distance labeling schemes designed for bounded doubling dimension graphs would have promising applications. On the other hand, the navigability of a network is actually closely related to the existence of efficient compact routing and distance labeling schemes on the network. Indeed, long range links can be turned into small labels, e.g. via the technique of rings of neighbors [24]. Interestingly enough, our paper emphasizes that the small doubling dimension hypothesis of real networks is crucial. Indeed, for doubling dimension above $\log\log n$, networks may become not navigable. It would therefore be important to study precisely to which extent real networks do present a low doubling dimension.

In a more general framework, our result of non navigability shows that the small world phenomenon, in its algorithmic definition of navigability, is not only due to the good spread of additional links over distances in a network, but is also highly dependent of the base metric itself, in particular in terms of dimensionality.

Peleg recently proposed the more general question of f-navigability. For a function f, we say that a n-node graph G is f-navigable if there exists a distribution φ of long range links such that the greedy diameter of the augmented graph (G, φ) is at most $f(n)$. From [23], all n-node graphs are $\sqrt{n}$-navigable by giving an uniform random distribution of the long range links. From Theorem 1, we get as a corollary that, for all graphs to be f-navigable, $f(n) = \Omega(2^{\sqrt{\log n}})$. It thus remains to close the gap between these upper and lower bounds for the f-navigability of arbitrary graphs.

Acknowledgments. We are thankful to Ph. Duchon for having pointed to us an error in an earlier version of this paper.

References

1. I. Abraham, C. Gavoille, A. V. Goldberg, and D. Malkhi. Routing in networks with low doubling dimension. In *26th International Conference on Distributed Computing Systems (ICDCS)*, to appear, 2006.
2. I. Abraham and D. Malkhi. Name independent routing for growth bounded networks. In 17^{th} *Annual ACM Symposium on Parallel Algorithms and Architecture (SPAA)*, pages 49–55, 2005.
3. J. Aspnes, Z. Diamadi, and G. Shah. Fault-tolerant routing in peer-to-peer systems. In *In 21st ACM Symp. on Principles of Distributed Computing (PODC)*, pages 223–232, 2002.
4. P. Assouad. Plongements lipshitzien dans $\mathbf{R}^n$. *Bull. Soc. Math.*, 111(4):429–448, 1983.
5. L. Barrière, P. Fraigniaud, E. Kranakis, and D. Krizanc. Efficient routing in networks with long range contacts. In *LNCS Proceedings of 15th International Symposium on Distributed Computing (DISC)*, volume 2180, pages 270–284, 2001.
6. H.T-H. Chan, A. Gupta, B. M. Maggs, and S. Zhou. On hierarchical routing in doubling metrics. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms (SODA)*, pages 762–771, 2005.
7. K. L. Clarkson. *Nearest-Neighbor Methods for Learning and Vision: Theory and Practice*, chapter Nearest-neighbor searching and metric space dimensions. (Survey). MIT Press, T. Darrell, P. Indyk, G. Shakhnarovich, and P. Viola, editors, 2006.
8. P. Duchon, N. Hanusse, E. Lebhar, and N. Schabanel. Could any graph be turned into a small world? *Theoretical Computer Science*, Special issue on Complex Networks, volume 355(1), pages 96–103, 2006.
9. M. Flammini, L. Moscardelli, A. Navarra, and S. Perennes. Asymptotically optimal solutions for small world graphs. In *LNCS Proceedings of the* 19*th International Symposium on Distributed Computing (DISC)*, volume 3724, pages 414–428, 2005.
10. M. Fomenkov, K. Claffy, B. Huffaker, and D. Moore. Macroscopic internet topology and performance measurements from the dns root name servers. In *USENIX LISA*, 2001.

11. P. Fraigniaud. Greedy routing in tree-decomposed graphs: a new perspective on the small-world phenomenon. In *Proceedings of the 13th Annual European Symposium on Algorithms (ESA)*, pages 791–802, 2005.
12. P. Fraigniaud, C. Gavoille, and C. Paul. Eclecticism shrinks even small worlds. In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing (PODC)*, pages 169–178, 2004.
13. A. Gupta, R. Krauthgamer, and J.R. Lee. Bounded geometries, fractals, and low-distortion embeddings. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 534–543, 2003.
14. S. Har-Peled and M. Mendel. Fast construction of nets in low dimensional metrics and their applications. In *Proceedings of the 21th ACM Symposium on Computational Geometry (SoCG)*, pages 150–158, 2005.
15. J. Heinonen. *Lectures on analysis on metric spaces*. Springer-Verlag, New York, 2001.
16. D. R. Karger and M. Ruhl. Finding nearest-neighbors in growth-restricted metrics. In *Proceedings of the 34th annual ACM symposium on Theory of computing (STOC)*, pages 741–750, 2002.
17. J. Kleinberg. The Small-World Phenomenon: An Algorithmic Perspective. In *Proceedings of the 32nd ACM Symposium on Theory of Computing (STOC)*, pages 163–170, 2000.
18. J. Kleinberg. Complex networks and decentralized search algorithm. In *Proceedings of the International Congress of Mathematicians (ICM)*, 2006. To appear.
19. E. Lebhar and N. Schabanel. Close to optimal decentralized routing in long-range contact networks. *Theoretical Computer Science special issue on ICALP'04*, 348(2-3):294–310, 2005.
20. G. S. Manku, M. Naor, and U. Wieder. Know thy neighbor's neighbor: the power of lookahead in randomized p2p networks. In *Proceedings of the 36th ACM Symposium on Theory of Computing (STOC)*, pages 54–63, 2004.
21. C. Martel and V. Nguyen. Analyzing Kleinberg's (and other) small-world models. In *In 23rd ACM Symp. on Principles of Distributed Computing (PODC)*, pages 178–187, 2004.
22. S. Milgram. The small world problem. *Psychology Today*, 61(1), 1967.
23. D. Peleg. Private communication, 2006.
24. A. Slivkins. Distance estimation and object location via rings of neighbors. In *Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 41–50, 2005.

Violator Spaces: Structure and Algorithms*

Bernd Gärtner[1], Jiří Matoušek[2], Leo Rüst[1], and Petr Škovroň[2]

[1] Institute of Theoretical Computer Science, ETH Zurich, 8092 Zurich, Switzerland
{gaertner, ruestle}@inf.ethz.ch
[2] Department of Applied Mathematics and Institute of Theoretical Computer Science, Charles University, Malostranské nám. 25, 118 00 Praha 1, Czech Republic
{matousek, xofon}@kam.mff.cuni.cz

Abstract. Sharir and Welzl introduced an abstract framework for optimization problems, called *LP-type problems* or also *generalized linear programming problems*, which proved useful in algorithm design. We define a new, and as we believe, simpler and more natural framework: *violator spaces*, which constitute a proper generalization of LP-type problems. We show that Clarkson's randomized algorithms for low-dimensional linear programming work in the context of violator spaces. For example, in this way we obtain the fastest known algorithm for the P-*matrix generalized linear complementarity problem* with a constant number of blocks. We also give two new characterizations of LP-type problems: they are equivalent to *acyclic* violator spaces, as well as to *concrete* LP-type problems (informally, the constraints in a concrete LP-type problem are subsets of a linearly ordered ground set, and the value of a set of constraints is the minimum of its intersection).

1 Introduction

The framework of LP-type problems, invented by Sharir and Welzl in 1992 [2], has become a well-established tool in the field of geometric optimization. Its origins are in linear programming: Sharir and Welzl developed a randomized variant of the dual simplex algorithm for linear programming and showed that this algorithm actually works for a more general class of problems they called LP-type problems.

For the theory of linear programming, this algorithm constituted an important progress, since it was later shown to be *subexponential* in the RAM model [3]. Together with a similar result independently obtained by Kalai [4], this was the first linear programming algorithm provably requiring a number of arithmetic operations subexponential in the dimension and number of constraints (independent of the precision of the input numbers).

For many other geometric optimization problems in fixed dimension, the algorithm by Sharir and Welzl was the first to achieve expected linear runtime,

* The first and the third author acknowledge support from the Swiss Science Foundation (SNF), Project No. 200020-112068/1. The fourth author acknowledges support from the Czech Science Foundation (GACR), Grant No. 201/05/H014. For a full paper see [1].

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 387–398, 2006.

simply because these problems could be formulated as LP-type problems. The class of LP-type problems for example includes the problem of computing the minimum-volume ball or ellipsoid enclosing a given point set in $\mathbb{R}^d$, and the problem of finding the distance of two convex polytopes in $\mathbb{R}^d$. Many other problems have been identified as LP-type problems over the years [3, 5, 6, 7, 8].

Once it is shown that a particular optimization problem is an LP-type problem, and certain algorithmic primitives are implemented for it, several efficient algorithms are immediately at our disposal: the Sharir–Welzl algorithm, two other randomized optimization algorithms due to Clarkson [9] (see [10, 11] for a discussion of how it fits the LP-type framework), a deterministic version of it [11], an algorithm for computing the minimum solution that violates at most k of the given n constraints [12], and probably more are to come in the future.

The framework of LP-type problems is not only a prototype for concrete optimization problems, it also serves as a mathematical tool by itself, in algorithmic [13, 14] and non-algorithmic contexts [15].

An (abstract) LP-type problem is given by a finite set H of *constraints* and a *value* $w(G)$ for every subset $G \subseteq H$. The values can be real numbers or, for technical convenience, elements of any other linearly ordered set. Intuitively, $w(G)$ is the minimum value of a solution that satisfies all constraints in G. The assignment $G \mapsto w(G)$ has to obey the axioms in the following definition.

Definition 1. *An* abstract LP-type problem *is a quadruple $(H, w, W, \leq)$, where H is a finite set, W is a set linearly ordered by $\leq$, and $w\colon 2^H \to W$ is a mapping satisfying the following two conditions for all $F \subseteq G \subseteq H$.*

Monotonicity: $w(F) \leq w(G)$, *and*
Locality: *for $w(F) = w(G)$ and all $h \in H$, $w(G) < w(G \cup \{h\})$ implies $w(F) < w(F \cup \{h\})$.*

As our running example, we will use the smallest enclosing ball problem, where H is a finite point set in $\mathbb{R}^d$ and $w(G)$ is the radius of the smallest ball that encloses all points of G. In this case monotonicity is obvious, while verifying locality requires the nontrivial but well known geometric result that the smallest enclosing ball is unique for every set.

It seems that the order $\leq$ of subsets is crucial; after all, LP-type problems model *optimization problems*, and indeed, the subexponential algorithm for linear programming and other LP-type problems [3] heavily relies on such an order.

A somewhat deeper look reveals that often, we only care whether two subsets have the *same* value, but not how they compare under the order $\leq$. The following definition is taken from [2]:

Definition 2. *Consider an abstract LP-type problem $(H, w, W, \leq)$. We say that $B \subseteq H$ is a* basis *if for all proper subsets $F \subset B$ we have $w(F) \neq w(B)$. For $G \subseteq H$, a* basis of G *is a minimal subset B of G with $w(B) = w(G)$.*

We observe that a minimal subset $B \subseteq G$ with $w(B) = w(G)$ is indeed a basis.

Solving an abstract LP-type problem $(H, w, W, \leq)$ means to find a basis of H. In the smallest enclosing ball problem, a basis of H is a minimal set B of

points such that the smallest enclosing ball of B has the same radius (and is in fact the same) as the smallest enclosing ball of H, $w(B) = w(H)$.

In defining bases, and in saying what it means to solve an LP-type problem, we therefore do not need the order $\leq$. The main contribution of this paper is that many of the things we can say or prove about LP-type problems do not require a concept of order. We formalize this by defining the new framework of *violator spaces*. Intuitively, a violator space is an LP-type problem without order. This generalization of LP-type problems is proper, and we can exactly characterize the violator spaces that "are" LP-type problems. In doing so, we also establish yet another equivalent characterization of LP-type problems that is closer to the applications than the abstract formulation of Definition 1.

These are our main findings on the structural side. Probably the most surprising insight on the algorithmic side is that Clarkson's algorithms [9] work for violator spaces of fixed dimension, leading to an expected linear-time algorithm for "solving" the violator space.

We give an application of Clarkson's algorithms in the more general setting by linking our new violator space framework to well-known abstract and concrete frameworks in combinatorial optimization. For this, we show that any *unique sink orientation* (USO) of the cube [16, 17, 18, 19, 20, 21, 22, 23, 24, 25] or the more general grid [17] gives rise to a violator space, but not to an LP-type problem in general. Grid USO capture some important problems like linear programming over products of simplices, *generalized linear complementarity problems* over P-matrices [17] or games like parity, mean-payoff, and simple stochastic games [26, 27, 28].

We show that we can find the sink in a unique sink orientation by solving the violator space. A concrete new result is obtained by applying this to P-matrix generalized linear complementarity problems. These problems are not known to be polynomial-time solvable, but NP-hardness would imply NP=co-NP [29, 17]. Since any P-matrix generalized linear complementarity problem gives rise to a unique sink orientation [17], we may use violator spaces and Clarkson's algorithms to solve the problem in expected linear time in the (polynomially solvable) case of a *fixed* number of *blocks*. This is optimal and beats all previous algorithms.

2 Structural Results

Concrete LP-type problems. Although intuitively one thinks about $w(G)$ as the value of an optimal solution of an optimization problem, the solution itself is not explicitly represented in Definition 1. In specific geometric examples, the constraints can usually be interpreted as a subset of some ground set X of points, and the optimal solution for G is the point with the smallest value in the intersection of all constraints in G. For example, in linear programming, the constraints are halfspaces, the value is given by the objective function, and the optimum is the point with minimum value in the admissible region, i.e., the intersection of the halfspaces. In order to have a unique optimum for every set of constraints (which is needed for w to define an LP-type problem), one assumes

that the points are linearly ordered by the value; for linear programming, we can always take the lexicographically smallest optimal solution, for instance.

Such an interpretation is possible for the smallest enclosing ball problem too, although it looks a bit artificial. Namely, the "points" of X are all balls in $\mathbb{R}^d$, where the ordering can be an arbitrary linear extension of the partial ordering of balls by radius. The "constraint" for a point $h \in H$ is the set of all balls containing h. The following definition captures this approach to LP-type problems.

Definition 3. *A* concrete LP-type problem *is a triple* $(X, \preceq, \mathcal{H})$*, where* X *is a set linearly ordered by* $\preceq$*,* $\mathcal{H}$ *is a finite multiset whose elements are subsets of* X*, and for any* $\mathcal{G} \subseteq \mathcal{H}$*, if the intersection* $\bigcap \mathcal{G} := \bigcap_{G \in \mathcal{G}} G$ *is nonempty, then it has a minimum element with respect to* $\preceq$ *(for* $\mathcal{G} = \emptyset$ *we define* $\bigcap \mathcal{G} := X$*).*

The definition allows $\mathcal{H}$ to be a multiset, i.e., a constraint set $A \subseteq X$ may be included several times. For example, in an instance of linear programming, some constraints can be the same, which we can reflect by this.

A similar model has been presented in [6] (mathematical programming problem). The slight difference is that it allows several points to have the same value but the constraints form a set rather than a multiset.

Bases are defined analogously to Definition 2.

Definition 4. *Consider a concrete LP-type problem* $(X, \preceq, \mathcal{H})$*. We say that* $\mathcal{B} \subseteq \mathcal{H}$ *is a* basis *if for all proper submultisets* $\mathcal{F} \subset \mathcal{B}$ *we have* $\min(\bigcap \mathcal{F}) \prec \min(\bigcap \mathcal{B})$*. For* $\mathcal{G} \subseteq \mathcal{H}$*, a* basis of $\mathcal{G}$ *is a minimal* $\mathcal{B} \subseteq \mathcal{G}$ *with* $\min(\bigcap \mathcal{B}) = \min(\bigcap \mathcal{G})$*.*

As before, a minimal $\mathcal{B} \subseteq \mathcal{G}$ with $\min(\bigcap \mathcal{B}) = \min(\bigcap \mathcal{G})$ is indeed a basis.

Given any concrete LP-type problem $\mathcal{P} = (X, \preceq, \mathcal{H})$, we obtain an abstract LP-type problem $P = (\mathcal{H}, w, X, \preceq)$ according to Definition 1 by putting $w(\mathcal{G}) = \min(\bigcap \mathcal{G})$ (or $w(\mathcal{G}) = +\infty$, if $\bigcap \mathcal{G}$ is empty), as is easy to check (proof omitted). It is clear that $\mathcal{B} \subseteq \mathcal{G}$ is a basis of $\mathcal{G}$ in P if and only if $\mathcal{B}$ is a basis of $\mathcal{G}$ in $\mathcal{P}$. We say that P is *basis-equivalent* to $\mathcal{P}$. (Some care is needed if $\mathcal{H}$ consists of elements with multiplicity bigger than 1, see [1] for details.)

Somewhat surprising is the converse, which we prove below in Theorem 1: Any abstract LP-type problem $(H, w, W, \leq)$ has a "concrete representation", that is, a concrete LP-type problem that is basis-equivalent to $(H, w, W, \leq)$.

Violator spaces. Let $(H, w, W, \leq)$ be an abstract LP-type problem. It is natural to define that a constraint $h \in H$ *violates* a set $G \subseteq H$ of constraints if $w(G \cup \{h\}) > w(G)$. For example, in the smallest enclosing ball problem, a point h violates a set G if it lies outside of the (unique) smallest ball enclosing G.

Definition 5. *The* violator mapping *of* $(H, w, W, \leq)$ *is defined by* $\mathsf{V}(G) = \{h \in H : w(G \cup \{h\}) > w(G)\}$*. Thus,* $\mathsf{V}(G)$ *is the set of all constraints violating* G*.*

It turns out that the knowledge of $\mathsf{V}(G)$ for all $G \subseteq H$ is enough to describe the "structure" of an LP-type problem. That is, while we cannot reconstruct W, $\leq$, and w from this knowledge, it is natural to consider two LP-type problems with the same mapping $\mathsf{V}: 2^H \to 2^H$ the same (isomorphic). Indeed, the algorithmic

primitives needed for implementing the Sharir–Welzl algorithm and the other algorithms for LP-type problems mentioned above can be phrased in terms of testing violation (does $h \in \mathsf{V}(G)$ hold for a certain set $G \subseteq H$?), and they never deal explicitly with the values of w.

We now introduce the notion of *violator space*:

Definition 6. *A* violator space *is a pair (H, V), where H is a finite set and V is a mapping $2^H \to 2^H$ such that:*

Consistency: $G \cap \mathsf{V}(G) = \emptyset$ *holds for all $G \subseteq H$, and*
Locality: *for all $F \subseteq G \subseteq H$ with $G \cap \mathsf{V}(F) = \emptyset$, we have $\mathsf{V}(G) = \mathsf{V}(F)$.*

Definition 7. *Consider a violator space (H, V). We say that $B \subseteq H$ is a* basis *if for all proper subsets $F \subset B$ we have $B \cap \mathsf{V}(F) \neq \emptyset$. For $G \subseteq H$, a* basis of G *is a minimal subset B of G with $\mathsf{V}(B) = \mathsf{V}(G)$.*

We will check in Section 3 that the violator mapping of an abstract LP-type problem satisfies the two axioms above. We actually show more: given an abstract LP-type problem $(H, w, W, \leq)$, the pair (H, V), with V being the violator mapping, is an *acyclic* violator space. (Acyclicity of a violator space will be defined later in Definition 9.) It turns out that acyclicity already characterizes the violator spaces obtained from LP-type problems, and thus any acyclic violator space can be represented as an LP-type problem (abstract or concrete). These equivalences are stated in our main theorem.

Theorem 1. *The axioms of abstract LP-type problems, of concrete LP-type problems, and of acyclic violator spaces are equivalent. More precisely, every problem in one of the three classes has a basis-equivalent problem in each of the other two classes.*

The construction is illustrated on a simple instance of linear programming (see Figure 1). Several more results concerning violator spaces have been achieved in the MSc. thesis of the fourth author [30].

3 Equivalence of LP-Type Problems and Acyclic Violator Spaces

Preliminaries on Violator Spaces. To show that every acyclic violator space (H, V) originates from some concrete LP-type problem, we need an appropriate linearly ordered set X of "points", and then we will identify the elements of H with certain subsets of X.

What set X will we take? Recall that for smallest enclosing balls, X is the set of all balls, and the subset for $h \in H$ is the subset of balls containing h. It is not hard to see that we may restrict X to smallest enclosing balls of *bases*; in fact, we may choose X as the set of bases, in which case the subset for h becomes the set of bases not violated by h.

This also works for general acyclic violator spaces, with bases suitably ordered. The only blemish is that we may get several minimal bases for $G \subseteq H$; for smallest enclosing balls, this corresponds to the situation in which several bases define the same smallest enclosing ball. To address this, we will declare such bases as equivalent and choose X as the set of all equivalence classes instead.

In the following, we fix a violator space (H, V). The set of all bases in (H, V) will be denoted by $\mathcal{B}$.

Definition 8. *$B, C \in \mathcal{B}$ are* equivalent*, $B \sim C$, if $\mathsf{V}(B) = \mathsf{V}(C)$.*

Clearly, the relation $\sim$ defined on $\mathcal{B}$ is an equivalence relation. The equivalence class containing a basis B will be denoted by $[B]$.

Now we are going to define an ordering of the bases, and we derive from this an ordering of the equivalence classes as well as the notion of acyclicity in violator spaces.

Definition 9. *For $F, G \subseteq H$ in a violator space (H, V), we say that $F \leq_0 G$ (F is* locally smaller *than G) if $F \cap \mathsf{V}(G) = \emptyset$.*

For equivalence classes $[B], [C] \in \mathcal{B}/\!\sim$, we say that $[B] \leq_0 [C]$ if there exist $B' \in [B]$ and $C' \in [C]$ such that $B' \leq_0 C'$.

We define the relation $\leq_1$ on the equivalence classes as the transitive closure of $\leq_0$. The relation $\leq_1$ is clearly reflexive and transitive. If it is antisymmetric, we say that the violator space is acyclic*, and we define the relation $\leq$ as an arbitrary linear extension of $\leq_1$.*

The intuition of the *locally-smaller* notion comes from LP-type problems: if no element of F violates G, then $G \cup F$ has the same value as G, and monotonicity yields that value-wise, F is smaller than or equal to G.

Note that in the definition of $[B] \leq_0 [C]$ we do not require $B' \leq_0 C'$ to hold for *every* B' and C'.

To show that acyclicity does not always hold, we give an example of a cyclic violator space where $H = \{f, g, h\}$, and V is given by the following table:

G	$\emptyset$	f	g	h	f, g	f, h	g, h	f, g, h
$\mathsf{V}(G)$	f, g, h	h	f	g	h	g	f	$\emptyset$

We can easily check both consistency and locality. The bases are $\emptyset$, one-element sets, and H. We have $\{f\} \leq_0 \{h\} \leq_0 \{g\} \leq_0 \{f\}$, but none of the one-element bases are equivalent; i.e., $\leq_1$ is not antisymmetric.

Abstract LP-type Problems yield Acyclic Violator Spaces. The following proposition is not immediate and we refer the reader to [1] for a proof of it.

Proposition 1. *Consider an abstract LP-type problem $(H, w, W, \leq)$, and let V be its violator mapping. Then (H, V) is an acyclic violator space. Moreover, (H, V) is basis-equivalent to $(H, w, W, \leq)$.*

Acyclic Violator Spaces yield Concrete LP-type Problems. The following proposition is the last ingredient for Theorem 1.

Proposition 2. *Every acyclic violator space* (H, V) *can be represented as a concrete LP-type problem that is basis-equivalent to* (H, V).

Proof. We are given an acyclic violator space (H, V) and we define the mapping $S\colon H \to 2^{\mathcal{B}/\sim}$ that will act as a "concretization" of the constraints in H:

$$S(h) = \{[B]\colon B \in \mathcal{B},\, h \notin \mathsf{V}(B)\} \ .$$

Further, let $\mathcal{H}$ be the image of the mapping S taken as a multiset, i.e.,

$$\mathcal{H} = \{S(h)\colon h \in H\} \ .$$

Thus, S is a bijection between H and $\mathcal{H}$. By saying that a mapping S is a bijection between a set and a multiset we mean that for any $\bar{h} \in \mathcal{H}$, the number of $h \in H$ that map to $\bar{h}$ is equal to the multiplicity of $\bar{h}$. Note that we cannot use some common properties of set bijections; for instance we have to avoid using the inverse mapping S^{-1}.

Additionally, let σ be the induced bijection of 2^H and $2^{\mathcal{H}}$ defined by $\sigma(G) = \{S(h)\colon h \in G\}$, for $G \subseteq H$.

Consider the triple $(\mathcal{B}/\sim, \leq, \mathcal{H})$, where $\leq$ is an arbitrary linear extension of $\leq_1$ (such an extension exists since (H, V) is acyclic and $\leq_1$ therefore antisymmetric). This is a concrete LP-type problem: The only thing to check is the existence of a minimal element of every nonempty intersection $\bigcap \mathcal{G}$ $(\mathcal{G} \subseteq \mathcal{H})$, which is guaranteed by the linearity of $\leq$ (remember from Definition 3 that $\bigcap \mathcal{G} := \bigcap_{G \in \mathcal{G}} G$).

It remains to prove basis-equivalence, which is done in the full paper [1]. □

Propositions 1 and 2, together with the fact that every concrete LP-type problem can be transformed into an abstract one (as described below Definition 4), yield Theorem 1.

Example. Here we present an abstract LP-type problem and we demonstrate the construction (via acyclic violator spaces) of its concrete representation.

For better intuition we actually *start* with a concrete LP-type problem, namely, the following linear programming problem in the positive orthant (rotated by 45 degrees for convenience). Beside the restriction to the positive orthant, the constraints are the four halfplanes depicted in Figure 1. The optimization direction is given by the arrow.

Here the violator space bases are $\emptyset$, a, b, c, d, ac, ad, bc, bd; the equivalence classes are $O = \emptyset$, $A = \{a\}$, $B = \{b\}$, $C = \{c\}$, $D = \{d\}$ and $Q = \{ac\} \sim \{ad\} \sim \{bc\} \sim \{bd\}$. Note that the equivalence classes correspond to the points in the plane. We have $O \leq_1 B \leq_1 A \leq_1 Q$ and $O \leq_1 C \leq_1 D \leq_1 Q$; we choose $\leq$ to be $O < B < A < C < D < Q$. The concrete representation is

h	a	b	c	d
$S(h)$	A, Q	A, B, Q	C, D, Q	D, Q

.

Here we may interpret $S(a)$ as the set of all "canonical" points lying in the halfplane a.

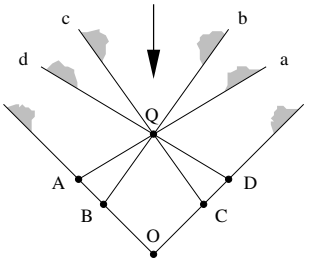


Fig. 1. Illustration example—linear programming

4 Clarkson's Algorithms

We show that Clarkson's randomized reduction scheme, originally developed for linear programs with many constraints and few variables, actually works for general (possibly cyclic) violator spaces. The two algorithms of Clarkson involved in the reduction have been analyzed for LP and LP-type problems before [9, 10, 11]. Our new contribution is that the combinatorial properties underlying Clarkson's algorithms also hold for violator spaces. We omit the analysis of Clarkson's reduction schemes (it can be found in the full paper [1]), since it is almost identical to the analysis in [9, 10, 11].

In this section we will view a violator space as an "LP-type problem without the order", and it turns out that the order is irrelevant for Clarkson's algorithms. Even without an order, we can talk about monotonicity in violator spaces (see [1] for a proof of the following lemma):

Lemma 1. *Any violator space* (H, V) *satisfies*
Monotonicity: $\mathsf{V}(F) = \mathsf{V}(G)$ *implies* $\mathsf{V}(E) = \mathsf{V}(F) = \mathsf{V}(G)$*, for all sets* $F \subseteq E \subseteq G \subseteq H$*.*

Observation 1. *Let* (H, V) *be a violator space. For* $G \subseteq H$ *and all* $h \in H$*, we have*

(i) $\mathsf{V}(G) \neq \mathsf{V}(G \cup \{h\})$ *if and only if* $h \in \mathsf{V}(G)$*, and*
(ii) $\mathsf{V}(G) \neq \mathsf{V}(G \setminus \{h\})$ *if and only if* h *is contained in every basis of* G*.*

An element h *such that (ii) holds is called* extreme *in* G*.*

We are particularly interested in violator spaces with small bases.

Definition 10. *Let* (H, V) *be a violator space. The size of a largest basis is called the* combinatorial dimension $\delta = \delta(H, \mathsf{V})$ *of* (H, V)*.*

Observation 1 implies that in a violator space of combinatorial dimension δ, every set has at most δ extreme elements. This in turn yields the following bound for the *expected* number of violators of a random subset of constraints, using the *sampling lemma* from [13]:

Corollary 1. *Let (H, V) be a violator space of combinatorial dimension δ and $G \subseteq H$ some fixed set. Let v_r be the expected number of violators of the set $G \cup R$, where $R \subseteq H$ is a random subset of size $r < n = |H|$. Then*

$$v_r \leq \delta \frac{n-r}{r+1} \ .$$

Given a violator space (H, V) of combinatorial dimension δ, the goal is to find a basis of H. For this, we assume availability of the following primitive.

Primitive 1. *Given $G \subseteq H$ and $h \in H \setminus G$, decide whether $h \in \mathsf{V}(G)$.*

Given this primitive, the problem can be solved in a brute-force manner by going through all sets of size $\leq \delta$, testing each of them for being a basis of H. It is easily seen that the number of times the primitive needs to be invoked is bounded by $O(n^{\delta+1})$. This can be substantially improved by using Clarkson's algorithms and slightly adapting the analysis in [9, 10, 11] with the help of Corollary 1. We then obtain (see [1]):

Theorem 2. *A basis of H in a violator space (H, V) can be found calling Primitive 1 expected $O\left(\delta n + \delta^{O(\delta)}\right)$ many times.*

5 Grid USO as Models for Violator Spaces

We show in this section that the problem of finding the sink in a δ-dimensional *grid unique sink orientation* [17] can be reduced to the problem of finding the (unique) basis of a violator space of combinatorial dimension δ.

Unique sink orientations of grids arise from various problems, including linear programming over products of simplices and generalized linear complementarity problems (GLCP) over P-matrices [17]. The GLCP has been introduced by Cottle and Dantzig [31] as a generalization of the well known LCP [32]. There are also applications in game theory; for instance [26, 27, 28] show how parity, mean-payoff, and simple stochastic games are related to grid USO.

Grid USO. Fix a partition $\Pi = (\Pi_1, \ldots, \Pi_\delta)$ of the set $H := \{1, \ldots, n\}$ into δ nonempty subsets, where we refer to Π_i as the *block* i. A subset $J \subseteq H$ is called a *vertex* if $|J \cap \Pi_i| = 1$ for all i. The vertices naturally correspond to the Cartesian product of the Π_i. Let $\mathcal{V}$ be the set of all vertices.

In the following definition, we introduce the *grid spanned* by subsets Π_i' whose union is $G \subseteq H$. The vertex set of this grid contains all vertices $J \subseteq G$ ($J \in \mathcal{V}$), with two vertices being adjacent whenever they differ in exactly two elements.

Definition 11. *The δ-dimensional* grid *spanned by $G \subseteq H$ is the undirected graph $\mathcal{G}(G) = (\mathcal{V}(G), \mathcal{E}(G))$, with $\mathcal{V}(G) := \{J \in \mathcal{V} \colon J \subseteq G\}$ and $\mathcal{E}(G) := \{\{J, J'\} \subseteq \mathcal{V}(G) \colon |J \oplus J'| = 2\}$, where $\oplus$ denotes the symmetric difference of sets.*

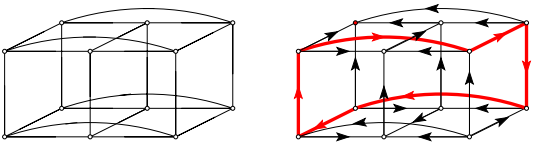


Fig. 2. A 3-dimensional grid $\mathcal{G}(H)$ with $H = \{1, \ldots, 7\}$ where $\Pi = (\{1,2,3\},\{4,5\},\{6,7\})$ and a USO of it

$\mathcal{V}(G)$ is in one-to-one correspondence with the Cartesian product $\prod_{i=1}^{\delta} G_i$, $G_i := G \cap \Pi_i$, and the edges in $\mathcal{E}(G)$ connect vertices in $\mathcal{V}(G)$ whose corresponding tuples differ in exactly one coordinate. See Figure 2 left for an example grid.

Note that $\mathcal{G}(G)$ is the empty graph whenever $G_i = G \cap \Pi_i = \emptyset$ for some i. We say that such a G is *not Π-valid*, and it is *Π-valid* otherwise.

A *subgrid* of $\mathcal{G}(G)$ is any graph of the form $\mathcal{G}(G')$, for $G' \subseteq G$.

Definition 12. *An orientation ψ of the graph $\mathcal{G} := \mathcal{G}(H)$ is called a* unique sink orientation *(USO) if all nonempty subgrids of $\mathcal{G}$ have unique sinks w.r.t. ψ.*

Note that a USO ψ can be cyclic (see the thick edges in Figure 2 right). If ψ induces the directed edge (J, J'), we also write $J \xrightarrow{\psi} J'$. Any USO can be specified by associating each vertex J with its outgoing edges. Given J and $j \in H \setminus J$, we define $J \rhd j$ to be the unique vertex $J' \subseteq J \cup \{j\}$ that is different from J, and we call J' the *neighbor* of J *in direction* j.

Definition 13. *Given an orientation ψ of $\mathcal{G}$, the function $s_\psi : \mathcal{V} \to 2^H$ is called the* outmap *of ψ and is defined by*

$$s_\psi(J) := \{j \in H \setminus J \colon J \xrightarrow{\psi} J \rhd j\} \ . \tag{1}$$

Reduction to Violator Spaces. Let us fix a unique sink orientation ψ of $\mathcal{G}$. Given a Π-valid subset $G \subseteq H$, we define $\mathrm{sink}(G) \in \mathcal{V}(G)$ to be the unique sink vertex in $\mathcal{G}(G)$. For a subset G that is not Π-valid, let $\bar{G} := \bigcup_{i\colon G_i = \emptyset} \Pi_i$. Thus $\bar{G}$ is the set of elements occurring in blocks of Π disjoint from G.

Definition 14. *For $G \subseteq H$, define*

$$\mathsf{V}(G) = \begin{cases} s_\psi(\mathrm{sink}(G)) & \text{if } G \text{ is } \Pi\text{-valid}, \\ \bar{G} & \text{if } G \text{ is not } \Pi\text{-valid} \ . \end{cases}$$

Theorem 3. *The pair (H, V) from Definition 14 is a violator space of combinatorial dimension δ. Moreover, for all Π-valid $G \subseteq H$, the unique sink of the subgrid $\mathcal{G}(G)$ corresponds to the unique basis of G in (H, V).*

The proof is not difficult: consistency is immediate, and as for locality, we distinguish several cases for sets $F \subseteq G \subseteq H$, depending on whether G is Π-valid and whether F is Π-valid. We omit further details here (see [1]).

Note that a violator space obtained from a cyclic USO is also cyclic and that the global sink of the grid USO corresponds to the unique δ-element (and Π-valid) set B with $\mathsf{V}(B) = \emptyset$. This is exactly the set output by Clarkson's algorithms, when we apply it to the violator space constructed in Definition 14. Primitive 1 corresponds to one *edge evaluation* in the USO setting. With Theorem 2, we therefore have:

Theorem 4. *The sink of a unique sink grid orientation can be found by evaluating expected* $O\left(\delta n + \delta^{O(\delta)}\right)$ *edges.*

For small δ, this is faster than the *Product Algorithm* [17] which needs expected $O(\delta! n + H_n^{\delta})$ edge evaluations, where H_n is the n-th harmonic number.

References

1. Gärtner, B., Matoušek, J., Rüst, L., Škovroň, P.: Violator spaces: Structure and algorithms. arXiv.org e-Print archive (2006)
2. Sharir, M., Welzl, E.: A combinatorial bound for linear programming and related problems. In: Proc. 9th Symposium on Theoretical Aspects of Computer Science (STACS). Volume 577 of Lecture Notes in Computer Science., Springer-Verlag (1992) 569–579
3. Matoušek, J., Sharir, M., Welzl, E.: A subexponential bound for linear programming. Algorithmica **16** (1996) 498–516
4. Kalai, G.: A subexponential randomized simplex algorithm. In: Proc. 24th Annual ACM Symposium on Theory of Computing (STOC). (1992) 475–482
5. Amenta, N.: Bounded boxes, Hausdorff distance, and a new proof of an interesting Helly-type theorem. In: Proc. 10th Annual Symposium on Computational Geometry (SCG), ACM Press (1994) 340–347
6. Amenta, N.: Helly theorems and generalized linear programming. Discrete and Computational Geometry **12** (1994) 241–261
7. Björklund, H., Sandberg, S., Vorobyov, S.: A discrete subexponential algorithm for parity games. In: Proc. 20th Annual Symposium on Theoretical Aspects of Computer Science (STACS), Springer-Verlag (2003) 663–674
8. Halman, N.: Discrete and Lexicographic Helly Theorems and Their Relations to LP-type problems. PhD thesis, Tel-Aviv University (2004)
9. Clarkson, K.L.: Las Vegas algorithms for linear and integer programming. Journal of the ACM **42** (1995) 488–499
10. Gärtner, B., Welzl, E.: Linear programming - randomization and abstract frameworks. In: Proc. 13th Annual Symposium on Theoretical Aspects of Computer Science (STACS), London, UK, Springer-Verlag (1996) 669–687
11. Chazelle, B., Matoušek, J.: On linear-time deterministic algorithms for optimization problems in fixed dimension. Journal of Algorithms **21** (1996) 579–597
12. Matoušek, J.: On geometric optimization with few violated constraints. Discrete and Computational Geometry **14** (1995) 365–384
13. Gärtner, B., Welzl, E.: A simple sampling lemma - analysis and applications in geometric optimization. Discrete and Computational Geometry **25**(4) (2001) 569–590
14. Chan, T.: An optimal randomized algorithm for maximum Tukey depth. In: Proc. 15th ACM-SIAM Symposium on Discrete Algorithms (SODA). (2004) 423–429

15. Amenta, N.: A short proof of an interesting Helly-type theorem. Discrete and Computational Geometry **15** (1996) 423–427
16. Szabó, T., Welzl, E.: Unique sink orientations of cubes. In: Proc. 42nd IEEE Symposium on Foundations of Computer Science (FOCS). (2000) 547–555
17. Gärtner, B., Morris, Jr., W.D., Rüst, L.: Unique sink orientations of grids. In: Proc. 11th Conference on Integer Programming and Combinatorial Optimization (IPCO). Volume 3509 of Lecture Notes in Computer Science., Springer-Verlag (2005) 210–224
18. Morris, Jr., W.D.: Randomized principal pivot algorithms for P-matrix linear complementarity problems. Mathematical Programming, Series A **92** (2002) 285–296
19. Matoušek, J.: The number of unique sink orientations of the hypercube. Combinatorica, to appear (2006)
20. Morris, Jr., W.D.: Distinguishing cube orientations arising from linear programs. Manuscript (2002)
21. Develin, M.: LP-orientations of cubes and crosspolytopes. Advances in Geometry **4** (2004) 459–468
22. Matoušek, J., Szabó, T.: Random Edge can be exponential on abstract cubes. In: Proc. 45th Annual IEEE Symposium on Foundations of Computer Science (FOCS). (2004) 92–100
23. Schurr, I., Szabó, T.: Finding the sink takes some time. Discrete and Computational Geometry **31** (2004) 627–642
24. Schurr, I., Szabó, T.: Jumping doesn't help in abstract cubes. In: Proc. 11th Conference on Integer Programming and Combinatorial Optimization (IPCO). Volume 3509 of Lecture Notes in Computer Science., Springer-Verlag (2005) 225–235
25. Gärtner, B., Schurr, I.: Linear programming and unique sink orientations. In: Proc. 17th Annual Symposium on Discrete Algorithms (SODA). (2006) 749–757
26. Björklund, H., Vorobyov, S.: Combinatorial structure and randomized subexponential algorithms for infinite games. Theoretical Computer Science (in press) (2005)
27. Björklund, H., Sandberg, S., Vorobyov, S.: A combinatorial strongly subexponential strategy improvement algorithm for mean payoff games. In: Proc. 29th International Symposium on Mathematical Foundations of Computer Science (MFCS). Volume 3153 of Lecture Notes in Computer Science., Springer-Verlag (2004) 673–685
28. Gärtner, B., Rüst, L.: Simple stochastic games and P-matrix generalized linear complementarity problems. In: Proc. 15th International Symposium on Fundamentals of Computation Theory (FCT). Volume 3623 of Lecture Notes in Computer Science., Springer-Verlag (2005) 209–220
29. Megiddo, N.: A note on the complexity of P-matrix LCP and computing an equilibrium. Technical report, IBM Almaden Research Center, San Jose (1988)
30. Škovroň, P.: Generalized linear programming. Master's thesis, Charles University, Prague, Faculty of Mathematics and Physics (2002)
31. Cottle, R.W., Dantzig, G.B.: A generalization of the linear complementarity problem. Journal on Combinatorial Theory **8** (1970) 79–90
32. Cottle, R.W., Pang, J., Stone, R.E.: The Linear Complementarity Problem. Academic Press (1992)

Region-Restricted Clustering for Geographic Data Mining

Joachim Gudmundsson[1,⋆], Marc van Kreveld[2,⋆⋆], and Giri Narasimhan[3]

[1] National ICT Australia Ltd., Sydney, Australia
[2] Dept. of Computer Science, Utrecht University, The Netherlands
[3] Florida International University, Miami, USA

Abstract. Cluster detection for a set P of n points in geographic situations is usually dependent on land cover or another thematic map layer. This occurs for instance if the points of P can only occur in one land cover type. We extend the definition of clusters to *region-restricted clusters*, and give efficient algorithms for exact computation and approximation. The algorithm determines all axis-parallel squares with exactly m out of n points inside, size at most some prespepcified value, and area of a given land cover type at most another prespecified value. The exact algorithm runs in $O(nm \log^2 n + (nm + nn_f) \log^2 n_f)$ time, where n_f is the number of edges that bound the regions with the given land cover type. The approximation algorithm allows the square to be a factor $1+\varepsilon$ too large, and runs in $O(n \log n + n/\varepsilon^2 + n_f \log^2 n_f + (n \log^2 n_f)/(m\varepsilon^2))$ time. We also show how to compute largest clusters and outliers.

1 Introduction

Spatial data mining is concerned with the detection of interesting patterns from large spatial data sets [12, 17, 19]. For instance, if only one data set is considered, patterns may be related to the concepts of clusters, regularities, or outliers. If more than one data set is considered, patterns of interest may be related to co-locations in space. Objects with many scalar attributes can also be seen as a spatial data set by using the attribute values as coordinates.

In contrast, geographic data mining is a type of spatial data mining where objects or features occupy the geographic space (in the literature, the distinction between spatial and geographical data mining is often not made) [21]. It is a form of geographical analysis: the study into the explanations of geographical phenomena. Geographical analysis includes statistical analysis of geographic data, trend analysis (which includes time), and location planning (which involves combining different data themes) as well.

Clustering has been widely studied in data analysis. Several books and surveys [14, 15, 16] have appeared on the topic, and any book on data mining discusses

⋆ Funded by the Australian Government's Backing Australia's Ability initiative, in part through the Australian Research Council.
⋆⋆ Supported by the Netherlands Organisation for Scientific Research (NWO) through the BRICKS project GADGET.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 399–410, 2006.

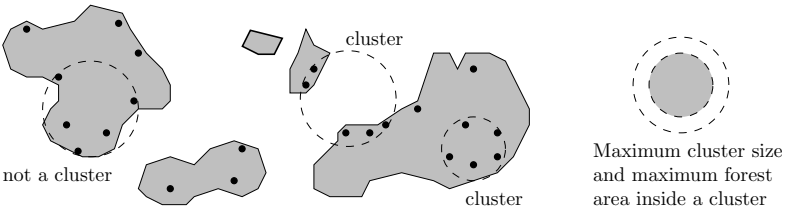


Fig. 1. Clustering when five nests are required to form a cluster

clustering. Clustering can be hierarchical or partitional, the number of clusters may be specified beforehand or not, and many different clustering methods exist, each with their properties. In the algorithms field, clustering is still a very active area of research; major conferences have papers that discuss clustering nearly every year. In this paper we study algorithms for clustering in geographic data mining. The objects to be clustered occupy a geographic space, and that space has other relevant aspects as well. We give two examples.

1. Consider a set of points representing bird nests, and imagine a biologist that is interested to what degree the birds of that species seek each other's company when nesting. If the birds always nests in trees, then the only locations where they can have their nests is where the trees are. If the birds are sea birds that nest on islands, then the water in between cannot contain nests. Clustering of bird nest locations should take this into account to decide if a group of nests is a cluster or not.
2. Consider a set of points representing burglary locations in a city. A cluster of such locations is a group of points that are near to each other. However, if the points occur around the perimeter of a park, then we would like to see this as a cluster as well, because there cannot be buglary locations inside the park. Similarly, car break-ins can only occur where cars may be parked.

We abstract the above situations in a simple way. Future research should extend these abstractions to more realistic versions, an issue we discuss in the conclusions section of this paper. Let P be a set of points in the plane, representing the objects that are analyzed for clustering. Assume further that a subdivision into two land cover types A and B is given, and the points of P only occur in the one land cover type, say, B. A cluster is a subset $P' \subseteq P$ with the following properties: (i) P' should be large enough. (ii) The region occupied by P' should not be too large. (iii) The region occupied by P' that is of type B should not be too large. See Figure 1 for an example. We will model (i) by an integer value m, denoting the minimum size of a subset to be called a cluster. We will model the region occupied by P' by a circle or square that contains P'; only smallest circles and squares are of interest. Properties (ii) and (iii) can now be specified by an area value; the value for (ii) should of course be larger than the value for (iii). We give a more formal definition in the next section. Note that the combination of properties (i) and (iii) specifies a lower bound on the density of points from P' in the region where they can be.

According to several sources, clusters are *partitions* of a set of objects. Other sources also use the term cluster for large enough *subsets* of points that are close. For lack of a better term we will also use the term cluster in this paper. In GIS, finding properties of point sets is known as point pattern analysis [20]. Our definition of region-restricted clusters gives a density-based measure for point pattern analysis.

The squares that we find can be used to define larger clusters and of different shapes, by taking the connected components of the union of the squares. Points are then clustered by these connected components. With our definition of a cluster, we can also define outliers. Any point $p \in P$ that does not occur in any subset P' of P with the three properties listed above is an outlier.

Within the research area of (spatial) data mining, one of the most problematic issues is that there are many more potentially interesting patterns than actually interesting patterns. The importance of our cluster definition is linked to this. Assume a clustering algorithm does not have property (iii), which is the case for all existing clustering algorithms. If the algorithm must be able to find clusters like burglary locations around a park as well, then the cluster region size must be chosen large enough. However, then many clusters will be found that do not include any park, and after human inspection do not appear to be real clusters. Hence, many non-interesting clusters are generated. Our definition overcomes this problem by separating the extent of the cluster from the density of locations in the relevant areas. Clusters of burglary locations around parks are detected without detecting many false clusters in neighborhoods where there are no parks. The same is true for bird nests in trees.

In Section 2 we formally state the definition of a *region-restricted cluster*, in Section 3 we present the algorithm that finds such clusters, and in Section 4 we give an approximation algorithm. As a result of independent interest we develop a data structure that stores a set of polygons with n_f edges in total and has size $O(n_f \log^2 n_f)$, which can answer area containment queries: for any axis-parallel query rectangle, the total area of the polygons inside it can be reported in $O(\log^2 n_f)$ time. Some additional results and details have been omitted from this version, they can be found in the full version [11].

2 Problem Definition

Given a set of disjoint polygonal regions, a distance of interest r, a subset size of interest m, and a set P of points, a cluster is a subset of P of size at least m for which an enclosing circle exists of radius at most $2r$, such that the intersection of this circle and the polygonal regions has total area at most πr^2.

Let us examine the area of intersection of a circle and a single polygon. It is the sum of square roots, in total linearly many in the number of edges of the polygon that intersect the circle. If we allow the circle to translate slightly, and express its location by its center (x, y) then the area of intersection becomes a function in x and y, whose terms appear in the square roots. To find the position of the circle that minimizes the area of intersection, while intersecting the same set of

polygon edges, requires analytical operations that cannot be executed exactly in any reasonable model of computation. In particular, high-degree polynomials must be solved. If the circle is replaced by an axis-parallel square, the situation is quite different. The function giving the area expressed in the center (x, y) of the square is quadratic, so it can have only a constant number of terms regardless of how many polygon edges intersect the square. It can be evaluated and minimized easily in constant time. To avoid the algebraic issues involved with circles, we define clusters with respect to squares in this paper. It allows us to concentrate on the combinatorial aspects of the problem.

Definition 1. *Given a set of disjoint polygonal regions, a distance of interest s, a subset size of interest m, and a set P of points, a* region-restricted cluster *is a subset of P of size at least m for which an enclosing axis-parallel square exists of side length at most $2s$, such that the intersection of this square and the polygonal regions has total area at most s^2.*

For any region-restricted cluster with more than m points, a subset of m points exists that also is a cluster. To avoid redundant information, our algorithm will only find clusters with exactly m points.

3 An Exact Algorithm for Region-Restricted Clusters

Following the analogy of bird nests in trees in forests, we will call the polygonal regions *forests* from now on. The algorithm consists of the following steps. First, for every point $p \in P$, we find the smallest square that has p in the lower left corner and contains exactly m points of P. Second, we trace the collection of all squares that have p on the left side, contain exactly m points, and are smallest. Third, the trace gives us a collection of squares for which we must test whether the side length is at most $2s$ and the area of forest inside is at most s^2. The former test is easy, the latter test is done with a data structure that returns the area of forest inside any query square (or rectangle, for that matter) efficiently. These three steps are described in detail in the next three subsections.

As just mentioned, in Sections 3.1 and 3.2 we show how to find all $O(nm)$ squares that contain exactly m points. This is closely related to the order-m Voronoi diagram of the points in the L_∞-metric. For the L_2-metric, the fastest algorithm takes $O(n \log^3 n + nm \log n)$ expected time and $O(nm)$ space [1]. An algorithm of Eppstein and Erickson [10] determines the $O(m)$ nearest points to each point in the L_∞-metric in $O(n \log n + nm)$ time and $O(n \log n)$ space, but requires a RAM model and bit manipulation. It is not clear whether these results can be used to compute the order-m Voronoi diagram in the L_∞-metric. Various other results exist on determining the smallest square or circle that encloses m points [3, 8, 13]. Most of these approaches, however, do not imply the computation of the squares that we require within the same time bound. Furthermore, approaches that compute the order-m Voronoi diagram [1, 3] require $\Theta(nm)$ space and are complicated. We present a simple algorithm that runs in $O(nm \log^2 n)$ time and requires $O(n \log n)$ space.

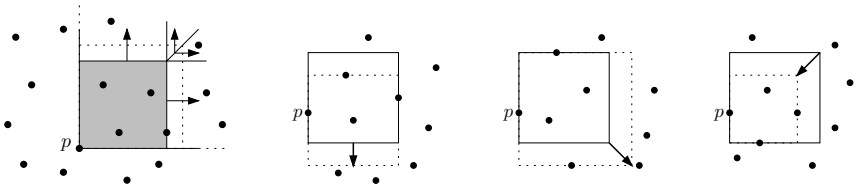


Fig. 2. Left, finding the next point inside a growing square by four queries. Right, situations 1, 2, and 3, where the next situations will be 2, 3, and 1, respectively.

3.1 Initializing for the Sweep

We describe how to determine, for each point $p \in P$, the smallest square that has p in the lower left corner and contains exactly m points of P in $O(nm \log^2 n)$ time. The problem is easy to solve in $O(n^2)$ time: for each point p, find the $(m-1)$-th nearest point in the upper right quadrant of p with respect to the L_∞-metric, using a linear time selection algorithm [7].

Our solution is based on range query data structures. For each point $p \in P$, we grow a square whose lower left corner is fixed at p, and detect the next point of P that will be inside. After $m-1$ steps, we have the desired square for p. The next point to be inside is determined by four queries, see Figure 2. One query finds the first point reached when the top side of the square is translated vertically upwards, a second query finds the first point reached when the right side of the square is translated horizontally to the right, and the third and fourth queries find the lowest and leftmost points in wedges, each bounded by two lines through the upper right corner of the square. The lowest point is found in the wedge bounded by a vertical line and a line with slope 1, and the leftmost point is found in the wedge bounded by a horizontal line and the same line with slope 1. One of the four answers gives the next point that will enter the growing square. With the new point inside, we have the next square, and we perform the same four queries again, but now based on the new, larger square.

The first two queries can easily be solved using a standard, two-dimensional orthogonal range tree [9]. If we apply fractional cascading [6], we get a query time of $O(\log n)$ using a data structure of size and preprocessing time $O(n \log n)$.

The third and fourth queries can be answered using a binary search tree on P sorted by $x - y$, so that a line with slope 1 has the points above and left of it in the left part of the tree, and the points to the right and below it in the right part of the tree. Every internal node of the tree is augmented with a priority search tree [9]. This structure allows us to answer the third and fourth queries in $O(\log^2 n)$ time, using a structure of size $O(n \log n)$ and preprocessing time $O(n \log^2 n)$. Fractional cascading cannot be used to improve the query time.

For all points $p \in P$, we will perform $O(nm)$ queries in total, and hence the total query time is $O(nm \log^2 n)$. The preprocessing time is $O(n \log^2 n)$ and the storage requirements are $O(n \log n)$. Depending on the machine model used, slight variations on these bounds are possible, for which we refer to [2].

3.2 Sweeping Squares Along Points

We show how to find all "interesting" squares that have a point $p \in P$ on the left side and contain exactly m points inside. To this end, we sweep, grow and shrink the square while keeping its left side in contact with p. The previous section showed how to find the first interesting square for the sweep.

There are three ways in which the sweep can advance: 1. The square translates vertically downward. 2. The square grows to the bottom right. 3. The square shrinks from the top right. We describe these situations in more detail; see Figure 2.

1. The square translates vertically downward when it is in contact with p on the left side and some other point of P on the right side, and continues until either the top side of the square reaches a point of P, or the bottom side of the square reaches a point of P. In the former case we go to situation 2, and in the latter case we go to situation 3.

   All squares during the translation are interesting, in the sense that they may give rise to a region-restricted cluster.
2. The square grows to the bottom right when it is in contact with p on the left side and some point of P on the top side. We cannot lower the top side, or else the square would contain only $m - 1$ points. So we let it grow to the bottom right, until either the right side or the bottom side reaches a point of P. In the former case we go to situation 1, and in the latter case we go to situation 3.

   When the right or bottom side reaches a point of P, the square contains $m + 1$ points and therefore is not interesting. As soon as we proceed in situation 1 or 3 we lose the $(m + 1)$-th point again. Other squares during the growing are not interesting, since they properly contain a square with m points inside.
3. The square shrinks from the top right when it has p on the left side and some point of P on the bottom side (but no point of P on the top or right side). The shrinking continues until either the top or the right side reaches a point of P. In the former case we go to situation 2, and in the latter case we go to situation 1.

   Only the final square of the shrinking process is interesting, because it is a subsquare of all others with the same m points inside.

To determine the next event in the sweep efficiently, we use the same two types of data structures as for the initialization of the squares. We next analyze how many events occur during the sweep along p. Note that any point that leaves the square through the top side cannot re-enter, and any point that enters the square through the bottom side can do so only once. At the end of situation 2, we always lose a point through the top. At the end of situation 1, we either lose a point through the top or gain a point at the bottom. At the end of situation 3 we lose a point on the bottom side which goes inside the square. Hence, all situations can occur only $O(n)$ times. It follows that for one point $p \in P$, the sweep takes $O(n \log^2 n)$ time, and summed over all points of P this is $O(n^2 \log^2 n)$ time. Preprocessing takes less time, asymptotically. However, we can also show:

Lemma 1. *The running time of all sweeps is* $O(nm \log^2 n)$ *time.*

Proof. The number of subsets of m points in smallest enclosing squares is $O(nm)$, due to the complexity of order-m Voronoi diagrams in the L_∞-metric [3, 18]. Each event gives a new subset, and each event is handled in $O(\log^2 n)$ time. □

We need to do such sweeps for each point $p \in P$ in contact with each side of a square. It is obvious that we can deal with the other sides of squares within the same time bounds.

3.3 A Data Structure for Area Intersection Queries

The previous section shows how to compute a set of $O(nm)$ subsets of m points that are contained in a smallest square. These squares have a fixed size, but have some x-interval or y-interval of possible locations (for example, the interval of y-coordinates for the top side). The sweeps with the points of P in contact with four possible sides of squares give these intervals. We refer to each such interval as an *interval of squares*.

For all intervals of squares, we first test their size. All that have side length at most s give a region-restricted cluster, even if they are completely covered by forest. All that have side length greater than $2s$ cannot give a region-restricted cluster, because the subset of m points is not close enough. For all intervals of squares whose size is between s and $2s$ we must find out how much forest area is inside each possible location of the square to determine if it forms a region-restricted cluster. In this section we only consider vertical intervals; the horizontal case is symmetric.

We first describe a data structure on the forest regions that, for any query rectangle R, can report the total area of forest inside R. If the forest regions have n_f edges, then the data structure has size $O(n_f \log n_f)$ and answers queries in time $O(\log^2 n_f)$. The structure is based on the hereditary segment tree [5].

The area of a polygon with n edges can easily be computed as the sum of the areas of n quadrilaterals with a horizontal bottom side, vertical left and right sides, and a polygon edge as the top side. Assuming the polygon lies above the x-axis and the vertices are listed clockwise, the area of the polygon $(x_1, y_1), \ldots, (x_n, y_n)$ is $(x_1 - x_n) \cdot (y_1 + y_n)/2 + \sum_{i=1}^{n-1} (x_{i+1} - x_i) \cdot (y_i + y_{i+1})/2$. Note that trapezoids of edges that bound the polygon locally to the top give a positive area contribution, whereas edges that bound the polygon locally from below give a negative area contribution.

In our situation we also may assume that all forest polygons lie above the x-axis. With every edge of a forest polygon we associate the area of its trapezoid, which can be positive or negative, depending on whether the edge bounds a forest region from above or below. We use the x-intervals of all forest edges to get one-dimensional intervals that are stored on the main tree, which is the same as the main tree of a normal segment tree. The associated structures, stored with all nodes, are used similar to the hereditary segment tree [5].

In a hereditary segment tree, every node ν (internal or leaf) corresponds to some interval I_ν. Let e be some forest edge with forest locally below it. Then e

is stored *as a short edge* at every node ν for which an endpoint of e lies in I_ν (in the x-projection). Furthermore, edge e is stored *as a long edge* at every node ν for which I_ν is contained in the x-interval of e, but this does not hold for the parent node of ν (necessarily, e is stored as a short edge at this parent). This is the standard approach for hereditary segment trees.

Each node ν has two associated structures, one for the short edges and one for the long edges stored at ν. Node ν represents a vertical strip $I_\nu \times (-\infty, +\infty)$. All long edges stored at ν cross the strip from left to right. All short edges come in chains that either connect the left to the right boundary, or have both ends on the left boundary, or have both ends on the right boundary. The forest can always be on either side of long edges and of chains of short edges. Within one strip, the specification of the side that contains the forest may seem inconsistent, e.g., two adjacent long edges may both say that the forest is below it. The specification is only local to each edge, however, and the seeming inconsistency does not give problems with the design of the associated structures.

Assume we query with a rectangle $R = [x_1, x_2] \times [y_1, y_2]$ to determine the area of forest inside R. We will query separately with the horizontal edges of R to determine the aggregated area of forest below those edges. A subtraction then gives the area inside R. So we need to be able to determine the area of forest in the half-strip vertically below a horizontal edge $[x_1, x_2] \times y_1$. In the tree, we will query the long segments at all nodes on the search paths to x_1 and x_2 (the query segment is short). Furthermore, we will query the short segments at each node ν for which $[x_1, x_2]$ contains I_ν, but this is not the case for the parent of ν (the query segment is long at ν). Details on how to store long segments and short segments, and query with a short and long segment, respectively, are given in the full paper [11].

Theorem 1. *A set of disjoint polygons with n_f edges in total can be stored in a data structure of size $O(n_f \log n_f)$, such that for any axis-parallel query rectangle, the area inside can be computed in $O(\log^2 n_f)$ time. The construction time is $O(n_f \log^2 n_f)$.*

We use this structure to test our set of $O(nm)$ vertical intervals of squares. We perform a query with the topmost position, which gives us the area of forest inside. However, instead of returning the *area* of forest inside, we can also obtain the *quadratic function in y* that gives the forest area inside the square if the set of forest edges intersected by the sides of the square is the same. This function will be valid for a small subinterval at the top of the interval that we are testing. When the square is translated down, the combinatorial structure of the forest edges intersecting it will change, and so will the quadratic function in y giving the forest area inside. This is an event in the sweep of the square through the forest regions.

There are two types of event. A corner of the square may pass an edge of a forest region, and a side of the square may pass a vertex of a forest region. We preprocess the forest regions into two data structures that allow us to detect all events on time, before they occur. One is a vertical ray shooting structure in the forest edges. A standard locus approach combined with planar point location

solves this; the structure has linear size and logarithmic query time. The other is a segment dragging query, which can be solved using orthogonal range trees with fractional cascading once again.

Between any two consecutive events, we consider the quadratic function in y and minimize it, restricted to the relevant subinterval. If the minimum area is at most s^2, we found a region-restricted cluster and report it. Otherwise, we update the quadratic function based on the change of intersected forest edges in $O(1)$ time, and continue the sweep.

For any sweep, there can be $O(n_f)$ events, giving $O(nm \cdot n_f)$ events for all $O(nm)$ sweeps. However, we can show that the number of events during all $O(nm)$ sweeps over the vertical intervals is only $O(n \cdot n_f)$.

Lemma 2. *The $O(nm)$ vertical sweeps of a square have $O(nm + n \cdot n_f)$ events in total (proof in the full paper [11]).*

Theorem 2. *Given a set P of n points in the plane, a set $\mathcal{F}$ of disjoint polygons with n_f edges, a positive integer m, and a positive real s, we can determine all subsets of P of m points for which a smallest enclosing square exists with side length at most $2s$, and total area of polygons from $\mathcal{F}$ inside at most s^2, in $O(nm \log^2 n + (nm + nn_f) \log^2 n_f)$ time and $O(n \log n + n_f \log n_f)$ space. To report $O(nm)$ clusters of m points each explicitly, we need additional $O(nm^2)$ time.*

Proof. Only the space bound still needs to be proved. We simply note that the $O(nm)$ intervals of squares need not be computed all at once. As soon as we generate a candidate square we test it and report or discard it. So we only need the size of the data structures, which is $O(n \log n + n_f \log n_f)$. □

To find outliers, we simply compute the union of the $O(nm)$ squares that give the clusters, and preprocess it for planar point location. Then we query with all points of P. All points that do not lie in the union are outliers. These steps take less time, asymptotically, than the determination of the squares.

Corollary 1. *All region-restricted cluster outliers can be found in $O(nm \log^2 n + (nm + nn_f) \log^2 n_f)$ time.*

4 Approximation for the Square Size

The size of the square, s, is a value that may be user-specified. In any case, the precise value is not crucial, and running the algorithm with a value of s that is, for example, 10% smaller or larger will generally give just as interesting clusters. Therefore, it makes sense to study approximation algorithms, where the size of the square may deviate slightly from what is specified. Approximation allows us to obtain faster running times of the clustering and outlier detection algorithms. At the same time, the number of clusters we find may be significantly smaller than in the exact case. Recall that this is important in spatial data mining.

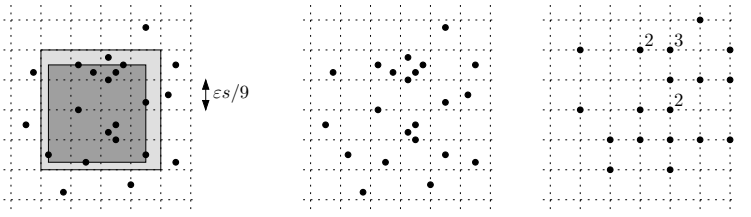


Fig. 3. Left, overlaying a grid on P, and enlargement of squares containing exact clusters (dark grey) to squares containing approximate clusters (light grey). Right, snapping to a grid with multiplicity of grid points.

For a constant $0 < \varepsilon < 1$, the ε-approximate region-restricted cluster reporting problem must determine a set $\mathcal{Q}$ of subsets of P, such that for every region-restricted cluster P' of P, a subset $Q \in \mathcal{Q}$ exists such that $P' \subseteq Q$, the enclosing square S_Q of Q has side length at most $(1+\varepsilon)$ times the side length of the smallest square $S_{P'}$ enclosing P', and area of forest inside S_Q is at most $(1+\varepsilon) \cdot s^2$.

The idea is to overlay a regular grid with spacing $\varepsilon s/9$ over the points of P, snap them to grid points, and only consider squares whose vertices lie on the grid (see Figure 3). If a square S' gives a region-restricted cluster for a subset P', then our approximation algorithm will find the square S'' whose vertices are snapped outwards onto the grid. The side length of S'' is at most that of S', plus $2\varepsilon s/9$, which is within a factor of $(1+\varepsilon)$ of the side length of S'. The area inside S'' that is not in S' is at most $8s \cdot \varepsilon s/9 + 4(\varepsilon s/9)^2 < \frac{76}{81}\varepsilon s^2$, since $\varepsilon < 1$. We observe that the first, second and third conditions will be satisfied with this idea. We may find approximate region-restricted clusters that do not contain any exact region-restricted cluster, but then they would have been an exact cluster for $s' = (1+\varepsilon) \cdot s$. Snapping to a grid for a factor $(1+\varepsilon)$ approximation has been used in several papers before (see e.g. [4, 13]).

Given P, m, s, and ε, we solve the ε-approximate region-restricted cluster reporting problem as follows. Choose a grid with spacing $\varepsilon s/9$ and place the points of P in the appropriate cells. Then we snap the point to the upper right grid vertex. For each snapped point $p \in P$, we select a subgrid of $(1 + \lceil 18/\varepsilon \rceil) \times (1 + \lceil 18/\varepsilon \rceil)$ grid vertices, where the vertex with p is the upper right corner. The snapped coordinates of p are stored with the selected grid vertices. In total, $O(n/\varepsilon^2)$ grid vertices are selected, many of which may be the same. Selected grid vertices are lower left corners of candidate squares that will be tested.

Note that only grid vertices that are chosen with multiplicity $\Omega(m)$ can be part of a square in which an ε-approximate region restricted cluster lies. There are only $O(n/(m\varepsilon^2))$ such grid points, and hence we need to report no more than $O(n/(m\varepsilon^2))$ clusters to solve the ε-approximate region-restricted cluster reporting problem. For each such grid point, we determine the smallest square that has the lower left corner at this grid point, the upper right corner at another

grid point, and contains at least m points. Using a selection algorithm [7] on the snapped points stored with a grid point, this can be done in $O(n/\varepsilon^2)$ time overall.

On the forest edges we build the data structure that was described in the previous section. We query with the $O(n/(m\varepsilon^2))$ squares to determine how much forest is inside. If there is more than allowed, then we discard the square. Otherwise, we report the cluster. We conclude:

Theorem 3. *Given a set P of n points in the plane, a set of disjoint polygons with n_f edges, a positive integer m, a positive real s, and an approximation constant $0 < \varepsilon < 1$, we can solve the ε-approximate region-restricted cluster reporting problem in $O(n \log n + n/\varepsilon^2 + n_f \log^2 n_f + (n \log^2 n_f)/(m\varepsilon^2))$ time.*

To determine the outliers we can again determine the union of the $O(n/(m\varepsilon^2))$ squares, and locate the points of P. Alternatively, we can store P in a semi-dynamic orthogonal range tree, query with each square, and remove the points of P that lie in any query square.

5 Conclusions and Future Research

This paper introduced the concept of region-restricted clustering, which is important for geographic data mining. Our clustering definition takes into account the situation where data points may only be possible in certain regions of the plane. It may help to alleviate the problem of detecting many clusters that are not interesting, by using a more appropriate definition of clusters in geographic situations. We have also given efficient algorithms to compute clusters and outliers according to the new definition. A more efficient approximation algorithm was also presented.

A result of independent interest is a new data structure for a set of disjoint polygons with n edges, such that for any query rectangle, the total polygon area inside it can be determined in $O(\log^2 n)$ time. The data structure has size $O(n \log n)$ and can be built in $O(n \log^2 n)$ time.

As we remarked in the introduction, our definition of clusters is restricted in the sense that the shape of a cluster is fixed and its size must be specified. Common clustering methods like k-means, single link, and complete link [16] do not have this restriction. An important topic for further research is therefore to develop region-restricted versions of these clustering methods.

Open problems include developing more efficient algorithms, and extending to the version where the (forest) area requirement of the square is independent of the side length of the square (in other words: where properties (ii) and (iii) of region-restricted clusters specify two values s_1 and s_2 that are unrelated). Other, more general problems of interest include clustering where the points also have attribute values on which clustering is done, and region-restricted clustering in higher-dimensional geographic spaces.

References

1. P.K. Agarwal, M. de Berg, J. Matoušek, and O. Schwarzkopf. Constructing levels in arrangements and higher order Voronoi diagrams. *SIAM J. Comput.*, 27:654–667, 1998.
2. P.K. Agarwal and J. Erickson. Geometric range searching and its relatives. In B. Chazelle, J.E. Goodman, and R. Pollack, editors, *Advances in Discrete and Computational Geometry*, volume 223 of *Contemporary Mathematics*, pages 1–56. American Mathematical Society, Providence, RI, 1999.
3. A. Aggarwal, H. Imai, N. Katoh, and S. Suri. Finding k points with minimum diameter and related problems. *J. Algorithms*, 12:38–56, 1991.
4. S. Arora. Polynomial time approximation schemes for Euclidean traveling salesman and other geometric problems. *J. ACM*, 45(5):753–782, 1998.
5. B. Chazelle, H. Edelsbrunner, L.J. Guibas, and M. Sharir. Algorithms for bichromatic line segment problems and polyhedral terrains. *Algorithmica*, 11:116–132, 1994.
6. B. Chazelle and L.J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1(3):133–162, 1986.
7. T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 2nd edition, 2001.
8. A. Datta, H.-P. Lenhof, C. Schwarz, and M. Smid. Static and dynamic algorithms for k-point clustering problems. *J. Algorithms*, 19:474–503, 1995.
9. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 2nd edition, 2000.
10. D. Eppstein and J. Erickson. Iterated nearest neighbors and finding minimal polytopes. *Discrete Comput. Geom.*, 11:321–350, 1994.
11. J. Gudmundsson, M. van Kreveld, and G. Narasimhan. Region-restricted clustering for geographic data mining. Technical Report UU-CS-2006-031, Department of Information and Computing Sciences, Utrecht University, 2006.
12. J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Academic Press, San Diego, 2001.
13. S. Har-Peled and S. Mazumdar. Fast algorithms for computing the smallest k-enclosing circle. *Algorithmica*, 41:147–157, 2005.
14. J.A. Hartigan. *Clustering Algorithms*. John Wiley & Sons, New York, 1975.
15. A.K. Jain and R.C. Dubes. *Algorithms for Clustering Data*. Prentice Hall, Englewood Cliffs, NJ, 1988.
16. A.K. Jain, M.N. Murty, and P.J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31:264–323, 1999.
17. K. Koperski, J. Adhikary, and J. Han. Spatial data mining: Progress and challenges. In *Proc. SIGMOD'96 Workshop on Research Issues on Data Mining and Knowledge Discovery*, 1996.
18. D.T. Lee. On k-nearest neighbor Voronoi diagrams in the plane. *IEEE Trans. Comput.*, C-31:478–487, 1982.
19. H.J. Miller and J. Han, editors. *Geographic Data Mining and Knowledge Discovery*. Taylor & Francis, London, 2001.
20. D. O'Sullivan and D.J. Unwin. *Geographic Information Analysis*. John Wiley & Sons, Hoboken, NJ, 2003.
21. J. Roddick, K. Hornsby, and M. Spiliopoulou. An updated bibliography of temporal, spatial, and spatio-temporal data mining research. In *TSDM 2000*, number 2007 in Lect. Notes in Art. Int., pages 147–163, Berlin, 2001. Springer.

An $O(n^3(\log\log n/\log n)^{5/4})$ Time Algorithm for All Pairs Shortest Paths*

Yijie Han

University of Missouri at Kansas City
5100 Rockhill Road
Kansas City, MO 64110, USA
hanyij@umkc.edu
http://www.sce.umkc.edu/~hanyij

Abstract. We present an $O(n^3(\log\log n/\log n)^{5/4})$ time algorithm for all pairs shortest paths. This algorithm improves on the best previous result of $O(n^3/\log n)$ time.

1 Introduction

Given an input directed graph $G = (V, E)$, the all pairs shortest path problem (APSP) is to compute the shortest paths between all pairs of vertices of G assuming that edge costs are real values. The APSP problem is a fundamental problem in computer science and has received considerable attention. Early algorithms such as Floyd's algorithm ([2], pp. 211-212) computes all pairs shortest paths in $O(n^3)$ time, where n is the number of vertices of the graph. Improved results show that all pairs shortest paths can be computed in $O(mn + n^2 \log n)$ time [8], where m is the number of edges of the graph. Recently Pettie showed [12] an algorithm with time complexity of $O(mn + n^2 \log\log n)$. See [13] for recent development. There are also results for all pairs shortest paths for graphs with integer weights[9, 14, 15, 18, 19, 20]. Fredman gave the first subcubic algorithm [7] for all pairs shortest paths. His algorithm runs in $O(n^3(\log\log n/\log n)^{1/3})$ time. Fredman's algorithm can also run in $O(n^{2.5})$ time nonuniformly. Later Takaoka improved the upper bounds for all pairs shortest paths to $O(n^3(\log\log n/\log n)^{1/2})$ [16]. Dobosiewicz [6] gave an upper bound of $O(n^3/(\log n)^{1/2})$ with extended operations such as normalization capability of floating point numbers in $O(1)$ time. In 2004 we obtained an algorithm with time complexity $O(n^3(\log\log n/\log n)^{5/7})$ [10]. Very recently Takaoka obtained an algorithm with time $O(n^3 \log\log n/\log n)$ [17] and Zwick gave an algorithm with time $O(n^3\sqrt{\log\log n}/\log n)$ [21]. Chan gave an algorithm with time complexity of $O(n^3/\log n)$ [5]. Chan's algorithm does not use tabulation and bit-wise parallelism. His algorithm also runs on a pointer machine. We were unaware of Chan's result and published [11] in which we achieved $O(n^3/\log n)$ time using large word size. Since Chan published [5] before us the result of $O(n^3/\log n)$ time should be fully attributed to Chan.

* Research supported in part by NSF grant 0310245.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 411–417, 2006.

Takaoka raise the question [17] whether $O(n^3/\log n)$ can be achieved. It is thought that $O(n^3/\log n)$ is a natural bound for this problem [5]. We show, however, that this bound can be improved to $O(n^3(\log\log n/\log n)^{5/4})$. Our technique is the traditional table lookup technique. However, we construct many tables and maximize the saving the table lookup could bring to us. We shall give reasons why the results presented in this paper would be difficult to improve on.

2 The Approach

Since it is well known that the all pairs shortest paths computation has the same time as computing the distance product of two matrices [1] ($C = AB$), we will concentrate on the computation of distance product.

We divide the first matrix A into $(n/c_1)(\log\log n/\log n)^{1/2} \cdot (n/c_2)(\log\log n/\log n)^{1/4}$ small matrices each has dimension $c_1(\log n/\log\log n)^{1/2} \times c_2(\log n/\log\log n)^{1/4}$, where $0 < c_1, c_2 \leq 1$ is a suitable constant to be chosen later. We divide the second matrix B into $(n/c_2)(\log\log n/\log n)^{1/4} \cdot (n/c_1)(\log\log n/\log n)^{1/2}$ small matrices each has dimension $c_2(\log n/\log\log n)^{1/4} \times c_1(\log n/\log\log n)^{1/2}$.

For a $c_1(\log n/\log\log n)^{1/2} \times c_2(\log n/\log\log n)^{1/4}$ matrix $E = (e_{ij})$, we obtain the ranks for $e_{ir} - e_{is}$ for all $1 \leq r < s \leq c_2(\log n/\log\log n)^{1/4}$. Here rank is defined in [17] and will be given explicitly in the next section. These ranks are $O(\log\log n)$ bits numbers. We therefore obtain a matrix E' of ranks. E' has dimension $e \times f$ where $e = c_1(\log n/\log\log n)^{1/2}$ and $f = O((\log n/\log\log n)^{1/2})$. E' has $c_3 \log n$ bits with $c_3 < 1/2$ being a constant. The corresponding $c_2(\log n/\log\log n)^{1/4} \times c_1(\log n/\log\log n)^{1/2}$ matrix F is processed similarly. Now EF can be computed in one step by a table lookup. Since direct or naive computation of EF needs $O((\log n/\log\log n)^{5/4})$ time we saved a factor of $(\log n/\log\log n)^{5/4}$ in the time complexity. However, the result R has $c_1^2 \log n/\log\log n$ numbers (indices) each having $O(\log\log n)$ bits. We cannot disassemble R immediately for otherwise we lose a factor of $\log n/\log\log n$ in the time complexity.

What we do is to combine the results for $\log n/\log\log n$ small matrix product into one. This should take $O(\log n/\log\log n)$ time by repeated table lookup (details in the later section). After that we disassemble R and get the $c_1^2 \log n/\log\log n$ resulting indices.

This concludes the strategy of our approach.

3 Obtaining the Ranks

We first divide the first matrix A into $n/\log^4 n \cdot n(\log\log n/\log n)^{5/4}$ medium matrices each of dimension $\log^4 n \times (\log n/\log\log n)^{5/4}$ and second matrix B into $n(\log\log n/\log n)^{5/4} \cdot n/\log^4 n$ medium matrices each of dimension $(\log n/\log\log n)^{5/4} \times \log^4 n$. We will further divide each medium matrix into small matrices in the next section. The reason we need these medium matrices is that we need to compute the ranks to be defined below.

Let A_1 and B_1 be two medium matrices. We want to compute $C_1 = A_1B_1$, where $A_1 = (a_{ij})$, $B_1 = (b_{ij})$ and $C_1 = (c_{ij})$. We have that $c_{ij} = \min_k\{a_{ik}+b_{kj}\}$. Fredman noticed [7] that $a_{ir}+b_{rj} \leq a_{is}+b_{sj} \Longleftrightarrow a_{ir}-a_{is} \leq b_{sj}-b_{rj}$. Takaoka[16] showed that by sorting $\{a_{ir} - a_{is}\}_{i=1,2,...,m}$ into sorted list E_{rs} and sorting $\{b_{sj} - b_{rj}\}_{j=1,2,...m}$ into sorted list F_{rs} and then merging E_{rs} and F_{rs} we obtain a sorted list G_{rs}. Here $m = \log^4 n$. Let H_{rs} be the list of ranks of $a_{ir} - a_{is}$ $(i = 1, 2, ..., m)$ in G_{rs} and let L_{rs} be the list of ranks of $b_{sj} - b_{rj}$ $(j = 1, 2, ...m)$ in G_{rs}. Let $H_{rs}[i]$ and $L_{rs}[j]$ be the i-th and the j-th component of H_{rs} and L_{rs}, respectively. Then:

$$G_{rs}[H_{rs}[i]] = a_{ir} - a_{is}, G_{rs}[L_{rs}[j]] = b_{sj} - b_{rj}.$$

The lists H_{rs} and L_{rs} for $1 \leq r < s \leq l$ can be made in $O(l^2m)$ time, when the sorted lists are available. Here $m = \log^4 n$ and $l = (\log n/\log\log n)^{5/4}$. The following fact is established in [16, 17].

$$a_{ir} + b_{rj} \leq a_{is} + b_{sj} \Longleftrightarrow a_{ir} - a_{is} \leq b_{sj} - b_{rj} \Longleftrightarrow H_{rs}[i] \leq L_{rs}[j].$$

Note that each $H_{rs}[i]$ and $L_{rs}[j]$ has $4\log\log n + 1$ bits.

Here we calculate the time for sorting and for merging. Sorting for each pair of rs for each medium matrix takes $O(m\log m)$ time. For all pairs of rs this is $O(ml^2\log m)$ time. For the input matrix A this takes $O((n/m)(n/l)ml^2\log m)$= $O(n^2l\log m)$= $O(n^2(\log n/\log\log n)^{5/4}\log\log n)$ time. The same time holds for matrix B. The merging takes $O((n/m)^2(n/l)l^2m)$= $O(n^3l/m)$ = $O(n^3(\log n/\log\log n)^{5/4}/\log^4 n)$= $O(n^3(\log\log n/\log n)^{5/4})$ time.

The purpose of this section is to obtain all the ranks $H_{rs}[i]$'s and $L_{rs}[j]$'s.

4 Computing the Small Matrix Product

We further divide each medium matrix into small matrices. Divide the first medium matrix A_1 into small matrices each of dimension $c_1(\log n/\log\log n)^{1/2} \times c_2(\log n/\log\log n)^{1/4}$ and divide the second medium matrix B_1 into small matrices each of dimension $c_2(\log n/\log\log n)^{1/4} \times c_1(\log n/\log\log n)^{1/2}$. Let $E = (e_{ij})$ be a small matrix from A_1 and let $F = (f_{ij})$ be a small matrix from B_1. We are to compute $H = EF$.

Thus if we have $H_{rs}[i]$ and $L_{rs}[j]$ for $1 \leq r < s \leq q$ we can determine the index k such that $h_{ij} = e_{ik} + f_{kj}$. Since $q = c_2(\log n/\log\log n)^{1/4}$ we have $O((\log n/\log\log n)^{1/2})$ rs pairs. We form matrix E' and F':

$$E' = \begin{bmatrix} H_{11}[1] & H_{12}[1] & ... & H_{23}[1] & ... & H_{q-1,q}[1] \\ H_{11}[2] & H_{12}[2] & ... & H_{23}[2] & ... & H_{q-1,q}[2] \\ ... & ... & ... & ... & ... & ... \\ H_{11}[p] & H_{12}[p] & ... & H_{23}[p] & ... & H_{q-1,q}[p] \end{bmatrix}$$

$$F' = \begin{bmatrix} L_{11}[1] & L_{12}[1] & ... & L_{23}[1] & ... & L_{q-1,q}[1] \\ L_{11}[2] & L_{12}[2] & ... & L_{23}[2] & ... & L_{q-1,q}[2] \\ ... & ... & ... & ... & ... & ... \\ L_{11}[p] & L_{12}[p] & ... & L_{23}[p] & ... & L_{q-1,q}[p] \end{bmatrix}$$

where $p = c_1(\log n/\log\log n)^{1/2}$ and $q = c_2(\log n/\log\log n)^{1/4}$.

Thus E' and F' have $c_3 \log n / \log\log n$ numbers each having $4\log\log n + 1$ bits. That is E' and F' have $c_4 \log n$ bits. Here c_3 and c_4 are constants which can be made small. Now E' and F' can be used to index into a lookup table T_1 to get the $p^2 = c_1^2 \log n / \log\log n$ indices for the matrix multiplication of EF. These p^2 indices are stored in one word. The lookup table T_1 can be built in $O(n)$ time. Thus small matrix multiplication takes constant time.

5 Combining Results

Each small matrix multiplication returns a word R of $c_5 \log n$ bits. R can be viewed as an $p \times p$ matrix $R[1..p, 1..p]$ with each element taking $(1/4)(\log\log n - \log\log\log n) + \log c_2$ bits to indicate the winner's index. Suppose we have two small matrix multiplication result R_1 and R_2. Let $R_1[i,j] = r_{ij}$ and $R_2[i,j] = s_{ij}$, if we can obtain R_3 with $R_3[i,j] = (H_{r_{ij}s_{ij}}[i], r_{ij}, s_{ij})$, then our computation can continue. Here $H_{r_{ij}s_{Ij}}[i]$ depends on both i and j because for different j's it may have different values (since H_{rs}'s are the ranks resulting from merging various differences in A_1 and differences in B_1, it therefore depends on the choice of B_1). When A_1 and B_1 are fixed $H_{r_{ij}s_{Ij}}[i]$ is fixed also.

We now use R_{u1v} and R_{u2v}, for a fixed u and $0 \le v < \log^4 n/(c_1(\log n/\log\log n)^{1/2})$, to represent the results of matrix multiplication of the two (consecutive) small matrices in A_1 with a row of (a total of $\log^4 n/(c_1(\log n/\log\log n)^{1/2})$) small matrices in B_1. We first group R_{u1v} and R_{u2v} into one matrix R_{uv} by a table lookup such that $R_{u1v}[i,j]$ and $R_{u2v}[i,j]$ are consecutively placed in $R_{uv}[i,j]$. Therefore $R_{uv}[i,j]$ has $(1/2)(\log\log n - \log\log\log n) + 2\log c_2$ bits. We may assume that the address (that is (u,v,i,j)) is stored together with each original number $R_{uv}[i,j]$. Since each address is a $9\log\log n$ bit number this should not create any problem. We shall call (u,v,i,j) the address of $R_{uv}[i,j]$ and the original number in $R_{uv}[i,j]$ the winning index of $R_{uv}[i,j]$.

We build a table for each row of a medium matrix in A and each medium matrix of a row of medium matrices in B. That is we need $(n^2/\log^{5/4} n)(n/\log^4 n) = n^3/\log^{21/4} n$ tables. Each of such a table has size $O(\log^{2*5/4} n) = O(\log^{5/2} n)$ because $R_{uv}[i,j]$ has 2 values each can be as large as $\log^{5/4} n$. For each of $R_{uv}[i,j]$ value we can index into a table to find the H_{rs} value.

We cannot disassemble R_{uv} to get the $R_{uv}[i,j]$ values to index into the tables built above because that will incur extra cost which we cannot afford. What we do is to sort the $R_{uv}[i,j]$'s within each R_{uv} by their $(i, \text{winning index})$ key and move the addresses with the winning indices. This is done by a table lookup (not the table built above). Since there are $c_1(\log n/\log\log n)^{1/2}$ winning indices with the same i address and each winning index has $(1/2)(\log\log n - \log\log\log n) + 2\log c_2$ bits, by a suitable choice of c_1 and c_2 so that $2^{(1/2)(\log\log n - \log\log\log n) + 2\log c_2} \le c_1(\log n/\log\log n)^{1/2}$ and therefore there are more winning indices than the different values of winning indices. Within each R_{uv} for each value of winning indices we just need to keep one copy. We use a table lookup to obtain the H_{rs} value for each different winning index value in R_{uv}. Since there are $c_2^2(\log n/\log\log n)^{1/2}$ different winning index values we

need to take $O((\log n/\log\log n)^{1/2})$ time. This is for a fixed i in $R_{uv}[i,j]$. For all R_{uv}'s for a fixed u we need $O(\log n/\log\log n)$ time. After that we assemble all these obtained H_{rs} values into one word w. This is done for all R_{uv}'s and there are $\log^4 n/(c_1(\log n/\log\log n)^{1/2})$ of them. We make copies of w and concatenate one copy of w with each R_{uv}. That is the way each R_{uv} obtains its H_{rs} values.

The similar process is done on the second input matrix B. If $R_1[i,j] = r_{ij}$ and $R_2[i,j] = s_{ij}$, then we will obtain R_4 with $R_4[i,j] = (L_{r_{ij}s_{ij}}[j], r_{ij}, s_{ij})$.

Now use R_3 and R_4 to index into yet another table T_4. T_4 will give result R_5 with $R_5[i,j]$ being the winner between r_{ij} and s_{ij} for $1 \le i,j \le c_1(\log n/\log\log n)^{1/2}$. R_3 and R_4 have $c_5 \log n$ bits and therefore table T_4 can be built in $O(n)$ time. This accomplishes the combining of R_1 and R_2 into R_5. This is the 0-th step of combining. Each step of the combining pairs-off every 2 small matrices.

In the t-th step, we group every 2^{2t} R_{uv} into one group and work on every group the same way. We need to discuss the first group R_{uv}, $0 \le v < 2^{2t}$. We replace each duplicate winning index in $R_{uv}[i,*]$ with a dummy (can be set to max) but keep the first winning index among the same duplicated winning indices. Let the resulting word be R^1_{uv}. We use each distinct winning index value of $R_{uv}[i.j]$ and index into the lookup table to obtain the H_{rs} values. There are $2^{2t}c_2^2(\log n/\log\log n)^{1/2}$ distinct winning indices for a fixed i. Look them up in the tables takes $O(2^{2t}(\log n/\log\log n)^{1/2})$ time. For all i's in u this takes $O(2^{2t}\log n/\log\log n)$ time. But this is done once for all groups. Since $s^{2t} \le O(\log^{5/2} n)$ and the number of groups is $\Omega(\log^4 n/(2^{2t}(\log n/\log\log n)^{1/2})) = \Omega(\log n(\log\log n)^{1/2}) > \Omega(\log n/\log\log n)$ (the number of winning indices in R_{uv}). Therefore the time needed can be allocated.

We put these H_{rs} values together with its winning indices into 2^{2t} words w_i, $0 \le i < 2^{2t}$. We then sort R^1_{uv}, $0 \le v < 2^{2t}$, together with w_i, $0 \le i < 2^{2t}$, by the packed winning indices in them. We can do this by a serialized version of the bitonic sort[3, 4]. The bitonic sort will bring a factor of $O(t^2)$ to the complexity (note that we are sorting $O(2^{2t})$ words, the winning indices within each word can be sorted by a table lookup.). Since in the t-th step we have reduced the data amount by a factor of 2^t the factor of $O(t^2)$ in the sorting can be absorbed. Therefore we can achieve linear time for sorting.

After sorting we now copy H_{rs} value to attach them to winning indices. For each winning index there are at most 2^{2t} copies since duplicates have been removed. Therefore copying can be done by incuring a factor of $O(t)$ in the time complexity and since data amount have been reduced this extra factor can be absorbed.

Now use bitonic sort to sort on the addresses to bring the H_{rs} values to each R_{uv}.

The combining of each small matrix thus takes constant time. Therefore after $O(\log n/\log\log n)$ steps we have combined $\log n/\log\log n$ small matrix multiplication result into one resulting matrix R_6. R_6 can be viewed as an

$c(\log n/\log\log n)^{1/2} \times c(\log n/\log\log n)^{1/2}$ matrix with $R_6[i,j]$ giving the index k for $c_{ij} = a_{ik} + b_{kj}$ (mentioned in the previous section).

6 The Result

Since we expend $O(\log n/\log\log n)$ time multiplying a $p \times l$ matrix with an $l \times p$ matrix where $p = c(\log n/\log\log n)^{1/2}$ and $l = (\log n/\log\log n)^{5/4}$, and direct or naive matrix multiplication takes $O((\log n/\log\log n)^{9/4})$ time we save a factor of $(\log n/\log\log n)^{5/4}$. By choosing small enough constant c_i's all our preprocessing and table built up can be done in $O(n^3(\log\log n/\log n)^{5/4})$ time. Thus our matrix multiplication algorithm takes $O(n^3(\log\log n/\log n)^{5/4})$ time.

Theorem. All-pairs shortest paths can be computed in $O(n^3(\log\log n/\log n)^{5/4})$ time.

We give reasons why the results presented in this paper would be difficult to improve on. When using tabulation and bit-parallelism we can expect to save a factor of roughly $\log n$ because $\log n$ bits are encoded in one word. However, in all pairs shortest paths computation if we encode $\log n$ numbers into a word then we can compute $\log^2 n$ results in constant time by table lookup. That is we can speed up algorithm by a factor of $\log^2 n$. However, the $\log^2 n$ results have to be stored in at least $\log n$ words and therefore takes $\log n$ time to access. Thus the speedup factor is reduced to $\log n$. Since we need $\log\log n$ bits to encode a number, with $\log n$ bits we can use one word to store $\log n/\log\log n$ results. Thus the best strategy is to encode a $(\log n/\log\log n)^{1/2} \times a$ matrix and an $a \times (\log n/\log\log n)^{1/2}$ matrix into a word. Here the larger the a, the better. However, we can encode at most $\log n/\log\log n$ numbers into one word because each number takes $\log\log n$ bits, plus if we use Fredman-Takaoka approach[7, 16] then the encoding will blow a numbers to a^2 numbers and thus we have that $a^2(\log n/\log\log n)^{1/2} = \log n/\log\log n$. Therefore $a = (\log n/\log\log n)^{1/4}$. This is the dimension of the small matrices we used earlier in the paper.

Acknowledgment

The author wishes to thank the referees for pointing mistakes in the paper, for helpful comments and for bringing several papers to my attention.

References

1. A. V. Aho, J. E. Hopcroft, J. D. Ullman. The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, MA, 1974.
2. A. V. Aho, J. E. Hopcroft, J. D. Ullman. Data Structures and Algorithms, Addison-Wesley, Reading, MA, 1983.
3. S. Albers, T. Hagerup. Improved parallel integer sorting without concurrent writing. *Information and Computation* 136, 25-51(1997).

4. K.E. Batcher. Sorting networks and their applications. *Proc. 1968 AFIPS Spring Joint Summer Computer Conference*, 307-314(1968).
5. T.M. Chan. All-pairs shortest paths with real weights in $O(n^3/\log n)$ time. *Proc. 9th Workshop Algorithms Data Structures, Lecture Notes in Computer Science*, Vol. 3608, Spriner-Verlag, 318-324(2005).
6. W. Dobosiewicz. A more efficient algorithm for min-plus multiplication. *Inter. J. Comput. Math.* **32**, 49-60(1990).
7. M. L. Fredman. New bounds on the complexity of the shortest path problem. *SIAM J. Computing* **5**, 83-89(1976).
8. M. L. Fredman, R. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34, 596-615, 1987.
9. Z. Galil, O. Margalit. All pairs shortest distances for graphs with small integer length edges. *Information and Computation*, 134, 103-139(1997).
10. Y. Han. Improved algorithms for all pairs shortest paths. *Information Processing Letters*, 91, 245-250(2004).
11. Y. Han. Achieving $O(n^3/\log n)$ time for all pairs shortest paths by using a smaller table. *Proc. 21st Int. Conf. on Computers and Their Applications (CATA-2006)*, Seattle, Washington, 36-37(2006).
12. S. Pettie. A faster all-pairs shortest path algorithm for real-weighted sparse graphs. *Proceedings of 29th International Colloquium on Automata, Languages, and Programming (ICALP'02), LNCS Vol. 2380*, 85-97(2002).
13. S. Pettie, V. Ramachandran. A shortest path algorithm for real-weighted undirected graphs. *SIAM J. Comput.*, Vol. 34, No. 6, 1398-1431(2005).
14. P. Sankowski. Shortest paths in matrix multiplication time. *Proceedings of 13th Annual European Symposium on Algorithms*. Lecture Notes in Computer Science 3669, 770-778(2005).
15. R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *J. Comput. Syst. Sci.*, 51, 400-403(1995).
16. T. Takaoka. A new upper bound on the complexity of the all pairs shortest path problem. *Information Processing Letters* **43**, 195-199(1992).
17. T. Takaoka. An $O(n^3\log\log n/\log n)$ time algorithm for the all-pairs shortest path problem. *Information Processing Letters* 96, 155-161(2005).
18. M. Thorup. Undirected single source shortest paths with positive integer weights in linear time. *Journal of ACM*, 46(3), 362-394(1999).
19. R. Yuster, U. Zwick. Answering distance queries in directed graphs using fast matrix multiplication. *46th Annual IEEE Symposium on Foundations of Computer Science . IEEE Comput. Soc. 2005*, pp. 389-96. Los Alamitos, CA, USA.
20. U. Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM*, vol.49, no.3, May 2002, pp. 289-317.
21. U. Zwick. A slightly improved sub-cubic algorithm for the all pairs shortest paths problem. Proceedings of ISAAC 2004, Lecture Notes in Computer Science, Vol. 3341, Springer, Berlin, 921-932(2004).

Cheating by Men in the Gale-Shapley Stable Matching Algorithm

Chien-Chung Huang

Department of Computer Science
Dartmouth College, NH 03755, USA
villars@cs.dartmouth.edu

Abstract. This paper addresses strategies for the stable marriage problem. For the Gale-Shapley algorithm with men proposing, a classical theorem states that it is impossible for every cheating man to get a better partner than the one he gets if everyone is truthful. We study how to circumvent this theorem and incite men to cheat. First we devise coalitions in which a non-empty subset of the liars get better partners and no man is worse off than before. This strategy is limited in that not everyone in the coalition has the incentive to falsify his list. In an attempt to rectify this situation we introduce the element of randomness, but the theorem shows surprising robustness: it is impossible that every liar has a chance to improve the rank of his partner while no one gets hurt. To overcome the problem that some men lack the motivation to lie, we exhibit another randomized lying strategy in which every liar can expect to get a better partner on average, though with a chance of getting a worse one. Finally, we consider a variant scenario: instead of using the Gale-Shapley algorithm, suppose the stable matching is chosen at random. We present a modified form of the coalition strategy ensuring that every man in the coalition has a new probability distribution over partners which majorizes the original one.

1 Introduction

Suppose that n men and n women seek life-long partners. Each of them has a preference list of the members of the other sex and submits it to a centralized authority. In the spirit of making all the participants maintain a long-term relationship, the authority has to make sure that the matching does not involve any *blocking pair*: a couple each of whom prefers the other over his (her) partner in the matching. A matching without any blocking pair is *stable*. The goal of the authority, given the men's and women's preference lists, is to find a stable matching.

The above situation is the classical stable marriage problem formulated by Gale and Shapley [3]. Suppose the match-making mechanism is known beforehand, and all men's and women's preference lists are made public. Can a group of persons (of either sex) falsify their lists to get better partners?

For the Gale-Shapley men-optimal algorithm, some studies have partly answered the question. If women are allowed to submit incomplete lists (i.e., they can declare some men unacceptable), they can force a men-optimal matching

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 418–431, 2006.

into a women-optimal one [4]. For men, researchers reached the opposite conclusion: honesty is the best policy [2, 10]. The following theorem by Dubins and Freedman [2] (Roth also gave a restricted version [10]) inspires this work and is the key to our results.

Theorem 1. *A subset of men cannot falsify their preference lists so that every one of them gets a better partner than in the Gale-Shapley men-optimal algorithm.*

This work studies how to circumvent this theorem and encourages men to falsify their lists. The statement of the theorem does not rule out the possibility that some of the liars get better partners while the others get the same partners as before. Based on this observation, we devise a coalition strategy. Moreover, we prove this is the *only* cheating strategy in which none of the liars is worse off.

The coalition strategy has a drawback: it relies on the cooperation of some men who cannot benefit themselves. We consider that a randomized version of the coalition strategy might give every liar a chance to get a better partner. However, we reach an impossibility result which states that such a randomized strategy is unrealizable, thus in this sense strengthening the Dubins-Freedman Theorem.

Relaxing the requirement that liars can never be worse off, we present a randomized strategy in which every liar can *expect* to get a better partner. Thus, in an amortized sense, our third attempt in circumventing the Dubins-Freedman Theorem does succeed.

Finally, we discuss a different scenario: the stable matching is chosen at random, what would be men's strategy? This question is raised by Roth and Vate [14]. We study how the lattice structure underlying the set of stable matchings evolves with regard to the coalition strategy. A corollary of our observation is a modified coalition strategy guaranteeing that *every* man in the coalition has a probability distribution over partners which majorizes the original one.

The main contribution of this work is the re-examination of the classical Dubins-Freedman Theorem and its associated strategy issues. To our knowledge, ours is the first result about men-lying strategies (deterministic or randomized) under the Gale-Shapley algorithm. We also present the first men's group lying strategy without relying on truncating lists in the context of random stable matching.

The outline of this paper is as follows. In Section 2, we observe the interaction between the preference lists and the men-optimal matching. Section 3 formally presents the coalition strategy. In Section 4, we prove that there always exist some men who do not gain by lying. In Section 5, we exhibit another randomized lying strategy in which men on the average can get better partners. Section 6 considers the scenario that the stable matching is chosen at random and analyzes the effectiveness of the coalition strategy in this context. Section 7 concludes and discusses related work.

2 Falsifying Preference Lists

In this section, we observe the interaction between falsified lists and the resulting matchings. Before plunging into technical details, we establish some notation and

terminology and give background. From Section 3 to 5, we assume that the Gale-Shapley men-optimal algorithm is used and that we know the preferences of all participants. The sets of men and women are denoted by $\mathcal{M}$ and $\mathcal{W}$, both of size n. When everyone is honest, M_0 and M_z are the men-optimal and women-optimal matchings; M_s denotes the men-optimal matching when some subset of people lie. For any matching M and some subset of people $S \subseteq \mathcal{M} \bigcup \mathcal{W}$, the collection of partners of people in S is $M(S)$. For example, $M_0(m)$ is the partner of man m in the men-optimal matching. We express the fact that man m prefers woman w over woman w' by $w \succ_m w'$. For man m, w is his stable partner if there exists any stable matching containing the pair (m, w).

Every man and woman has a strictly ordered preference list of size n (note that our result still holds even if lists are incomplete). Specifically, for man m, his preference list is composed of $(P_L(m), M_0(m), P_R(m))$, where $P_L(m)$ and $P_R(m)$ are respectively those women ranking higher and lower than $M_0(m)$. More colloquially, we say the women in $P_L(m)$(or $P_R(m)$) are on the left (right) of man m's list. If for every man $m \in \mathcal{M}$, $M(m) \succeq_m M'(m)$, matching M is said to be "at least as good as" matching M' and is denoted as $M \succeq M'$. If, besides $M \succeq M'$, there exists at least one man m such that $M(m) \succ_m M'(m)$, we write $M \succ M'$ and say M is strictly better than M'; if some men are better off and some are worse off in M than in M', these two matchings are said to be *incomparable*, denoted by $M \parallel M'$. Finally, if A is a set of distinct objects, $\pi(A)$ denotes the set of all $|A|!$ permutations and $\pi_r(A)$ a random permutation from this set.

The celebrated Gale-Shapley algorithm is recreated below.

```
1:   assign each person to be free;
2:   while some man m is free do
3:   begin
4:       w:= first woman on m's list to whom m has not yet proposed;
5:       if w is free then
6:           assign m and w to be engaged to each other;
7:       else
8:           if w prefers m to her fiance m' then
9:               assign m and w to be engaged and m' to be free;
10:          else
11:              w rejects m;
12   end;
13:  output the matching
```

Fig. 1. Gale-Shapley men-optimal algorithm. The women-optimal version can be derived by reversing the roles of men and women.

Our first lemma hints at the necessary ingredient in men's falsified lists if we wish for a better outcome for men: Men shifting women from the left to the right of their lists will not cause any man to be worse off.

Lemma 1. *For a subset of men $S \subseteq \mathcal{M}$, if every member $m \in S$ submits a falsified list of the form $(\pi_r(P_L(m) - X), M_0(m), \pi_r(P_R(m) \cup X)), X \subseteq P_L(m)$, then $M_s \succeq M_0$.*

Proof. We proceed by contradiction. In M_s, suppose some man m gets a worse partner than $M_0(m)$. Without loss of generality, assume that during the execution of the algorithm with true lists, m is the first person rejected by his M_0-partner. The rejection can only be caused by another man m', who ranks higher than m in $M_0(m)$ preference list. Since m' has not been accepted by his M_0-partner yet, he must prefer $M_0(m)$ over $M_0(m')$. Therefore, $(m', M_0(m'))$ compose a blocking pair in M_0. □

Interestingly, Lemma 1 also has an intuitive interpretation: if some men know beforehand that they have no chance of getting certain women, they may as well avoid proposing to them. Doing this, they do not run any risk of getting worse partners and may help others get better ones.

The next lemma indicates that if men simply permute the left and/or right portion of their lists, nothing will change. This lemma goes a long way toward explaining why Lemma 1 is a useful lying stratagem.

Lemma 2. *For a subset of men $S \subseteq \mathcal{M}$, if every member $m \in S$ submits a falsified list of the form $(\pi_r(P_L(m)), M_0(m), \pi_r(P_R(m)))$, then $M_s = M_0$.*

Proof. We can use the same argument in the proof of Lemma 1 to show that no man will ever be rejected by his M_0-partner. Hence, permuting the right portion of the men's preference lists will not cause men to be worse off. However, the permutation on the left portion of the preference lists might cause some men to be better off in M_s than in M_0. We have to eliminate this possibility.

Suppose there exists a nonempty subset $B \subset \mathcal{M}$ such that each man $m \in B$ is better off in M_s than in M_0. Given the falsified lists of men, the stability of M_s implies that every man $m \in B$ is preferred by his partner $M_s(m)$ over any other man $m' \in \mathcal{M} - B$ who puts $M_s(m)$ on the left of his preference list. In any execution of the Gale-Shapley algorithm with the true preference lists, the men in B must be rejected by their M_s-partners, and this rejection can be caused only by another man $m' \in B$. Moreover, after this rejection, his M_s-partner can be engaged only to men in B. Without loss of generality, assume that m is the last person in B who is rejected by his M_s-partner. At the point of this rejection, all the M_s-partners of men in B except $M_s(m)$ must have been engaged, and only to men in B. However, the rejection of m implies that $M_s(m)$ is also engaged to another man in B. Hence, $|B|$ women are engaged to $|B-1|$ men when the last rejection takes place, and we reach the desired contradiction. □

3 Coalition Strategy

In this section we present the coalition strategy. An example could be found in Figure 2.

Coalitions. We now formally explain the coalition strategy. A coalition is comprised of two parts: *cabal* and *accomplices*. Each man in the cabal prefers another's partner to his own and would be happier if they can exchange; the accomplices are the men who need to falsify their lists to help them accomplish this goal.

M_0-matching		M_s-matching	
Men's List	Women's List	Men's (Falsified) List	Women's List
A: ab**e**dc	a: **C**BDAE	A:<u>a**e**dcb</u>	a: **C**BDAE
B: be**d**ac	b: **D**EABC	B:**b**edac	b: DEA**B**C
C: eb**a**cd	c: BCD**E**A	C:eb**a**cd	c: BCD**E**A
D: da**b**ce	d: A**B**CED	D:**d**abce	d: ABCE**D**
E: edb**c**a	e: **A**BECD	E: <u>**e**c**a**bd</u>	e: **A**BECD
The Match Ranking for men in M_0: A(e,3), B(d,3), C(a,3), D(b,3), E(c,4)			
The Match Ranking for men in M_s: A(e,3), B(b,1), C(a,3), D(d,1), E(c,4)			

Fig. 2. Men A and E falsify their lists to help men B and D get a better partner. Falsified lists are underlined.

Definition 1. *The cabal of a coalition $K = (m_1, m_2, \cdots, m_{|K|})$ is a list of men such that each man $m_i, 1 \leq i \leq |K|$, prefers $M_0(m_{i-1})$ to his own partner $M_0(m_i)$, indices taken module $|K|$.*

Having formed the cabal, the men in the cabal need to to enlist the help of accomplices. Suppose man m_i in the cabal wishes to be matched to some woman w (who is m_{i-1}'s partner). All other men (accomplices) putting her on the left of their lists, if they are ranked higher than m_i in w's list, should avoid proposing to her by shifting her to the right of their lists (as implied by Lemma 1).

Definition 2. *The accomplices of cabal $K = (m_1, m_2, ...m_{|K|})$ is a set of men $A(K) \subseteq \mathcal{M}$ such that $m \in A(K)$ if*

1. *$m \notin K$, for any $m_i \in K$, if $M_0(m_i) \succ_m M_0(m)$ and $m \succ_{M_0(m_i)} m_{i+1}$, or*
2. *$m = m_j \in K$, for any $m_i \in K, i \neq j$, if $M_0(m_i) \succ_{m_j} M_0(m_{j-1})$ and $m_j \succ_{M_0(m_i)} m_{i+1}$.*

Note that cabal K and its accomplices $A(K)$ might not be disjoint, i.e., the people in the cabal might have to falsify their lists as well. An immediate consequence of the Dubins-Freedman Theorem is that $A(K) \cup K \supset K$.

We can now present the main result of this section.

Theorem 2. *Coalition Strategy: If in a coalition $C = (K, A(K))$, each accomplice $m \in A(K)$ submits a falsified list of the form $(\pi_r(P_L(m) - X), M_0(m), \pi_r(P_R(m) \cup X))$, and if*

- *$m \in A(K) - K, X = \{w | w = M_0(m_i) \in M_0(K), m \succ_w m_{i+1}\}$*
- *$m = m_j \in A(K) \cap K, X = \{w | w = M_0(m_i) \in M_0(K), w \succ_{m_j} M_0(m_{j-1}), m_j \succ_w m_{i+1}\}$,*

then in the resulting M_s-matching, $M_s(m_i) = M_0(m_{i-1})$ for $m_i \in K$ and $M_s(m) = M_0(m)$ for $m \notin K$.

Proof. As implied by Lemma 1, no man will be rejected by his M_0-partner, since men only shift some women from the left to the right of their lists. Moreover, no man m_i in K is going to be rejected by his preferred partner $M_0(m_{i-1})$, since all the accomplices have altered their lists. Finally, men not in the cabal can get only their M_0-partners and men in the cabal can get only their preferred

M_s-partners. If this is not so and some subset of men get even better partners, as in the proof of Lemma 2, we can use a pigeonhole argument to refute this possibility. □

The coalition strategy is the *only* strategy that has the nice property of ensuring that some men are better off and every liar is at least as well off as before. One might wonder whether there exist other strategies by means of which liars can manipulate the outcome at the expense of honest men without hurting themselves. The following theorem precludes this possibility.

Theorem 3. *The coalition strategy is the only way for men to falsify their lists such that in the resulting M_s-matching, some men are better off and every liar is at least as well off as when he is truthful.*

Proof. We proceed by contradiction. Suppose there exists another strategy for men such that some men can be better off at the expense of honest men, and all liars are at least as well off as when they are honest. Say some man m (whether he is honest or not) is better off by being matched to the partner of some honest man m', i.e. $M_s(m) = M_0(m')$, while the honest man m' is worse off. We claim that $(m', M_0(m'))$ must be a blocking pair in M_s, because (1) the stability of M_0 implies that $m' \succ_{M_0(m')} m$, and (2) since m' is honest, $M_0(m') \succ_{m'} M_s(m')$. □

Theorem 3 has an important implication: Liars, if intending to help other men (or themselves) get better partners, either have to adopt the coalition strategy (in which no one gets hurt) as defined in Theorem 2, or must accept worse partners for themselves. This observation prompts us to devise another strategy in Section 5. The algorithms for finding the coalitions (cabals) can be found in [7]. We discuss theoretical implications that directly follow from the coalition strategy.

Cabalists and Hopeless Men. Based on the preference lists and the M_0-matching, a large number (which can be exponential) of coalitions may exist. We define a man to be one of the *cabalists* $\mathcal{K}$ if he belongs to any one of the cabals of the coalitions; otherwise, he is one of the *hopeless men* $\mathcal{H}$. By this definition, men fall into two categories: $\mathcal{M} = \mathcal{K} \bigcup \mathcal{H}$ and $\mathcal{K} \bigcap \mathcal{H} = \emptyset$. Apparently, hopeless men cannot benefit from utilizing the coalition strategy. The following lemma, implying at least one man does not have incentive to cheat, is important in proving our next major result.

Lemma 3. *Whatever the true preference lists, there always exists at least one hopeless man, i.e., $\mathcal{H} \neq \phi$.*

Proof. If woman w is the last woman receiving a proposal during the execution of the Gale-Shapley algorithm, then (1) she has not received any other proposal before, and (2) she is not in the left portion of any man's preference list. If this is not so, then when the last proposal is made to w, she will either reject the proposer or dump her former partner. In both cases, this "last" proposal will not terminate the algorithm.

Since the last woman w receiving a proposal is not on the left of any man's preference list, $M_0(w)$ cannot belong to any cabal. Hence he must be one of the hopeless men. □

4 Impossibility of Forming Leagues

The coalition strategy has one unsatisfactory aspect: The Dubins-Freedman Theorem ordains that for every coalition, at least one accomplice does not gain from lying and hence has little motivation of doing so. Can we devise a stratagem such that everyone is predisposed to cheat? In this section, we show that even with a randomized strategy, we still cannot overcome the problem that some men lack the motivation of lying.

We formulate what would be a successful randomized strategy for men.

Definition 3. *A* league *is a subset $L \subseteq \mathcal{M}$ with the following properties. Each man $m_i \in L$ has a set of possible preference lists $s_i = \pi(\mathcal{W})$, and a joint probability distribution F:$s_1 \times s_2 \cdots \times s_{|L|} \rightarrow [0,1]$ exists such that for every man $m_i \in L$:*

- *(Positive Expectation): $E[Rank(M_s(m_i))] > Rank(M_0(m_i))$.*
- *(Elimination of Risk): If in event E, $Rank(M_0(m_i)) > Rank(M_s(m_i))$, then $Prob(E) = 0$.*

Based on Theorem 3, the two requirements imply that the only choice is to employ a mix of coalition strategies. We can randomly pick some coalition contained in the league and realize the strategy accordingly. The problem then boils down to whether we can find a union of coalitions $C_i = (K_i, A(K_i))$ such that $L = \bigcup_i K_i = \bigcup_i A(K_i)$. In other words, in this league, each accomplice belongs to the cabal of some coalition, and thus has a chance to improve the rank of his partner (and hence the incentive to lie).

A league would circumvent the Dubins-Freedman Theorem, by allowing every liar to improve the rank his partner (in a randomized sense) with no risk. However, leagues do not exist.

Theorem 4. *In any coalition $C = (K, A(K))$, at least one accomplice is a hopeless man, i.e., $A(K) \bigcap \mathcal{H} \neq \emptyset$.*

Proof. We first consider *maximal* coalitions and then go on to more general cases. A coalition $C = (K, A(K))$ is maximal if $K = \mathcal{M} - \mathcal{H}$. For every man m_i in the cabal of this maximal coalition, we move his preferred partner $M_0(m_{i-1})$ in the cabal to the front of his list and his M_0-partner $M_0(m_i)$ to the second place. Note that due to Lemma 1, after this alteration of the lists, a man in the cabal can be matched only to either his original M_0-partner or his preferred partner in the cabal.

Arrange the proposal sequence of the Gale-Shapley algorithm in the following way: all men in $\mathcal{M} - \mathcal{H}$ propose first and are temporarily engaged to their

preferred partners in the cabal. In the resulting matching, the Dubins-Freedman Theorem tells us that it is impossible that every liar gets a better partner, so at least one person m_j in the cabal is matched to his M_0-partner $M_0(m_j)$; consequently, m_{j+1} also can be matched only to his original M_0-partner $M_0(m_{j+1})$ and so forth. The only way to break the "balance" of this cabal is that some hopeless man m^* (there exists at least one hopeless man, as indicated by Lemma 3) proposes to some woman who is a partner of a man in the cabal and he is preferred by this woman over him. Hence, m^* must be one of the accomplices in this coalition.

If the coalition C is not maximal, i.e., $|K| < |\mathcal{M} - \mathcal{H}|$, we still can apply the above argument, with a little more complication. First, choose some cabalist m not in K, and move his M_0-partner to the front of his preference list. Then, for all other cabalists m_k, if $M_0(m) \succ_{m_k} M_0(m_k)$, shift $M_0(m)$ to the end of his list. Note that by Lemma 1, these operations will not make any man get a worse partner. We claim that now m becomes a hopeless man and the resulting M_s-matching is still identical to M_0. The reasons are as follows: (1) If there exists any other cabal K' involving m, then the coalition containing the cabal K' cannot be realized. Recall that for a coalition to be formed, the men in the cabal K' must have better partners, but m can only be matched to his M_0-partner, who is on the front of his list. (2) Cabals other than K not involving m also cannot be realized, because all we have done is to shift $M_0(m)$ to the right of other men's preference lists. If a coalition containing such a cabal is to be realized, the accomplices of the coalition have to shift the preferred women in the cabal to the right of their lists. But $M_0(m)$ is not one of them. Hence, such a coalition cannot succeed.

By applying the above argument repeatedly, we can make all cabalists in $\mathcal{M} - (\mathcal{H} \cup K)$ become hopeless men. For the men in the cabal K (which is now a maximal coalition), use the same argument we have used before: for each $m_i \in K$, shift $M_0(m_{i-1})$ and $M_0(m_i)$ to the first two places in his list. Let all men in K propose first. The "balance" of K can be broken only by some true hopeless men (those originally in $\mathcal{H}$, instead of those false ones we created, because the latter will only propose to their M_0-partners and stop. Moreover, Lemma 3 guarantees that $\mathcal{H}$ be non-empty). By the above argument, we reach the conclusion that every coalition has at least one accomplice who is a hopeless man. □

By Theorem 4, we know that an all-win league is impossible. A hopeless man never improves his lot by the coalition strategy, which means that he can never attain the first requirement in Definition 3. Combining Theorem 3 and Theorem 4, we derive our major result in this section:

Theorem 5. *It is impossible to find a league, thus a successful randomized strategy as defined in Definition 3 cannot be formed.*

5 In Pursuit of Motivation

In this section, we show it is possible to devise a randomized strategy in which every cheating man can expect to get a better partner. The crucial point is that

these liars must be willing to take the risk of getting worse partners. We first introduce another lying strategy.

Lemma 4. *Victim Strategy: Suppose* $M_0(m) \succ_m M_0(m')$ *and* $M_0(m) \succ_{m'} M_0(m')$. *And for all* $m_i \in \mathcal{M} - \{m, m'\}$, *if* $M_0(m') \in P_L(m_i)$, *then* $m \succ_{M_0(m')} m_i$. *Let* m *submit a falsified list of the form* $(\pi_r(P_L(m) \cup M_0(m')), M_0(m), \pi_r(P_R(m) - M_0(m')))$, *then in the resulting matching* M_s*:*

1. *For* m *(the victim),* $M_s(m) = M_0(m')$;
2. *For* m' *(the benefiter),* $M_s(m') \succ_{m'} M_0(m')$;
3. *For men* $m_i \in \mathcal{M} - \{m, m'\}, M_s(m_i) \succeq_{m_i} M_0(m_i)$.

Proof. We construct a stable matching M^* as follows: Retain all the couples in M_0 except exchange the partners of m and m'.

We claim that the constructed M^* is stable, since every man, except m, has either the same or a better partner. For m, he also gets a "better" partner, since $M_0(m')$ is now on the left of his perjured preference list. And there is no danger of the existence of a blocking pair containing $M_0(m')$, since m is more favored by $M_0(m')$ than any other man putting her on the left of his list.

If the constructed M^* is not men-optimal, then the true M_s will still have the stated properties. Men-optimality of M_s ensures that every man gets the best possible partner among all stable matchings. The only exception is m, who can not get a better partner than $M_0(m')$ in M_s, because of the Dubins-Freedman Theorem. □

M_0-matching		M_s-matching	
Men's List	Women's List	Men's (Falsified) List	Women's List
A: bd**a**ce	a: B**A**DCE	A:<u>bd**c**ae</u>	a: BAD**C**E
B: cd**b**ae	b: C**B**ADE	B:cd**b**ae	b: C**B**ADE
C: ad**c**be	c: A**C**BED	C:**a**dcbe	c: **A**CBED
D: a**e**bcd	d: **E**DBCA	D:a**e**bcd	d: **E**DBCA
E: **d**abce	e: **D**BECA	E: **d**abce	e: **D**BECA
The Match Ranking for men in M_0: A(a,3), B(b,3), C(c,3), D(e,2), E(d,1)			
The Match Ranking for men in M_s: A(c,4), B(b,3), C(c,1), D(e,2), E(d,1)			

Fig. 3. An example: Man A shifts woman c from the right to the left of his list. He gets woman c and man C gets woman a. Note man B (C) also can use the same strategy to help man A (B).

A simple example for the victim strategy can be found in Figure 3. The problem is the practicality of the victim strategy: where can we find people with such a self-sacrificing spirit? The randomness of the victim strategy makes possible that some men be willing to play the role of victim (occasionally). Here we present an easy example. As shown in Figure 3, a successful alliance is composed of men A, B and C. Man A (or B, or C) can play the role of victim to help man C (or A, or B). Suppose we assign the probability of 1/3 to each one of them to play the victim; then the expected rank of their partner would be 8/3, which is an improvement.

6 Coalition Strategy in Random Stable Matching

In this section, we modify the coalition strategy for the scenario that the stable matching is chosen at random. It is well-known that the set of stable matchings compose a distributive lattice. In the following, we investigate what happens to the lattice when a subset of men adopt the coalition strategy. It is easy to see that if men shift women from the left to the right of their lists but without permuting the right portions of their lists, all the original stable matchings remain stable with regard to the falsified lists.

The following is good news for cheating men: even if the authority all of a sudden decides to change the men-optimal matching to a women-optimal one, they will not be worse off than when they are truthful.

Lemma 5. *Given a subset of men $S \subseteq \mathcal{M}$, let every member $m \in S$ submit a falsified list of the form $(\pi_r(P_L(m) - X), M_0(m), \pi_r(X), P_R(m))$, $X \subseteq P_L(m)$. Then, in the women-optimal matching, every man still gets his M_0-partner.*

Proof. (**Sketch**) This can be observed by the fact that men never receive proposals from women ranking higher than their M_0-partners. □

By Lemma 5, when the coalition strategy is adopted, the new women-optimal matching will be identical to M_z, the original one when everyone is truthful. Therefore, M_z is still the minimal element in the new lattice $\mathcal{L}'$. As previously alluded to, the original men-optimal matching M_0 is also an element in the new lattice $\mathcal{L}'$ (but no longer the maximal element, which is now the new men-optimal matching M_s realized by the coalition strategy). We next show that there is no newly-created stable matching dominated by M_0.

As defined by Gusfield and Irving [6], given a stable matching M, an (exposed) rotation is a circular list $\sigma = ((m_1, w_1), (m_2, w_2), \cdots, (m_{|\sigma|}, w_{|\sigma|}))$, indices taken modulo $|\sigma|$, such that:

1. $M(m_i) = w_i$,
2. $m_{i-1} \succ_{w_i} m_i$,
3. If $w_i \succ_{m_i} w \succ_{m_i} w_{i+1}$, then $M(w) \succ_w m_i$.

"Eliminating" the rotation σ from M means that every man m_i changes his partner from w_i to w_{i+1} (a worse partner for him). Eliminating an exposed rotation σ in a stable matching M creates another stable matching $M' = M/\sigma$, which lies immediately below M in the lattice [6, Lemma 2.5.2]. The following lemma explains what would happen to these rotations when men shift women from the left to the right of their lists. We present a slightly stronger result then is required. Let A and B be any ordered lists. $\prod_r(A, B)$ denotes any mixed list of A and B such that the order of elements in A and in B is still preserved.

Lemma 6. *Given true preference lists, let M and $M' = M/\sigma$ be two stable matchings where σ is exposed in M and $M_0 \succeq M$. Given a subset of men $S \subseteq \mathcal{M}$, let every member $m \in S$ submit a falsified list of the form $(\pi_r(P_L(m) - X), M_0(m), \prod_r(\pi_r(X), P_R(m)))$, $X \subseteq P_L(m)$, then in M, σ is still exposed with regard to the falsified lists.*

Proof. (**Sketch**) Consider any man $m_i \in S$ involved in an original rotation σ. Consider any such woman w being shifted by m_i (who originally ranks higher than $M_0(m_i)$ in m_i's list); w must prefer her partner in M_0 over m_i, otherwise, (m_i, w) blocks M_0. By the fact that $M_0 \succeq M$, in M, w can not be matched to some one who ranks lower than $M_0(w)$ in her list (otherwise, $(M_0(w), w)$ blocks M), so $M(w) \succeq_w M_0(w) \succ_w m_i$. Therefore, the fact that in the falsified list of m_i, w appears between w_i and w_{i+1} does not affect the rotation σ. □

Gusfield and Irving prove that a stable matching can be generated by repeatedly eliminating the exposed rotations [6, Corollary 2.5.2]. Combining these observations, we have,

Theorem 6. *Given a subset of men $S \subseteq \mathcal{M}$, let every member $m \in S$ submit a falsified list of the form $(\pi_r(P_L(m) - X), M_0(m), \pi_r(X), P_R(m)), X \subseteq P_L(m)$.*

- *The new lattice $\mathcal{L}' \supseteq \mathcal{L}$, the old lattice.*
- *The set of rotations found along any maximal chain of $\mathcal{L}'$ is a superset of rotations found along any maximal chain of $\mathcal{L}$.M_0 can be generated by eliminating from M_s all the newly-created rotations with regards to the falsified lists. Moreover, the newly-created rotations only involve men who get a strictly better partner in M_s.*
- *For a new stable matchings $M^\flat$ in $\mathcal{L}' - \mathcal{L}$, either $M^\flat \| M_0$, or $M^\flat \succ M_0$.*

We now have all the necessary tools to present the major result.

Theorem 7. *Suppose M_s is a men-optimal stable matching realizable by the coalition strategy and $C = (K, A(K))$ be the corresponding coalition. Let men in the coalition cheat as follows:*

- *If $m \in A(K) - K$, m submits a falsified list of the form $(\pi_r(P_L(m) - X), M_0(m), \pi_r(X), P_R(m))$, where X is the set of women defined by the coalition strategy for realizing M_s.*
- *If $m \in K$, m submits a falsified list of the form $(M_s(m), M_0(m), \pi_r(P_L(m) - M_s(m)), P_R(m))$.*

Then in all the newly-created stable matchings, every man in the coalition C gets a partner whose rank is at least as high as his M_0-partner.

Proof. Consider the men in the cabal K. Since there is no woman between their M_s-partners and M_0-partners, there is only one rotation $\hat{\delta}$ between M_s and M_0. For a contradiction. Suppose there exists a newly-created matching M^ϕ in which men in K get worse partners than their M_0-partners, $\hat{\delta}$ must be eliminated. By Theorem 6, M^ϕ must be one of the stable matchings in the original lattice $\mathcal{L}$.

For the accomplices in $A(K) - K$, if in a stable matching, they, along with the men in the cabal K, get worse partners than their M_0-partners, the same argument in the preceding paragraph can be applied. The special case that needs to be taken care of is some newly-created stable matching M^ϕ which can be generated from M_s by eliminating some (original) rotations excluding $\hat{\sigma}$ (so the

men in the cabal K are still matched to their M_s-partners). Suppose that in M^ϕ, some accomplice m gets a worse partner than his M_0-partner. By Lemma 6, m must be matched to some woman ranks lower (in the falsified list) than all those women he shifts from the $P_L(m)$ (who now ranks lower than $M_0(m)$ but still higher than all women in $P_R(m)$). We claim that M^ϕ cannot be stable. Suppose w be any woman being shifted in m's list. Since $\hat{\sigma}$ is not eliminated, w is still a partner of some man in the cabal K, and, by definition of an accomplice, w prefers m over that man in the cabal. Therefore (m, w) blocks M^ϕ. □

In the newly-created matchings, since men in the coalition only get partners ranking at least as high as their M_0-partners, the following is immediate:

Corollary 1. *Suppose men submit their preference lists as defined in Theorem 7. Each man in the coalition has a new probability distribution over his partners which majorizes the original one when everyone is truthful.*

Hence, by this corollary, the accomplices are finally rewarded for their cooperation. As opposed to the Dubins-Freedman Theorem, in this random stable matching setting, a subset of men can cheat together and *all* get (expectedly) better partners.

7 Conclusion and Related Work

In this work, we propose a variety of lying strategies, both deterministic and randomized, for men in the Gale-Shapley algorithm. We also strengthen the classical theorem stating that honesty is the best policy for men. Even with a randomized strategy, this theorem still holds. The theorem can only be circumvented if liars are willing to take risk. We also display the greater applicability of the coalition strategy in the context of random stable matching.

Given that there are so many possible coalitions, a question inevitably arises: how can men in the coalition be sure that there will not be double-crossers, who suddenly decide to switch their allegiance to other coalitions? Even if there is only one coalition, how can men in the coalition be sure that some accomplices will not back out? The answer is that every coalition strategy is a *strong equilibrium point* for men. Even if a subset of liars betray their co-conspirators, they cannot all to get better partners (than the ones agreed upon). This can be easily shown by applying the Dubins-Freedman Theorem to the falsified lists.

The coalition strategy causes women to be worse off. In some situations women can have counter-measures if any one of them is going to receive more than one proposal. However, the Gale-Shapley algorithm has a feature that can be exploited by men: women cannot say no when they receive their first proposal. In other words, men can get together and decide upon a "best" coalition strategy by formulating the problem into the *house-swapping* problem. With each man initially being assigned his M_0-partner, the goal is to find the *strict core* of the market [12]. Once men agree with one another which women they are supposed to be matched to, they put these women at the tops of their lists. The related algorithmic details can be found in [7].

Related Work. The stable marriage problem, due to its theoretical appeal and practical applications, has spawned a large body of literature. For a summary, see [6, 9, 13]. Several early results [1, 2, 5, 10] indicated the futility of men-lying and this probably caused later work to focus mostly on women-lying strategies. Gale and Sotomayor [4] presented the women lying strategy of truncating their lists. Immorlica and Mahdian [8] showed that if men have preference lists of constant size while women have complete lists and both are drawn from an arbitrary distribution of preference lists, the chance of women gaining from lying is vanishingly small. Teo et al. [15] suggested lying strategies for an individual woman. About permuting men's preference lists to manipulate the outcome of the matching, there is an example in the book of Gusfield and Irving [6, P.65]. Another example is given by Roth and Sotomayor [13, P.115]. Roth and Vate [14] discussed strategy issues when the stable matching is chosen at random. They proposed a truncation strategy and showed that every stable matching can be achieved as an equilibrium in truncation strategies.

Acknowledgment

I thank my adviser Peter Winkler for many helpful discussions.

References

1. Gabrielle Demange, David Gale, and Marilda Sotomayor. A further note on the stable matching problem. *Discrete Applied Mathematics*, 16:217–222, 1987.
2. Lester Dubins and David Freedman. Machiavelli and the Gale-Shapley algorithm. *American Mathematical Monthly*, 88:485–494, 1981.
3. David Gale and Lloyd Shapley. College admissions and the stability of marriage. *American Mathematical Monthly*, 69(1):9–15, 1962.
4. David Gale and Marilda Sotomayor. Ms. Machiavelli and the stable matching problem. *American Mathematical Monthly*, 92:261–268, 1985.
5. David Gale and Marilda Sotomayor. Some remarks on the stable matching problem. *Discrete Applied Mathematics*, 11:223–232, 1985.
6. Daniel Gusfield and Robert Irving. *The Stable Marriage Problem.* The MIT Press, 1989.
7. Chien-Chung Huang. How hard is it to cheat in the gale-shapley stable matching algorithm? Technical Report TR2005-565, Computer Science Department, Dartmouth College.
8. Nicole Immorlica and Mohammad Mahdian. Marriage, honesty, and stability. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 53–62, 2005.
9. Donald Knuth. *Mariages stables et leurs relations avec d'autre problèmes combinatoires.* Les Presses de l'université de Montréal, 1976.
10. Alvin Roth. The economics of matching: Stability and incentives. *Mathematics of Operations Research*, 7(4):617–628, 1982.
11. Alvin Roth. The college admissions problem is not equivalent to the marriage problem. Journal of Economic Theory, 27:75–96, 1985.

12. Alvin Roth and Andres Postlewaite. Weak versus strong domination in a market with indivisible goods. *Journal of Mathematical Economics*, 4:131–137, 1977.
13. Alvin Roth and Marilda Sotomayor. *Two-sided matching: A study in game-theorectic modeling and analysis*. Cambridge University Press, 1990.
14. Alvin Roth and Vande Vate. Incentives in two-sided matching with random stable mechanisms. *Economic Theory*, 1 (1):31–44, 1991.
15. Chung-Piaw Teo, Jay Sethuraman, and Wee-Peng Tan. Gale-Shapley stable marriage problem revisited: Strategic issues and applications. *Management Science*, 47:1252–1267, 2001.

Approximating Almost All Instances of MAX-CUT Within a Ratio Above the Håstad Threshold*

A.C. Kaporis[1], L.M. Kirousis[1,2], and E.C. Stavropoulos[1]

[1] University of Patras, Department of Computer Engineering and Informatics
GR-265 04 Patras, Greece
{kaporis, kirousis, estavrop}@ceid.upatras.gr
[2] Research Academic Computer Technology Institute, P.O. Box 1122, GR-261 10
Patras, Greece

Abstract. We give a deterministic polynomial-time algorithm which for any given average degree d and *asymptotically almost all* random graphs G in $\mathcal{G}(n, m = \lfloor \frac{d}{2} n \rfloor)$ outputs a cut of G whose ratio (in cardinality) with the maximum cut is at least 0.952. We remind the reader that it is known that unless P=NP, for no constant $\epsilon > 0$ is there a MAX-CUT approximation algorithm that for *all inputs* achieves an approximation ratio of $(16/17) + \epsilon$ $(16/17 < 0.94118)$.

1 Introduction

There is a vast and growing literature on approximation algorithms for NP-hard problems. Both in the direction of designing algorithms that give good approximations, as well as in the direction of showing, under a putative hypothesis like P≠NP, that no approximation better than a given bound exists. In this work, we concentrate on the problem of MAX-CUT, that of partitioning the vertex set V of a graph $G = (V, E)$ in two parts so that the number of edges joining vertices in different parts is as large as possible. In more colorful language, MAX-CUT is the problem of coloring the vertices of a graph with two colors (red or blue) so that the bichromatic edges are as many as possible. It is probably needless to elaborate on the interest, from the point of view of either theory or practice, of the NP-hard optimization problem MAX-CUT. Just as an example, let us mention the early considerations of MAX-CUT in relation to circuit layout design and Statistical Physics mentioned in [1] (as pointed out in [4]). In the language of Statistical Physics, MAX-CUT is equivalent to computing the ground energy of the antiferromagnetic Ising model defined on graphs [15].

* The authors are partially supported by European Social Fund (ESF), Operational Program for Educational and Vocational Training II (EPEAEK II), and particularly *Pythagoras*. The second author is partially supported by Future and Emerging Technologies programme of the EU under contract 001907 *"Dynamically Evolving, Large-Scale Information Systems (DELIS)."*

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 432–443, 2006.

For maximization problems, like MAX-CUT, we say that an algorithm $\mathcal{A}$ achieves an approximation ratio $0 < \alpha < 1$, if for any input I, the output of the algorithm $\mathcal{A}(I)$ on I relates to an optimum solution $\mathrm{OPT}(I)$ for I as in:

$$\frac{|\mathcal{A}(I)|}{|\mathrm{OPT}(I)|} \geq \alpha.$$

Similarly, we define the approximation ratio of minimization problems. For general graphs, the best MAX-CUT approximation algorithm is, for more than a decade now, the one by Goemans and Williamson [12], which can achieve a ratio arbitrarily close (from below) to

$$\alpha_{\mathrm{GW}} = \min_{0 \leq \theta \leq \pi} \frac{2}{\pi} \frac{\theta}{1 - \cos\theta} > 0.87856.$$

Under the "Unique Games" and the "Majority is Stablest" conjectures, the above approximation ratio was shown to be optimal by Khot et al. [16] (however, very recently a hypothesis only slightly stronger than the Unique Games conjecture was falsified [5]).

Also by a now classical inapproximability result by Håstad [13] and Trevisan et al. [20], unless P≠NP, MAX-CUT cannot be approximated for general graphs by a deterministic algorithm that attains a ratio strictly exceeding 16/17 ($16/17 < 0.94118$).

Let now $\mathcal{G}(n;d)$ be the probability space of random graphs with n vertices and $m = \lfloor \frac{d}{2} n \rfloor$ edges selected uniformly at random. It is convenient for the probabilistic calculations to allow repetitions and even self-loops in the selection of edges. This does not affect the results as such selections happen with vanishingly small probability as n grows large. We say that a property $\mathcal{E}$ holds for asymptotically almost all (a.a.a.) random graphs from $\mathcal{G}(n;d)$ if $\lim_n \mathbf{Pr}[G \in \mathcal{G}(n;d) \ \& \ \mathcal{E} \text{ holds for } G] = 1$. Notice that the negative result for approximation ratios $> 16/17$ does not exclude the possibility of a deterministic algorithm that achieves a ratio of $(16/17) + c$ (c a positive constant) for a.a.a. input instances from $\mathcal{G}(n;d)$, for any given fixed d (see e.g. the pioneering work of Frieze and McDiarmid on graph algorithms on random instances [11]).

With respect to a different problem, namely MAX-SAT, Fernandez de la Vega and Karpinski [8] analyzed an algorithm that achieves an approximation ratio of 8/9 for a.a.a. instances, with any given ratio of clauses to variables (Håstad [13] has proved that there is no approximation algorithm for MAX-SAT whose ratio strictly exceeds 7/8). The ratio of 8/9 was further improved to 19/20 by Interian [14]. These algorithms for MAX-SAT are Davis-Putnam-style heuristics that do not take into account the number of occurrences of the variable selected to be assigned the value "true" at each step.

Similar heuristics, that ignore degree considerations of the vertices to be put into each part of the cut under construction, have been analyzed for the case of MAX-CUT, or more general versions of it like MAX-k-CUT, in various papers (see [15, 6, 7]), giving a series of interesting lower bound results for the optimal cut. The fact that degree considerations are not taken into account in these

algorithms, greatly simplifies their probabilistic analysis. However, as far as the question of the ratio of the size of the output of these algorithms to the size of the optimal cut is concerned, they all yield values that are far below the Håstad threshold, even below the Goemans-Williamson ratio, for values of the average degree in a sizable interval. Also heuristics that take into account degree considerations, but for different graph problems, are analyzed in the work of Beis et al. and others (see [2, 3] and references therein).

To break the Håstad barrier for MAX-CUT (for a.a.a. input instances with any given d), it became necessary to follow a double front approach. On one hand, since the size of optimum cut is not known, we had to find improved upper bounds for the optimum cut. On the other, we had to considerably improve the known algorithmic lower bounds. So that using both bounds we could come up with a ratio that exceeds 16/17. Both upper and lower bounds are computed not for the graph itself, but for its 2-core, the maximum induced subgraph whose vertices have degree at least 2. The reason being that, as it is easy to prove, the edges of a graph not belonging to its 2-core, belong to any max cut. Therefore, once we have a max cut of the 2-core, then a max cut of the original graph can be found by adding to the cut the edges that are outside the 2-core. This pruning preprocessing phase considerably improves the bounds, but necessitates carrying our analysis not in $G \in \mathcal{G}(n; d)$, but in the uniform probability space of graphs with a given degree sequence. Our algorithm for the lower bound takes into account the degree of each vertex. The numerical analysis makes use of computer aided computations.

The approximation ratio we get, besides crossing the Håstad threshold, substantially improves the Goemans-Williamson value (0.87856) and thus, to the best of our knowledge, constitutes the first after more than a decade improvement of the approximation ratio of MAX-CUT, valid for general graphs (but only for a.a.a. input instances with any given average degree). In the next section we give some necessary formal definitions, state the main result and give some preliminary facts. The main tools of the proof are given in the sections that come after the next one.

2 Preliminaries

Definition 1. *Given a cut $\mathcal{C}$ of a graph G, the cut size of $\mathcal{C}$, denoted by $|\mathcal{C}|$, is the number of edges of G that connect vertices in different parts of $\mathcal{C}$ (bichromatic edges).*

We now give definitions of a.a.a. upper and lower bounds that are given as percentages of (scaled with respect to) m, the number of edges.

Definition 2. *A function $\mathrm{ub} : \mathbb{R}^+ \to \mathbb{R}^+$ a.a.a. defines a scaled (with respect to the number of edges m) upper bound $\mathrm{ub}(d)$ for the maximum cut size $\mathrm{mc}(G)$ of a random graph $G \in \mathcal{G}(n; d)$ if*

$$\lim_n \mathbf{Pr}\left[G \in \mathcal{G}(n; d) \;\;\&\;\; (\mathrm{ub}(d) + o(1))m \geq \mathrm{mc}(G)\right] = 1, \forall d > 0.$$

Definition 3. *Given (i) a function* $\mathrm{lb} : \mathbb{R}^+ \to \mathbb{R}^+$ *and (ii) a deterministic algorithm* $\mathcal{A}$ *that on input a graph* G *outputs a cut* $\mathcal{A}(G)$ *of* G*, we say that* $\mathcal{A}$ *a.a.a. establishes a scaled (with respect to the number of edges* m*) lower bound* $\mathrm{lb}(d)$ *on the maximum cut size of a random graph in* $G \in \mathcal{G}(n;d)$ *if*

$$\lim_n \mathbf{Pr}\left[G \in \mathcal{G}(n;d) \;\;\&\;\; (\mathrm{lb}(d) - o(1))m \leq |\mathcal{A}(G)|\right] = 1, \forall d > 0.$$

Proposition 1. *If there are functions* ub *and* lb *and an algorithm* $\mathcal{A}$ *as in Definitions (2) and (3), then* $\forall d$ *and* $\forall \epsilon > 0$*,* $\mathcal{A}$ *achieves an approximation ratio* $\frac{\mathrm{lb}(d)}{\mathrm{ub}(d)} - \epsilon$ *for* MAX-CUT *for a.a.a. input instances* $G \in \mathcal{G}(n;d)$*.*

Proof. Immediate from the definitions. □

Theorem 1. *There are functions* ub *and* lb *and an algorithm* $\mathcal{A}$ *as in Definitions (2) and (3) such that* $\forall d$*,* $\frac{\mathrm{lb}(d)}{\mathrm{ub}(d)} > 0.952$*.*

Proof. For the case when $d < 1$, then by the proof in [7, Theorem 19] we know that the cut obtained by considering (i) all edges of the tree-components of G, (ii) all edges of its even cyclic components and (iii) all edges but one of its odd cyclic components has cardinality equal to the total number of edges of G within an $o(1)$ additive term. This procedure defines the algorithm $\mathcal{A}$. Also, we set $\mathrm{ub}(d) = \mathrm{lb}(d) = 1$.

For the case of large d, first observe that by coloring red an arbitrary half of the vertices of G, we get a trivial lower bound $\mathrm{lb}(d) = (1/2) - \epsilon$, for any $\epsilon > 0$. By combining this trivial lower bound with the upper bound $\mathrm{ub}(d) = 1/2 + \sqrt{(\ln 2/d)}$ [4], we easily get by solving for d the equation

$$\frac{1/2}{1/2 + \sqrt{(\ln 2/d)}} = \frac{16}{17},$$

that the Theorem holds for $d > 710$,

As for the interval $1 \leq d \leq 710$, in Section 4 we define the function ub, while in Section 5 we describe and analyze the algorithm $\mathcal{A}$ and define the function lb. The computations involved are computer-aided (but the probabilistic analysis and the derivations of all formulas are analytic). The computer-aided analysis shows that the for $d \geq 20$, the ratio $\frac{\mathrm{lb}(d)}{\mathrm{ub}(d)}$ is bounded below by numbers greater than 0.952 by 0.01 or more. Actually for $d \geq 20$, easier upper and lower bound functions, some of which already given in the literature [15, 6, 7], yield a ratio $\frac{\mathrm{lb}(d)}{\mathrm{ub}(d)}$ that easily exceeds 0.952. So in the following sections we concentrate in the interval $[1, 20]$, where the real difficulty lies, i.e. the interval where the ratio $\frac{\mathrm{lb}(d)}{\mathrm{ub}(d)}$ for the improved upper and lower bound functions that we define closely approaches from above the value 0.952. □

Corollary 1 (Main Result). *There is a deterministic algorithm* $\mathcal{A}$ *such that for any average degree* $d > 0$*,* $\mathcal{A}$ *achieves an approximation ratio* 0.952 *for* MAX-CUT *for a.a.a. random graphs in* $\mathcal{G}(n;d)$*.*

Proof. Immediate from Theorem 1 and Proposition 1. □

3 The 2-Core

The 2-core of a graph G is defined to be the largest induced subgraph of G with minimum degree at least 2. For technical reasons, we use an essentially equivalent but formally slightly different definition:

Definition 4. *Given a graph $G = (V, E)$ the 2-core of G, denoted by $K_2(G)$, is the unique subgraph $K_2(G) = (V, E')$, where E' is the maximum (with respect to set-inclusion) subset of E so that with respect to $K_2(G)$ all vertices in V have degree either zero (isolated vertices) or degree at least 2.*

By our definition, the 2-core results by edge-deletions only (and no change in the set of vertices) and the resulting graph has either isolated vertices or vertices of degree at least 2 (retaining throughout our analysis the same set of vertices avoids unnecessary technical complications).

It immediately follows by well known results that $K_2(G)$ can be obtained from G by recursively deleting one at a time and in any order edges that are incident on vertices of degree 1. By assumption, when we delete the edge incident on a vertex v of degree 1, v remains in the graph (but becomes isolated).

Consider now the uniform probability space of graphs such that the number of vertices of degree i is $(e^{-d}(d^i/i!)+o(1))n$, i.e. graphs whose degree sequence is Poisson distributed with mean d. It is known that a.a.a. graphs in $\mathcal{G}(n; d)$ have a Poisson distributed degree sequence with mean d.

In general, let $\mathcal{G}(n; \langle d_i\rangle_{i=0,\ldots,m})$ be the uniform probability space of graphs with n vertices and scaled degree sequence $\langle d_i\rangle_{i=0,\ldots,m}$ (i.e. the number of vertices of degree i is $(d_i+o(1))n$; d_i are assumed to be independent of n). For such graphs we use the configuration model which models random pairings of copies of the vertices, the number of copies of each vertex being equal to its degree. It is well known that results that hold for a.a.a. such pairings in the configuration model, also hold for a.a.a. uniformly distributed simple graphs with the same degree sequence.

It is known that if G is random with a Poisson degree sequence, then $K_2(G)$ is random in $\mathcal{G}(n; \langle d_i\rangle_{i=0,\ldots,m})$ for the same n and a new degree sequence, for which $d_1 = 0$. To compute the new degree sequence, we follow the technique of differential equations of Wormald [21]: we write differential equations that give the dynamics each d_i during the execution of the edge-deletion process. The solution of the differential equations give the final values of d_i within $o(1)$, for $i = 0, \ldots, n-1$. These values hold for a.a.a. input graphs. Our analysis closely follows the methodology given by Mitzenmacher [17] for the case of deletion of pure literals from 3-SAT formulas. We symbolically solve the resulting system of differential equations. Actually, the system of differential equations in our case is easier to obtain and solve, as we do not have the complication of handling the negation of a deleted literal. For reasons of space, we avoid the details (that follow standard techniques) and only give the final result without proof:

Theorem 2. *The number of vertices of degree $i = 0, \ldots, m$ of the 2-core $K_2(G)$ of a random graph G in $\mathcal{G}(n, m = \lfloor \frac{d}{2} n \rfloor)$ is a.a.a. $(d_i + o(1))n$, where*

$$d_i = \begin{cases} -\frac{1}{d} W_d (d + 1 + W_d) \text{ when } i = 0, \\ 0 \text{ when } i = 1, \\ -\frac{W_d}{d} \frac{(d+W_d)^i}{i!} \text{ when } i \geq 2, \end{cases} \tag{1}$$

and where W_d is Lambert$W(-de^{-d})$, i.e. the value of the principal branch of Lambert's W-function at $-de^{-d}$. Also the number of deleted edges during the edge deletion process that yields the 2-core a.a.a. is $\left(-W_d - \frac{W_d^2}{2d} + o(1)\right) n$. Finally, a property holds a.a.a. for $K_2(G)$ iff it holds a.a.a. for a random graph conditional its degree sequence is as described in Equation (1) above.

It can be easily seen that the number of the edges that are deleted to yield the 2-core are part of any maximum size cut. Therefore, the size of the max cut of G can be obtained from the size of the max cut of $K_2(G)$ by adding to the latter the number $\left(-W_d - \frac{W_d^2}{2d} + o(1)\right) n$. It easily follows that:

Proposition 2. *If the functions* $\mathrm{ub}(d)$ *and* $\mathrm{lb}(d)$ *give scaled (with respect to the number of edges m) upper and lower, respectively, bounds for the size of the max cut of a random graph conditional its degree sequence is that of Theorem 2, then the functions* $\mathrm{ub}(d) + 2\left(-W_d - \frac{W_d^2}{2d}\right)/d$ *and* $\mathrm{lb}(d) + 2\left(-W_d - \frac{W_d^2}{2d}\right)/d$ *give scaled (with respect to the number of edges m) upper and lower bounds, respectively, for the size of the maximum cut of a random graph in $\mathcal{G}(n, m = \lfloor \frac{d}{2} n \rfloor)$.*

The previous proposition allows us to work with a random graph conditional its degree sequence is as in Theorem 2.

4 The Upper Bound

For a random graph in $\mathcal{G}(n, m = \lfloor \frac{d}{2} n \rfloor)$, a simple application of the first moment method gives that the maximum cut is no more than $\left(\frac{1}{2} + \sqrt{\frac{\ln 2}{d}}\right) \frac{d}{2} n$ for a.a.a. input instances with average degree d, for $d \geq 4 \ln 2$ [4]. This bound is established by estimating the probability of existence of a cut of a given size z by the expectation of the number of cuts of size z. However, first moment estimations are in general, and in this particular case as well, rather gross.

Another well known approach to the question of finding an upper bound for the optimum cut is by semidefinite relaxation of the problem [12]. However, it is in general difficult to estimate the average-case (or typical-case, i.e. a.a.a.-instances-case) output of a semidefinite program. A related result can be found in [6, Theorem 4], which however gives an estimation of the SDP upper bound of MAX-CUT in terms of an unspecified constant only. An earlier bound was obtained by Linear Programming relaxation [1], but with respect to sparse graphs it is shown in [19] that the upper bound obtained by an LP relaxation of MAX-CUT is a.a.a. at most the total number of edges, i.e. no information is obtained.

So we have to resort to other means in order to compute a better bound suited for typical-case considerations. We compute the expected number of *majority* cuts of given size z for a random graph conditional its degree sequence is as in Theorem 2.

Definition 5. *A cut is called a majority cut if (i) at least half of the edges incident on any vertex are bichromatic (i.e., they connect vertices in different parts of the cut) and (ii) any vertex of even degree whose exactly half of its incident edges are bichromatic is necessarily colored red (i.e., it belongs to a prescribed part of the cut).*

Theorem 3. *If a cut of size z exists then also a majority cut of size at least z exists.*

Proof. Given a cut which is not necessarily a majority cut move —one at a time and recursively— vertices that violate any of the two conditions of Definition 5 to the other part of the cut. In any such move, the cut size either remains constant or strictly increases. Also, the process cannot continue indefinitely, as at each move either (i) the cut size increases strictly (when we move a vertex with a minority of bichromatic incident edges), or alternatively (ii) the cut size remains constant but the cardinality of the vertices colored red strictly increases in comparison to its immediately previous value (when we move to the red color a vertex with equal number of bichromatic and monochromatic incident edges). To prove more formally that the process does not continue indefinitely, introduce as the *potential* of a cut the pair of numbers (c, r), where c is the current size of the cut and r is the current cardinality of red vertices, order the set of these pairs lexicographically and observe that each move of a violating vertex to the other part drives the cut to a strictly higher potential, because each move either strictly increases c, or keeps c constant and strictly increases r (in comparison to its previous value). Therefore there must be a stopping time. □

Let $\mathcal{G}(n; d, 2\text{-core})$ denote the uniform probability space of the 2-core of a random graph in $\mathcal{G}(n, m = \lfloor \frac{d}{2} n \rfloor)$ (see Theorem 2 and Proposition 2 of Section 3). In the sequel, let G be a random graph in $\mathcal{G}(n; d, 2\text{-core})$.

Let $\mathcal{C}_\zeta(G)$ be the class of all majority cuts of G with cut size *at least* ζm, where ζ is a real in $[0, 1]$ and $\zeta m = \zeta \lfloor (d/2) n \rfloor$ is an integer in $\{0, \ldots, m\}$. We will compute an a.a.a. scaled upper bound $\mathrm{ub}(d)$ to the values of ζ for which $\mathcal{C}_\zeta(G) \neq \emptyset$ (which by Theorem 3 is also an a.a.a. scaled upper bound to the maximum cut size of G) by finding a minimum value of ζ such that:

$$\lim_n \mathbf{Pr}[|\mathcal{C}_\zeta(G)| > 0] = 0 \tag{2}$$

Towards this end, first observe that the following Markov-type inequality holds:

$$\mathbf{Pr}[|\mathcal{C}_\zeta(G)| > 0] \leq \mathbf{Ex}(|\mathcal{C}_\zeta(G)|). \tag{3}$$

Therefore, to find a minimum ζ for which Equation 2 holds, it is sufficient to find a minimum ζ for which

$$\lim_n \mathbf{Ex}(|\mathcal{C}_\zeta(G)|) = 0. \tag{4}$$

Let now $\mathcal{E}(b_{00}, b_{11}, b_{01})$ be the expected number of majority cuts whose edges connecting two red (respectively, blue, of different color) vertices have cardinality *exactly* $b_{00}n$ (respectively, $b_{11}n, b_{01}n$), where b_{00}, b_{11}, b_{01} belong to the interval $[0, 1]$ and sum to the scaled number of edges $d/2$. It is easy to see that Equation (4) holds iff the following is true:

$$\lim_{n} \left(\max_{\zeta m \leq b_{01}n, b_{00}, b_{11}} \{\mathcal{E}(b_{00}, b_{11}, b_{01})\} \right) = 0. \tag{5}$$

The analytic computation of $\mathcal{E}(b_{00}, b_{11}, b_{01})$ and the computer-aided numerical calculation of the smallest ζ for which Equation (5) holds follow techniques previously used in [10] (see also [9]). Details are omitted for reasons of space.

In Figure 1 we indicatively plot $\mathrm{ub}(d)$ for values of $d \in [1, 20]$, juxtaposing it with the plot of the scaled with respect to m upper bound $\frac{1}{2} + \sqrt{\frac{\ln 2}{d}}$ obtained in [4] by the simple first moment method.

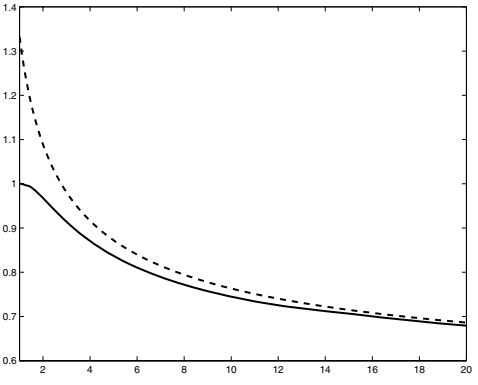


Fig. 1. The upper bound $\mathrm{ub}(d)$ given in Section 4 (solid line) versus the upper bound $\frac{1}{2} + \sqrt{\frac{\ln 2}{d}}$ given in [4] (dashed line) for values of average degree $d \in [1, 20]$

5 The Algorithmic Lower Bound

In this section we describe an algorithm $\mathcal{A}$ that on input a the 2-core $K_2(G)$ of a random graph G in $\mathcal{G}(n; d)$ outputs a coloring C of the vertices of $K_2(G)$ with one of the colors in $\{R, B\}$ (i.e. C is a cut). We remind the reader that by Theorem 2, $K_2(G)$ can be assumed to be random conditional its degree sequence is as in Equation (1). Let $|\mathcal{A}(K_2(G))|$ be the size of the cut C, i.e. the number of its bichromatic edges. From $|\mathcal{A}(K_2(G))|$, we can then compute a scaled lower bound of the max cut of the original graph G by Proposition 2.

The algorithm $\mathcal{A}$ colors the vertices of $K_2(G)$ one at a step. Let $d(v)$ be the degree of the vertex v in $K_2(G)$. At any step t of the algorithm, let U^t be the set

of yet uncolored vertices of $K_2(G)$. For $v \in U^t$, let $d_R^t(v)$ ($d_B^t(v)$, respectively) be the number of vertices that are neighbors of v and are already colored with R (B, respectively). Also let $d_U^t(v) = d(v) - d_R^t(v) - d_B^t(v)$, i.e. $d_U^t(v)$ is the number of neighbors of v in $K_2(G)$ that are yet uncolored. Finally let the *discrepancy* $\Delta^t(v)$ of a vertex $v \in U^t$ be $|d_R^t(v) - d_B^t(v)|$. The algorithm $\mathcal{A}$ at any step t first locates the vertices $v \in U^{t-1}$ that have the largest discrepancy $\Delta^{t-1}(v)$ and chooses among them one with the lowest $d_U^{t-1}(v)$. It then assigns to v the color R if $d_B^{t-1}(v) \geq d_R^{t-1}(v)$ and B otherwise. Intuitively, $\mathcal{A}$ at any step greedily maximizes the difference of the number of edges to be placed in the cut from the number of edges to remain out of it. At the same time, it minimizes the impact of each color assignment to future assignments.

The algorithm $\mathcal{A}$ is described in pseudo-code in Algorithm 5. Its analysis is based on the method of differential equations. The equations give a lower bound on the size of the maximum cut of $K_2(G)$. They are analytically obtained and numerically solved (details are omitted for reasons of space). In Figure 2, we give a plot of the final value of $\mathrm{lb}(d)$ (i.e. the value obtained after applying Proposition (2)) for $d \in [1, 20]$ compared with the values of the algorithms in Coja-Oghlan et al. [6] and Coppersmith et al. [7]. To corroborate the results obtained by numerically solving the analytically derived differential equations, we performed simulation experiments. The simulations gave, as expected, the same values for $\mathrm{lb}(d)$ as the numerical solutions of the differential equations. In Table 1 we juxtapose the simulation results with the results obtained from the differential equations, for certain indicative values of d.

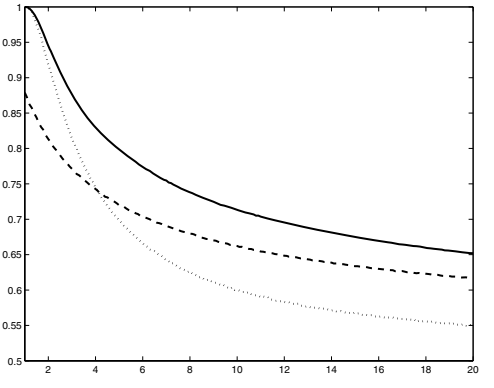


Fig. 2. Our values of the lower bound $lb(d)$ (solid line), obtained by the numerical solution of differential equations (and corroborated by simulation experiments), juxtaposed with the corresponding values obtained by simulating the algorithms in Coja-Oghlan et al. [6] (dashed line) and Coppersmith et al. [7] (dotted line), for values of average degree $d \in [1, 20]$

Algorithm. $\mathcal{A}(K_2(G) = (V_{K_2}, E_{K_2}), C)$
$t = 0$; $U^0 = V_{K_2}$; /* Initialize the set of yet uncolored vertices of $K_2(G)$ */
for all $v \in V_{K_2}$ **do** /* Initialize the number of neighbors of each vertex v of $K_2(G)$ */
$d_U^0(v) = d(v)$; $d_R^0(v) = 0$; $d_B^0(v) = 0$;
end for
while $U^t \neq \emptyset$ **do** /* while there are uncolored vertices */
$t = t + 1$;
Locate all vertices $v \in U^{t-1}$ having the largest discrepancy $\Delta^{t-1}(v)$;
Among them, arbitrarily choose a vertex v with the lowest $d_U^{t-1}(v)$;
if $d_B^{t-1}(v) \geq d_R^{t-1}(v)$ **then**
$C[v] = R$; /* Assign color R to v */
/* Update the number of colored R and yet uncolored neighbors of each neighbor u of v */
for each vertex u adjacent to v **do**
$d_R^t(u) = d_R^{t-1}(u) + 1$;
$d_U^t(u) = d_U^{t-1}(u) - 1$;
end for
else
$C[v] = B$; /* Assign color B to v */
/* Update the number of colored B and yet uncolored neighbors of each neighbor u of v */
for each vertex u adjacent to v **do**
$d_B^t(u) = d_B^{t-1}(u) + 1$;
$d_U^t(u) = d_U^{t-1}(u) - 1$;
end for
end if
$U^t = U^{t-1} \setminus \{v\}$; /* Update the set of yet uncolored vertices of $K_2(G)$ */
end while

Algorithm 1. Algorithm $\mathcal{A}$ takes as input the 2-core $K_2(G)$ of a random graph $G = (V, E)$ and returns a coloring C of its vertices

6 Conclusion and Discussion

Putting together the computations of the previous sections, we reach the conclusion that for every average degree $d > 0$, a.a.a. $\frac{\text{lb}(d)}{\text{ub}(d)} > 0.952$. Therefore our main result, Corollary 1, has been proved. In Figure 3 we give a plot of the ratio $\text{ub}(d)/\text{lb}(d)$ for various values of d, especially close to the average densities where the ratio approaches (from above) the Håstad threshold. Theoretically it is conceivable that there might exist a deterministic algorithm that a.a.a. computes exactly the maximum cut size of a random graph or, more realistically, offers a Polynomial Time Approximation Scheme to it (PTAS). We believe that for at least certain values of d there is no such PTAS valid a.a.a. However it is conceivable that for every given $\epsilon > 0$, one might come with an algorithm that yields an a.a.a. approximation scheme of ratio ϵ. Finally, when d is not constant, but approaches infinity with n (dense graphs), then it is known that a.a.a. $(1/2)|E| < |E|((1/2) + o(1))$ [18].

Table 1. Simulation experiment results (SE) versus numerical solution of the differential equations (DE) for indicative values of average degree d

d	2.0	3.5	4.0	4.5	5.0	6.0	8.0	10.0	12.0	14.0
SE	0.945	0.850	0.829	0.813	0.798	0.773	0.738	0.713	0.696	0.681
DE	0.945	0.851	0.830	0.813	0.798	0.774	0.738	0.713	0.695	0.681

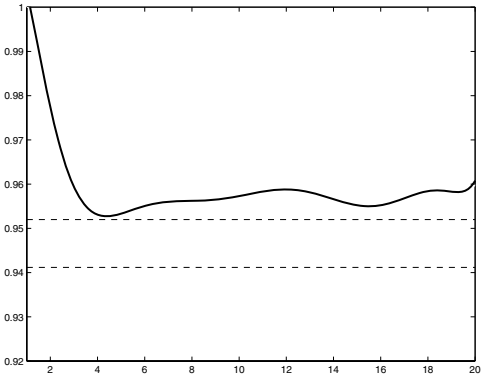


Fig. 3. The approximation ratio $\mathrm{lb}(d)/\mathrm{ub}(d)$, for values of average degree $d \in [1, 20]$. The lower dashed line corresponds to Håstad inapproximability threshold 16/17, while the upper dashed line to our approximation ratio 0.952.

We believe that these results can also be extended to the case of d-regular graphs. We are currently working on this. Also these results extend to k-Max-Cut for $k > 2$.

Acknowledgment

We thank Giorgos Kounenis for providing us with the symbolic solution of the system of differential equations that give the degree sequence of the 2-core.

References

1. F. Barahona, M. Grotschel, M. Junger, and G. Reinelt. An application of combinatorial optimization to statistical physics and circuit layout design. *Operations Research*, 36:493–513, 1988.
2. M. Beis, W. Duckworth, and M. Zito. Packing edges in random regular graphs. In K. Diks and W. Rytter, editors, *MFCS*, volume 2420 of *Lecture Notes in Computer Science*, pages 118–130. Springer, 2002.
3. M. Beis, W. Duckworth, and M. Zito. Large k-separated matchings of random regular graphs. In V. Estivill-Castro, editor, *ACSC*, volume 38 of *CRPIT*, pages 175–182. Australian Computer Society, 2005.

4. A. Bertoni, P. Campadelli, and R. Posenato. Un upper bound for the maximum cut mean value. In *Proceedings of the 23rd International Workshop on Graph-Theoretic Concepts in Computer Science (WG97)*, volume 1335, pages 78–84. Lecture Notes in Computer Science, Springer, 1997.
5. M. Charikar, K. Makarychev, and Y. Makarychev. Near-optimal algorithms for unique games. In *Proceedings of the 38th ACM Symposium on Theory of Computing Seattle, Washington, USA*, 2006.
6. A. Coja-Oghlan, C. Moore, and V. Sanwalani. Max k-cut and approximating the chromatic number of random graphs. In J.C.M. Baeten, J.K. Lenstra, J. Parrow, and G.J. Woeginger, editors, *ICALP*, volume 2719 of *Lecture Notes in Computer Science*, pages 200–211. Springer, 2003.
7. D. Coppersmith, D. Gamarnik, M.T. Hajiaghayi, and G.B. Sorkin. Random max sat, random max cut, and their phase transitions. *Random Struct. Algorithms*, 24(4):502–545, 2004.
8. W. Fernandez de la Vega and M. Karpinski. 9/8-approximation algorithm for random max-3sat. *Electronic Colloquium on Computational Complexity (ECCC)*, (070), 2002.
9. J. Díaz, G. Grammatikopoulos, A.C. Kaporis, L.M. Kirousis, X. Pérez, and D.G. Sotiropoulos. 5-regular graphs are 3-colorable with positive probability. In G.S. Brodal and S. Leonardi, editors, *ESA*, volume 3669 of *Lecture Notes in Computer Science*, pages 215–225. Springer, 2005.
10. O. Dubois and J. Mandler. On the non-3-colourability of random graphs. *ArXiv Mathematics e-prints*, September 2002.
11. A.M. Frieze and C. McDiarmid. Algorithmic theory of random graphs. *Random Struct. Algorithms*, 10(1-2):5–42, 1997.
12. M. Goemans and D. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, 42:1115–1145, 1995.
13. J. Håstad. Some optimal inapproximability results. *Journal of the ACM*, 48:798–869, 2001.
14. Y. Interian. Approximation algorithm for random MAX-k-SAT. In *International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, 2004.
15. V. Kalapala and C. Moore. Max-cut on sparse random graphs. *TR-CS-2002-24, University of New Mexico Department of Computer Science*, 2002.
16. S. Khot. Hardness of approximating the shortest vector problem in lattices. In *FOCS*, pages 126–135. IEEE Computer Society, 2004.
17. M. Mitzenmacher. Tight thresholds for the pure literal rule. Technical report, Digital Equipment Corporation, 1997. Available at: www.research.compaq.com/SRC/.
18. Ng. Van Ngoc and Zs. Tuza. Linear-time algorithms for the max cut problem. *Combinatorics, Probability & Computing*, 2:201–210, 1993.
19. S. Poljak and Z. Tuza. The expected relative error of the polyhedral approximation of the max-cut problem. *Operations Research Letters*, 16:191–198, 1994.
20. L. Trevisan, G. Sorkin, M. Sudan, and D. Williamson. Gadgets, approximation, and linear programming. *SIAM Journal on Computing*, 29(6):2074–2097, 2000.
21. N.C. Wormald. Differential equations for random processes and random graphs. *The Annals of Applied Probability*, 5(4):1217–1235, 1995.

Enumerating Spanning and Connected Subsets in Graphs and Matroids

L. Khachiyan[1,⋆], E. Boros[2], K. Borys[2], K. Elbassioni[3], V. Gurvich[2], and K. Makino[4]

[1] Department of Computer Science, Rutgers University, 110 Frelinghuysen Road, Piscataway NJ 08854
[2] RUTCOR, Rutgers University, 640 Bartholomew Road, Piscataway NJ 08854
{boros, kborys, gurvich}@rutcor.rutgers.edu
[3] Max-Planck-Institut für Informatik, Saarbrücken, Germany
elbassio@mpi-sb.mpg.de
[4] Division of Mathematical Science for Social Systems, Graduate School of Engineering Science, Osaka University, Toyonaka, Osaka, 560-8531, Japan
makino@sys.es.osaka-u.ac.jp

Abstract. We show that enumerating all minimal spanning and connected subsets of a given matroid can be solved in incremental quasi-polynomial time. In the special case of graphical matroids, we improve this complexity bound by showing that all minimal 2-vertex connected edge subsets of a given graph can be generated in incremental polynomial time.

1 Introduction

The level of connectivity in communications and computer networks is an important parameter influencing the reliability of the service such networks provide. The problem of computing network reliability, that is calculating the probability that the network is able to provide its services without interruption, assumes the enumeration of minimal subsets of links in the network which guarantee the required level of connectivity [5, 15].

In the simplest case the connectivity of an undirected graph is required. In this case minimal working subsets are the spanning trees, and reliability computations call for the enumeration of all spanning trees of the given graph. In case of a (directed) communication network, minimal working subnetworks are determined by subsets of the arcs which guarantee strong connectivity. Both minimal spanning trees and minimal strongly connected subgraphs can be efficiently enumerated [1, 4, 13].

Practical applications frequently demand higher levels of connectivity, resulting in higher reliability. Numerous research articles consider the problem of increasing efficiently the connectivity of a given (directed) graph, see e.g. [2, 8].

⋆ Our friend and co-author, Leonid Khachiyan passed away with tragic suddenness, while we were working on the final version of this paper.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 444–455, 2006.

Determining the reliability of such highly connected networks requires the enumeration of all minimal edge (arc) sets $F \subseteq E$ of a given (directed) graph $G = (V, E)$ which guarantee the required level of connectivity of the subgraph (V, F).

In this paper we consider such enumeration problems, corresponding to the next levels of connectivity. An undirected graph $G = (V, E)$ is called 2-*vertex connected* if between any pair $u, v \in V$ of its vertices there are at least two paths connecting u and v and having no other vertex in common. Obviously, adding edges to a graph can only increase its connectivity. In other words, the property that for a subset $X \subseteq E$ the subgraph (V, X) is 2-vertex connected is a monotone property, i.e., if $X \subseteq X' \subseteq E$, and (V, X) is 2-vertex connected, then (V, X') must also be 2-vertex connected. Thus, determining the reliability of such a graph, that is the probability that it remains 2-vertex connected if its edges are deleted (fail) according to some probability distribution, requires the enumeration of all minimal 2-vertex connected subgraphs of G:

> **Minimal 2-Vertex-Connected Spanning Subgraphs:** *Given a 2-vertex connected undirected graph $G = (V, E)$, enumerate all minimal edge sets $X \subseteq E$ such that $G' = (V, X)$ is still 2-vertex connected.*

Undirected graphs can be viewed as special cases of matroids (so called graphical or cycle matroids), and thus the above enumeration problems have a natural generalization for matroids. In our presentation we follow standard terminology of matroid theory (see e.g., [12] or [16]). Given a matroid M on ground set E, a subset $T \subseteq E$ is called *connected* if for every pair of distinct elements x, y of T there is a circuit C of M such that $T \supseteq C \supseteq \{x, y\}$. It is well-known that connectedness defines an equivalence relation on E, whose equivalence classes are called the *connected components* of M. Let us also recall that a subset $X \subseteq E$ is said to *span* the matroid M if $r(X) = r(E)$, where $r : E \to \mathbb{Z}_+$ denotes the rank function of M.

Note that in the cycle matroid of a graph $G = (V, E)$ spanning trees are the bases. Thus, the problem of enumerating all bases of a matroid includes as a special case the spanning tree enumeration problem. Note also that edge subsets $X \subseteq E$ for which (V, X) are 2-vertex connected are exactly the spanning and connected subsets in the cycle matroid of G (see e.g., [16, Theorem 3 on page 70]). Let us add that the family of spanning and connected subsets in a matroid form a monotone system (see Lemma 3), while connected subsets of a matroid may not form a monotone system. The following enumeration problem generalizes naturally the problem of enumerating minimal 2-vertex-connected spanning subgraphs:

> **Minimal Spanning and Connected Subsets in Matroids:** *Given a connected matroid M on ground set E, generate all minimal spanning and connected subsets of E.*

1.1 Main Results

It is easy to see that in enumeration problems, such as the ones mentioned above, the size of the output may be exponential in terms of the input size. For such

problems the efficiency of the enumeration method is measured both in the input and output sizes (see e.g., [10, 13, 15]). In particular, the enumeration procedure is said to run in *incremental (quasi-) polynomial*[1] *time*, if generating k elements of the target hypergraph (or generating all if it has less than k elements) can be done in (quasi-) polynomial time in the size of the input and k, for an arbitrary integer k.

Note that all of the above mentioned enumeration problems involve monotone systems. Among such monotone generation problems perhaps the most widely known is the so called *hypergraph transversal* problem (or equivalently, *monotone Boolean dualization*) which consists of generating all minimal transversals of a given hypergraph. This problem has numerous applications in several different areas (see e.g., [6]).

Our first result shows that the problem of generating minimal spanning and connected subsets of a matroid generalizes the hypergraph transversal problem:

Theorem 1. *The problem of enumerating all minimal spanning and connected subsets of a matroid includes, as a special case, the hypergraph transversal problem.*

This theorem implies that generating minimal spanning and connected subsets of a matroid is at least as hard as the hypergraph transversal problem, for which the most efficient currently known algorithm is incrementally quasi-polynomial [7]. Our next result shows that minimal spanning and connected subsets in a matroid can also be generated in incremental quasi-polynomial time.

Theorem 2. *All minimal spanning and connected subsets in a matroid can be generated in incremental quasi-polynomial time.*

As we noted above, the problem of generating minimal spanning and connected subsets in a graphical matroid coincides with the problem of generating minimal 2-vertex-connected subgraphs of the underlying undirected graph. Our third result shows that in this special case the problem can be solved more efficiently:

Theorem 3. *All minimal 2-vertex connected spanning subgraphs of a given graph can be generated in incremental polynomial time.*

The remainder of the paper is organized as follows. We prove Theorems 1 and 2 in Section 2, and the proof of Theorem 3 is included in Section 3.

2 Minimal Spanning and Connected Subsets in Matroids

2.1 Proof of Theorem 1

Let $\mathcal{H}$ be a hypergraph on n vertices consisting of $m = |\mathcal{H}|$ hyperedges. We denote by $\mathbf{v}_1, \ldots, \mathbf{v}_n$ the column vectors of the edge-vertex incidence matrix of $\mathcal{H}$.

[1] A function $f(x)$ is called quasi-polynomial if $f(x) = O(2^{polylog(x)})$.

We shall associate to $\mathcal{H}$ a binary matroid $M = M_{\mathcal{H}}$, defined by $m + 2n + 2$ binary vectors of dimension $2m + 2$. For this, let us introduce $\mathbf{o} = (0, ..., 0)$ denoting the zero vector of dimension m, and let $\mathbf{e}_i$ denote the ith unit vector of dimension m, for $i = 1, ..., m$. We shall define the vectors of $M_{\mathcal{H}}$ by concatenations from the above vectors, as follows: Let $\mu(\mathbf{v}_j) = (\mathbf{v}_j, \mathbf{o}, 1, 0)$ for $j = 1, ..., n$, let $\mathbf{a}_i = (\mathbf{e}_i, \mathbf{e}_i, 0, 0)$ and $\mathbf{b}_i = (\mathbf{o}, \mathbf{e}_i, 0, 0)$ for $i = 1, ..., m$, and finally let $\mathbf{c}_1 = (\mathbf{o}, \mathbf{o}, 1, 1)$ and $\mathbf{c}_2 = (\mathbf{o}, \mathbf{o}, 0, 1)$.

We group the above defined vectors into four groups: $H = \{\mu(\mathbf{v}_j) | j = 1, ..., n\}$, $A = \{\mathbf{a}_i | i = 1, ..., m\}$, $B = \{\mathbf{b}_i | i = 1, ..., m\}$ and $C = \{\mathbf{c}_1, \mathbf{c}_2\}$, and finally we consider the binary matroid $M = M_{\mathcal{H}}$ on the ground set $E = H \cup A \cup B \cup C$. For simplicity, we re-interpret $\mathcal{H}$ as a family of subsets of H.

Example 1. Consider the hypergraph $\mathcal{H}$ defined by the incidence matrix

$$(\mathbf{v}_1, \ldots, \mathbf{v}_5) = \begin{pmatrix} 1\,0\,0\,1\,0 \\ 0\,1\,0\,1\,1 \\ 0\,1\,1\,0\,0 \\ 0\,0\,1\,0\,1 \end{pmatrix}$$

Then the binary matroid $M_{\mathcal{H}}$ on the ground set $H \cup A \cup B \cup C$ is represented by a matrix

$$\left(\begin{array}{ccccc|cccc|cc} 1\,0\,0\,1\,0 & 1\,0\,0\,0 & 0\,0\,0\,0 & 0\,0 \\ 0\,1\,0\,1\,1 & 0\,1\,0\,0 & 0\,0\,0\,0 & 0\,0 \\ 0\,1\,1\,0\,0 & 0\,0\,1\,0 & 0\,0\,0\,0 & 0\,0 \\ 0\,0\,1\,0\,1 & 0\,0\,0\,1 & 0\,0\,0\,0 & 0\,0 \\ \hline 0\,0\,0\,0\,0 & 1\,0\,0\,0 & 1\,0\,0\,0 & 0\,0 \\ 0\,0\,0\,0\,0 & 0\,1\,0\,0 & 0\,1\,0\,0 & 0\,0 \\ 0\,0\,0\,0\,0 & 0\,0\,1\,0 & 0\,0\,1\,0 & 0\,0 \\ 0\,0\,0\,0\,0 & 0\,0\,0\,1 & 0\,0\,0\,1 & 0\,0 \\ \hline 1\,1\,1\,1\,1 & 0\,0\,0\,0 & 0\,0\,0\,0 & 1\,0 \\ 0\,0\,0\,0\,0 & 0\,0\,0\,0 & 0\,0\,0\,0 & 1\,1 \end{array}\right)$$

To complete the proof of Theorem 1, we show two lemmas which then readily imply the statement of the theorem. For this proofs, let us recall (e.g., from [16]) that in a matroid M on ground set E with rank function r a subset $X \subseteq E$ is spanning and connected if and only if for every nontrivial partition $X = Y \cup Z$ (i.e., for which $|Y| \geq 1$ and $|Z| \geq 1$) we have $r(Y) + r(Z) \geq r(M) + 1$.

Lemma 1. *Let X be a spanning and connected subset in M. Then $A \cup B \cup C \subseteq X$ and $X \cap H$ is a transversal of $\mathcal{H}$.*

Proof. First we show that for each $i = 1, \ldots, 2m + 2$ at least two vectors of X have their ith coordinates equal to 1. Indeed, since X is spanning and the matrix representing M has full row rank, there is at least one vector in X whose ith coordinate is 1. Suppose that there is exactly one such vector $\mathbf{x} \in X$. Then we consider the partition of X into $Y = \{\mathbf{x}\}$ and $Z = X \setminus \{\mathbf{x}\}$. For this partition we have $r(Y) = 1$, and $r(Z) < r(M)$ since all vectors of Z have their ith coordinates

equal to 0. Consequently, $r(Y)+r(Z) \leq r(M)$, contradicting the assumption that X is spanning and connected.

The above implies that X must contain all vectors of A, B and C, in order to have two 1's in rows from $m+1$ to $2m$ and in row $2m+2$. In order to contain two 1's in the first m rows, $H \cap X$ must form a transversal of $\mathcal{H}$. □

Lemma 2. *If X is a transversal of $\mathcal{H}$, then $X \cup A \cup B \cup C$ is a connected and spanning subset in M.*

Proof. First note that $r(M) = 2m+2$, and observe that $X \cup A \cup B \cup C$ is spanning, since $r(A \cup B \cup C) = 2m+2$.

To prove the statement, we show that $r(Y) + r(Z) \geq r(M) + 1 = 2m+3$ for every partition $Y \cup Z = X$ for which $|Y| \geq 1$ and $|Z| \geq 1$.

Suppose that all vectors of $A \cup B \cup C$ belong to Y. Then $r(Y) = 2m+2$ and $r(Z) \geq 1$, since Z is nonempty. For all other nontrivial partitions of X the vectors of $A \cup B \cup C$ must be split between Y and Z.

Without loss of generality assume that Y contains k vectors of $A \cup B \cup C$ including $\mathbf{c}_1$, where $1 \leq k \leq 2m+1$. Consequently $r(Y) \geq k$ and $r(Z) \geq 2m+2-k$. Let $I_i \subseteq \{1, \ldots, m\}$ denote the set of coordinates of $\mathbf{v}_i$ equal to 1. Observe that $\mu(\mathbf{v}_i) = \mathbf{c}_1 + \mathbf{c}_2 + \sum_{j \in I_i} (\mathbf{a}_j + \mathbf{b}_j)$ is the only combination of vectors in $A \cup B \cup C$ producing $\mu(\mathbf{v}_i)$. Depending whether Z contains a vector $\mu(\mathbf{v}_i) \in X$, or not, we have two cases:

Case 1: Z contain at least one vector of X. Since vectors of X cannot be obtained without $\mathbf{c}_1$, we must have $r(Z) \geq 2m+3-k$ implying thus $r(Y)+r(Z) \geq 2m+3$.
Case 2: Y contains all vectors of X. Since X is a transversal of $\mathcal{H}$, we have $\bigcup_{\mu(\mathbf{v}_i) \in X} I_i = \{1, \ldots, m\}$. As Y does not contain all vectors of $A \cup B \cup C$, there is a vector in X which cannot be obtained as a combination of vectors in Y. Thus $r(Y) \geq k+1$ and $r(Y) + r(Z) \geq 2m+3$ follows again. □

The statement of the theorem follows from Lemmas 1 and 2.

2.2 Proof of Theorem 2

Let M be a matroid on ground set S with rank function $r : S \to \mathbb{Z}_+$. We assume that M does not contain loops, i.e. singletons of rank 0.

Let us show first that spanning and connected subsets of a matroid form a monotone family.

Lemma 3. *If $X \subseteq E$ is spanning and connected then for an arbitrary element $f \in E \setminus X$ the set $X \cup f$ is again spanning and connected.*

Proof. Let $X \subseteq E$ be a spanning and connected subset and let $f \in E \setminus X$. Clearly $X \cup f$ is spanning. According to [16], to see that $X \cup f$ is also connected it is enough to show that $r(Y) + r(Z) \geq r(X \cup f) + 1$ holds for an arbitrary partition $Y \cup Z = X \cup f$ with $|Y| \geq 1$, $|Z| \geq 1$. Note that, since X is spanning, $r(X) + 1 = r(X \cup f) + 1$. Without loss of generality assume that $f \in Z$. If $|Z| = 1$ we have $r(Y) + r(Z) = r(X) + r(f) = r(X) + 1$, since we assumed

that all singletons of M have rank 1. In case $|Z| \geq 2$, we have $r(Y) + r(Z) \geq r(Y) + r(Z \setminus f) \geq r(X) + 1$, since $r(Z) \geq r(Z \setminus f)$ and the sets Y and $Z \setminus f$ form a partition of X, with $|Y| \geq 1$, $|Z \setminus f| \geq 1$, completing the proof of our claim. □

To prove Theorem 2, we show that for every matroid M, the family $\mathcal{F}$ of all minimal spanning and connected subsets in M is exactly the set of minimal solutions of a polymatroid inequality with polynomially bounded right hand side. For such inequalities, it is known that the generation of minimal feasible sets can be done in incremental quasi-polynomial time (see Theorem 3 in [3]).

For this end, let us define a set function $f(X)$ on subsets of E by:

$$f(X) = |E|\, r(X) - 1.$$

The *Dilworth truncation of* $f(X)$ is the set function $f_*(X)$ defined as follows:

$$f_*(\emptyset) = 0,$$

$$f_*(X) = \min\{f(X_1) + \ldots + f(X_k) \mid X_1, \ldots, X_k \text{ is a partition of } X\} \text{ for } X \neq \emptyset.$$

Clearly, $f(X)$ is a nondecreasing submodular function. Thus $f_*(X)$ is submodular and can be evaluated in polynomial time using $poly(|E|)$ calls to the membership oracle defining the matroid M (see [11]). We next show that $f_*(X)$ is also nondecreasing, implying that $f_*(X)$ is a polymatroid function.

Lemma 4. *$f_*(X)$ is nondecreasing.*

Proof. We will show that $f_*(X \cup e) \geq f_*(X)$, where $X \subseteq E$, $e \in E \setminus X$. Let $X_1, X_2, \ldots, X_k$ be an optimal partition for $X \cup e$, i.e., $f_*(X \cup e) = f(X_1) + f(X_2) + \ldots + f(X_k)$. Without loss of generality assume that $e \in X_1$. There are two cases:

Case 1: $X_1 \setminus e \neq \emptyset$. Then $X_1 \setminus e, X_2, \ldots, X_k$ is a partition of X. Hence

$$f_*(X) \leq f(X_1 \setminus e) + \sum_{i=2}^{k} f(X_i) \leq f(X_1) + \sum_{i=2}^{k} f(X_i) = f_*(X \cup e),$$

where the last inequality in the chain follows from $f(X_1 \setminus e) = |E| r(X_1 \setminus e) - 1 \leq |E| r(X_1) - 1 = f(X_1)$.

Case 2: $X_1 = \{e\}$. Consider the partition $X_2, \ldots, X_k$ of X, which again gives

$$f_*(X) \leq \sum_{i=2}^{k} f(X_i) \leq f(e) + \sum_{i=2}^{k} f(X_i) = f_*(X \cup e),$$

where the last inequality in the chain follows from the fact that $r(e) = 1$, thus $f(e) = |E| - 1 \geq 0$, for all $e \in X$. □

Consider now the polymatroid inequality

$$f_*(X) \geq |E|\, r(M) - 1.$$

Note that the right hand side of the above inequality is bounded by $|E|^2$. We prove that minimal connected spanning subsets are exactly minimal solutions to the above polymatroid inequality.

Lemma 5. *X is connected in M if and only if $f_*(X) \geq f(X)$.*

Proof. Let X be connected subset in M. Consider a partition $X_1, \ldots, X_k$ of X into at least $k \geq 2$ sets. Since the rank function is submodular and by the definition of connectivity we have $r(A)+r(E \setminus A) > r(E)$ for every proper subset A of E, we obtain $r(X_1) + r(X_2) + \ldots + r(X_k) \geq r(X_1) + r(X_2 \cup \ldots \cup X_k) \geq r(X) + 1$. Hence

$$f(X_1) + f(X_2) + \ldots + f(X_k) \geq |E|\, r(X) + |E| - k > |E|\, r(X) - 1 = f(X).$$

Comparing that with the trivial partition $X = X_1$ for $k = 1$, we conclude that $f_*(X) = f(X)$.

On the other hand, if X is not connected, then we can decompose X into two disjoint sets X_1 and X_2 such that $r(X_1)+r(X_2) = r(X)$. Hence $f(X_1)+f(X_2) = |E|\, r(X) - 2$, and consequently, $f_*(X) < |E|r(X) - 1 = f(X)$. □

Lemma 6. *X is connected and spanning subset in M if and only if $f_*(X) \geq |E|\, r(M) - 1$.*

Proof. If X is connected and spanning, the claim follows from Lemma 5 and the fact that $r(X) = r(M)$.

Conversely, suppose X satisfies $f_*(X) \geq |E|\, r(M) - 1$ and also suppose that X is not spanning. Then since $r(X) < r(M)$ for the trivial partition $X = X_1$, we obtain $f(X_1) = |E|r(X_1)-1 < |E|r(M)-1$, which implies $f_*(X) < |E|r(M)-1$, a contradiction. Thus X must be spanning and by Lemma 5 X must also be connected. □

This completes the proof of Theorem 2.

3 Minimal 2-Connected Spanning Subgraphs

3.1 The $X - e + Y$ Method

In this section we recall a technique from [9], which is a variant of the supergraph approach introduced by [14]. Let E be a finite set, and $\pi : 2^E \to \{0, 1\}$ be a monotone Boolean function, i.e., one for which $X \subseteq Y$ implies $\pi(X) \leq \pi(Y)$. We assume that $\pi(\emptyset) = 0$ and $\pi(E) = 1$. We also assume that an efficient algorithm for evaluating $\pi(X)$ in polynomial time in the size of E is available for every $X \subseteq E$. Let

$$\mathcal{F} = \{X \mid X \subseteq E \text{ is a minimal set satisfying } \pi(X) = 1\}.$$

Our goal is to generate all sets belonging to $\mathcal{F}$.

Let us remark first that for every $X \subseteq E$ for which $\pi(X) = 1$ we can derive a subset $Y \subseteq X$ such that $Y \in \mathcal{F}$, by evaluating π exactly $|X|$ times. This can be accomplished by deleting one-by-one elements of X the removal of which does not change the value of π. To formalize this, we can fix an arbitrary linear order $\prec$ on elements of E, without any loss of generality, and define a mapping $\mu : \{X \subseteq E \mid \pi(X) = 1\} \to \mathcal{F}$ by $\mu(X) = X \setminus Z$, where Z is the lexicographically first subset of X, with respect to $\prec$, such that $\pi(X \setminus Z) = 1$ and $\pi(X \setminus (Z \cup e)) = 0$ for every $e \in X \setminus Z$. Clearly, by trying to delete elements of X in their $\prec$-order, we can compute $\mu(X)$, as we remarked above, by evaluating π exactly $|X|$ times.

We next introduce a directed graph $\mathcal{G} = (\mathcal{F}, \mathcal{E})$ on vertex set $\mathcal{F}$, where the neighborhood $N(X)$ of $X \in \mathcal{F}$ and the family $\mathcal{Y}_{X,e}$ are defined by

$$N(X) = \{\mu((X \setminus e) \cup Y) \mid e \in X, Y \in \mathcal{Y}_{X,e}\}, \quad \text{and}$$

$$\mathcal{Y}_{X,e} = \{Y \mid Y \subseteq E \setminus X \text{ is a minimal set satisfying } \pi((X \setminus e) \cup Y) = 1\}.$$

In other words, for every set $X \in \mathcal{F}$ and for every element $e \in X$ (since $X \in \mathcal{F}$, we have $\pi(X \setminus e) = 0$) we extend $X \setminus e$ in all possible ways to a set $X' = (X \setminus e) \cup Y$ for which $\pi(X') = 1$, and introduce each time a directed arc from X to $\mu(X')$. We call the obtained directed graph $\mathcal{G}$ a *supergraph* of our generation problem.

Proposition 1 ([9]). *The supergraph $\mathcal{G} = (\mathcal{F}, \mathcal{E})$ is strongly connected.* □

Since $\mathcal{G}$ is strongly connected by performing a breadth-first search in $\mathcal{G}$ we can generate all elements of $\mathcal{F}$ as follows:

$Traversal(\mathcal{G})$
Find an initial vertex $X^0 \leftarrow \mu(E)$, and initialize two queues $\mathcal{P} = \mathcal{Q} = \emptyset$.
Perform a breadth-first search of $\mathcal{G}$ starting from X^o:

```
1  output X^0 and insert it to the queue P
2  while P ≠ ∅ do
3     take the first vertex X out of the queue P, and insert it to Q
4     for every e ∈ X do
5        for every Y ∈ Y_{X,e} do
6           compute the neighbor X' ← μ((X \ e) ∪ Y)
7           if X' is not in P ∪ Q then
8                 output X' and insert it to P
9        endfor
10    endfor
11 endwhile
```

Proposition 2 ([9]). *If the sets of $\mathcal{Y}_{X,e}$ can be generated in incremental polynomial time for every $X \in \mathcal{F}$ and $e \in X$, then $Traversal(\mathcal{G})$ generates all elements of $\mathcal{F}$ in incremental polynomial time.* □

3.2 Proof of Theorem 3

We apply the $X - e + Y$ method to the generation of all minimal 2-vertex connected spanning subgraphs of a 2-vertex connected graph $G = (V, E)$.

For $X \subseteq E$, we define a Boolean function π as follows:

$$\pi(X) = \begin{cases} 1, \text{ if (V,X) is 2-vertex connected;} \\ 0, \text{ otherwise.} \end{cases}$$

Then $\mathcal{F} = \{X \mid X \subseteq E \text{ is a minimal set satisfying } \pi(X) = 1\}$ is a family of edge sets of all minimal 2-vertex connected spanning subgraphs of (V, E). For $X \in \mathcal{F}$ and $e \in X$ we define

$$\mathcal{Y}_{X,e} = \{Y \mid Y \subseteq E \setminus X \text{ is a minimal set satisfying } \pi((X \setminus e) \cup Y) = 1\}.$$

Therefore by Proposition 2 we only need to prove that we can generate all elements of $\mathcal{Y}_{X,e}$ in incremental polynomial time. In fact, we show that we can do it, more efficiently, with *polynomial delay*, i.e., in which the generation of the first k elements can be accomplished in time polynomial in the input size and linear in k.

Recall that a maximal connected subgraph without a cutvertex is called a *block*. Thus, every block of a connected graph H is either a maximal 2-vertex connected subgraph, or a bridge (with its ends). Different blocks overlap in at most one vertex, which is a cutvertex of H. Hence, every edge of the graph lies in a unique block.

Let A denote the set of cutvertices of H and let $\mathcal{B}$ denote the set of its blocks. We then have a natural bipartite graph on vertex set $A \cup \mathcal{B}$ in which two vertices $B \in \mathcal{B}$, $a \in A$ are connected if a is a cutvertex of H belonging to B. We call such graph a *block graph* of H. Observe that the block graph of a connected graph is a tree.

Proposition 3. *All elements of $\mathcal{Y}_{X,e}$ can be generated with polynomial delay.*

Proof. Let (V, X) be a minimal 2-vertex connected spanning subgraph of (V, E) (see Figure 1).

First we show that the block graph of $(V, X \setminus e)$ is a path such that endpoints of e belong to its ends. As we observed above the block graph of $(V, X \setminus e)$ is a tree. Suppose it has a leaf B that does not contain an endpoint of e. Let a be a cutvertex of $(V, X \setminus e)$ adjacent to B in the block graph. But removing the vertex a from the 2-vertex connected graph (V, X) disconnects vertices of B from other vertices, a contradiction. Thus the block graph of $(V, X \setminus e)$ has only two leaves, each containing one endpoint of e.

We denote by $B_1, \ldots, B_r$ the blocks of $(V, X \setminus e)$ and by $a_1, \ldots, a_{r-1}$ its cutvertices. Without loss of generality we assume that the block graph of $(V, X \setminus e)$ is a path $B_1 a_1 B_2 \ldots a_{r-1} B_r$ (see Figure 2).

Let $f = uv$ be an edge of $E \setminus X$, such that u belongs to the block B_i and v belongs to B_j, where $i < j$. We define $\alpha(f) = \begin{cases} i, & \text{if } v \in B_i \setminus a_i \ ; \\ i+1, & \text{if } v = a_i, \end{cases}$ and $\beta(f) = \begin{cases} j, & \text{if } v \in B_j \setminus a_{j-1} \ ; \\ j-1, & \text{if } v = a_{j-1}. \end{cases}$

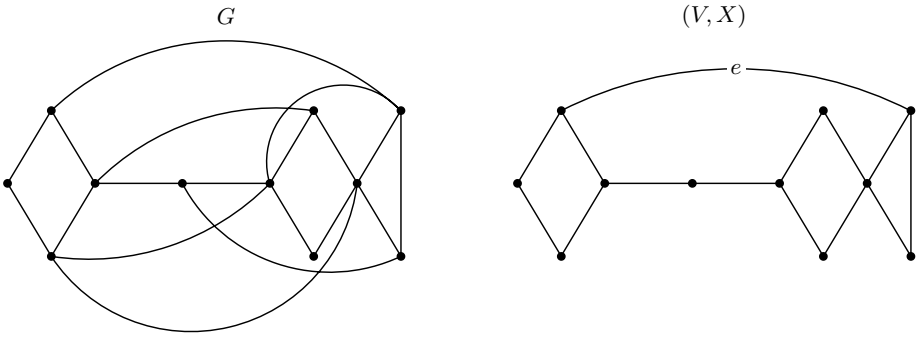


Fig. 1. 2-vertex connected graph $G = (V, E)$ and a minimal 2-connected spanning subgraph (V, X) of G

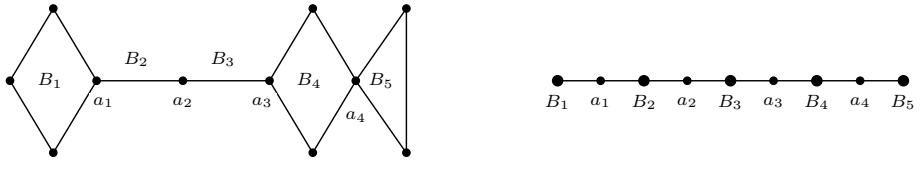


Fig. 2. $(V, X \setminus e)$ and its block graph

Then we construct a directed multigraph D on vertex set $B_1, \ldots, B_r$ whose edge set is defined as follows:

- for each $i = 1, \ldots, r-1$, we add an arc $B_{i+1}B_i$,
- for each edge $f \in E \setminus X$, such that $\alpha(f) < \beta(f)$, we add an arc $B_{\alpha(f)}B_{\beta(f)}$ (see Figure 3).

Now we show that the generation of elements of $\mathcal{Y}_{X,e}$ is equivalent to the generation of minimal directed B_1-B_r paths in D.

For every cutvertex a_k there is an edge $f \in Y$ such that $\alpha(f) \leq k < \beta(f)$. By minimality of Y, edges of $E \setminus X$ whose both endpoints belong to the same block cannot be in Y. We conclude that $Y = \{f_1, \ldots, f_s\}$ such that

$$1 = \alpha(f_1) < \alpha(f_2) \leq \beta(f_1) < \alpha(f_3) \leq \ldots < \alpha(f_s) \leq \beta(f_{s-1}) < \beta(f_s) = r.$$

Thus Y corresponds to a directed path $B_{\alpha(f_1)}\ B_{\beta(f_1)}\ B_{\beta(f_1)-1} \ldots B_{\alpha(f_2)+1}\ B_{\alpha(f_2)}\ B_{\beta(f_2)}\ B_{\beta(f_2)-1} \ldots B_{\alpha(f_3)+1}\ B_{\alpha(f_3)} B_{\beta(f_3)} \ldots B_{\beta(f_s)}$ (see Figure 4).

Since all minimal directed paths between two vertices can be generated via backtracking with polynomial delay [13], Proposition 3 follows. □

This completes the proof of Theorem 3.

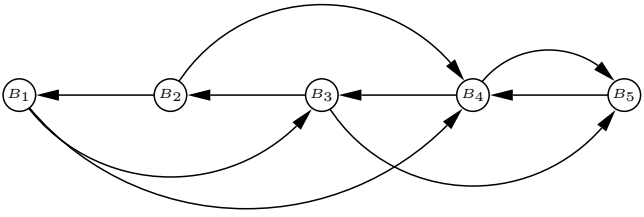


Fig. 3. Directed multigraph D

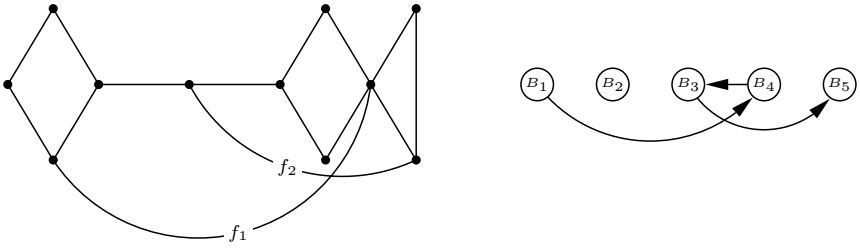


Fig. 4. $Y = \{f_1, f_2\}$ and corresponding directed path $B_1B_4B_3B_5$

References

1. A. Tamura A. Shioura and T. Uno. An optimal algorithm for scanning all spanning trees of undirected graphs. *SIAM Journal on Computing*, 26(3):678–692, 1997.
2. J. Bang-Jensen, H.N. Gabow, T. Jordán, and Z. Szigeti. Edge-connectivity augmentation with partition constraints. *SIAM Journal on Discrete Mathematics*, 12(2):160–207, 1999.
3. E. Boros, K. Elbassioni, V. Gurvich, and L. Khachiyan. An inequality for polymatroid functions and its applications. *Discrete Applied Mathematics*, 131:255–281, 2003.
4. E. Boros, K. Elbassioni, V. Gurvich, and L. Khachiyan. Enumerating minimal dicuts and strongly connected subgraphs and related geometric problems. In D. Bienstock and G. Nemhauser, editors, *Integer Programming and Combinatorial Optimization, 10th International IPCO Conference, New York, NY, USA*, volume 3064 of *Lecture Notes in Computer Science*, pages 152–162, Berlin, Heidelberg, New York, June 7-11 2004. Springer. (Full version is to appear in *Algorithmica*).
5. C.J. Coulbourn. *The Combinatorics of Network Reliability.* Oxford University Press, 1987.
6. T. Eiter and G. Gottlob. Identifying the minimal transversals of a hypergraph and related problems. *SIAM Journal on Computing*, 24:1278–1304, 1995.
7. M. Fredman and L. Khachiyan. On the complexity of dualization of monotone disjunctive normal forms. *Journal of Algorithms*, 21:618–628, 1996.
8. T. Ishii, H. Nagamochi, and T. Ibaraki. Optimal augmentation to make a graph k-edge-connected and triconnected. In *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, pages 280–289, San Francisco, California, United States, 1998.

9. L. Khachiyan, E. Boros, K. Borys, K. Elbassioni, V. Gurvich, and K. Makino. Generating cut conjunctions and bridge avoiding extensions in graphs. In *Algorithms and Computation: 16th International Symposium, ISAAC 2005, Sanya, Hainan, China, December 19-21, 2005*, pages 156–165, 2005.
10. E. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Generating all maximal independent sets: NP-hardness and polynomial-time algorithms. *SIAM Journal on Computing*, 9:558–565, 1980.
11. L. Lovasz. Submodular functions and convexity. In M. Grotschel A. Bachem and B. Korte, editors, *Mathematical Programming: The State of the Art*, pages 235–257, New York, 1983. Springer-Verlag.
12. J.G. Oxley. *Matroid Theory*. Oxford University Press, Oxford, New York, Tokyo, 1992.
13. R. C. Read and R. E. Tarjan. Bounds on backtrack algorithms for listing cycles, paths, and spanning trees. *Networks*, 5:237–252, 1975.
14. B. Schwikowski and E. Speckenmeyer. On enumerating all minimal solutions of feedback problems. *Discrete Applied Mathematics*, 117:253–265, 2002.
15. L. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8:410–421, 1979.
16. D.J.A. Welsh. *Matroid Theory*. Academic Press, London, 1976.

Less Hashing, Same Performance: Building a Better Bloom Filter

Adam Kirsch* and Michael Mitzenmacher**

Division of Engineering and Applied Sciences
Harvard University, Cambridge, MA 02138
{kirsch, michaelm}@eecs.harvard.edu

Abstract. A standard technique from the hashing literature is to use two hash functions $h_1(x)$ and $h_2(x)$ to simulate additional hash functions of the form $g_i(x) = h_1(x) + ih_2(x)$. We demonstrate that this technique can be usefully applied to Bloom filters and related data structures. Specifically, only two hash functions are necessary to effectively implement a Bloom filter without any loss in the asymptotic false positive probability. This leads to less computation and potentially less need for randomness in practice.

1 Introduction

A Bloom filter is a simple space-efficient randomized data structure for representing a set in order to support membership queries. Although Bloom filters allow false positives, the space savings often outweigh this drawback. The Bloom filter and its many variations have proven increasingly important for many applications (see, for example, the survey [3]). Although potential alternatives have been proposed [15], the Bloom filter's simplicity, ease of use, and excellent performance make it a standard data structure that is and will continue to be of great use in many applications. For space reasons, we do not review the standard Bloom filter results; for more background, see [3].

In this paper, we show that applying a standard technique from the hashing literature can simplify the implementation of Bloom filters significantly. The idea is the following: two hash functions $h_1(x)$ and $h_2(x)$ can simulate more than two hash functions of the form $g_i(x) = h_1(x) + ih_2(x)$. (See, for example, Knuth's discussion of open addressing with double hashing [11].) In our context i will range from 0 up to some number $k - 1$ to give k hash functions, and the hash values are taken modulo the size of the relevant hash table. We demonstrate that this technique can be usefully applied to Bloom filters and related data structures. Specifically, only two hash functions are necessary to effectively implement a Bloom filter without any increase in the asymptotic false positive probability. This leads to less computation and potentially less need

* Supported in part by an NSF Graduate Research Fellowship, NSF grants CCR-9983832 and CCR-0121154, and a grant from Cisco Systems.

** Supported in part by NSF grants CCR-9983832 and CCR-0121154 and a grant from Cisco Systems.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 456–467, 2006.

for randomness in practice. Specifically, in query-intensive applications where computationally non-trivial hash functions are used (such as in [5, 6]), hashing can be a potential bottleneck in using Bloom filters, and reducing the number of required hashes can yield an effective speedup. This improvement was found empirically in the work of Dillinger and Manolios [5, 6], who suggested using the hash functions $g_i(x) = h_1(x) + ih_2(x) + i^2 \bmod m$, where m is the size of the hash table.

Here we provide a full theoretical analysis that holds for a wide class of variations of this technique, justifies and gives insight into the previous empirical observations, and is interesting in its own right. In particular, our methodology generalizes the standard asymptotic analysis of a Bloom filter, exposing a new convergence result that provides a common unifying intuition for the asymptotic false positive probabilities of the standard Bloom filter and the generalized class of Bloom filter variants that we analyze in this paper. We obtain this result by a surprisingly simple approach; rather than attempt to directly analyze the asymptotic false positive probability, we formulate the initialization of the Bloom filter as a balls-and-bins experiment, prove a convergence result for that experiment, and then obtain the asymptotic false positive probability as a corollary.

We start by analyzing a specific, somewhat idealized Bloom filter variation that provides the main insights and intuition for deeper results. We then move to a more general setting that covers several issues that might arise in practice, such as when the size of the hash table is a power of two as opposed to a prime.

Because of space limitations, we leave some results in the full version of this paper [10]. For example, rate of convergence results appear in the full version [10], although in Section 6 we provide some experimental results showing that the asymptotics kick in quickly enough for this technique to be effective in practice. Also, in the full version we demonstrate the utility of this approach beyond the simple Bloom filter by showing how it can be used to reduce the number of hash functions required for Count-Min sketches [4], a variation of the Bloom filter idea used for keeping approximate counts of frequent items in data streams.

Before beginning, we note that Luecker and Molodowitch [12] and Schmidt and Siegel [17] have shown that in the setting of open addressed hash tables, the double hashing technique gives the same performance as uniform hashing. These results are similar in spirit to ours, but the Bloom filter setting is sufficiently different from that of an open addressed hash table that we do not see a direct connection. We also note that our use of hash functions of the form $g_i(x) = h_1(x) + ih_2(x)$ may appear similar to the use of pairwise independent hash functions, and that one might wonder whether there is any formal connection between the two techniques in the Bloom filter setting. Unfortunately, this is not the case; a straightforward modification of the standard Bloom filter analysis yields that if pairwise independent hash functions are used instead of fully random hash functions, then the space required to retain the same bound on the false positive probability increases by a constant factor. In contrast, we show that using the g_i's causes *no* increase in the false positive probability, so they can truly be used as a replacement for fully random hash functions.

2 A Simple Construction Using Two Hash Functions

As an instructive example case, we consider a specific application of the general technique described in the introduction. We devise a Bloom filter that uses k fully random hash functions on some universe U of items, each with range $\{0, 1, 2, \ldots, p-1\}$ for a prime p. Our hash table consists of $m = kp$ bits; each hash function is assigned a disjoint subarray of p bits in the filter, that we treat as numbered $\{0, 1, 2, \ldots, p-1\}$. Our k hash functions will be of the form $g_i(x) = h_1(x) + ih_2(x) \bmod p$, where $h_1(x)$ and $h_2(x)$ are two independent, uniform random hash functions on the universe with range $\{0, 1, 2, \ldots, p-1\}$, and throughout we assume that i ranges from 0 to $k-1$.

As with a standard partitioned Bloom filter, we fix some set $S \subseteq U$ and initialize the filter with S by first setting all of the bits to 0 and then, for each $x \in S$ and i, setting the $g_i(x)$-th bit of the i-th subarray to 1. For any $y \in U$, we answer a query of the form "Is $y \in S$?" with "Yes" if and only if the $g_i(y)$-th bit of the i-th subarray is 1 for every i. Thus, an item $z \notin S$ generates a false positive if and only if each of its hash locations in the array is also a hash location for some $x \in S$.

The advantage of our simplified setting is that for any two elements $x, y \in U$, exactly one of the following three cases occurs: $g_i(x) \neq g_i(y)$ for all i, or $g_i(x) = g_i(y)$ for exactly one i, or $g_i(x) = g_i(y)$ for all i. That is, because we have partitioned the bit array into disjoint hash tables, each hash function can be considered separately. Moreover, by working modulo p, we have arranged that if $g_i(x) = g_i(y)$ for at least two values of i, then we must have $h_1(x) = h_1(y)$ and $h_2(x) = h_2(y)$, so all hash values are the same. This codifies the intuition behind our result: the most likely way for a false positive to occur is when each element in the Bloom filter set S collides with at most one array bit corresponding to the element generating the false positive; other events that cause an element to generate a false positive occur with vanishing probability. It is this intuition that motivates our analysis; in Section 3, we consider more general cases where other non-trivial collisions can occur.

Proceeding formally, we fix a set $S = \{x_1, x_2, \ldots, x_n\}$ of n elements from U and another element $z \notin S$, and compute the probability that z yields a false positive. A false positive corresponds to the event $\mathcal{F}$ that for each i there is (at least) one j such that $g_i(z) = g_i(x_j)$. Obviously, one way this can occur is if $h_1(x_j) = h_1(z)$ and $h_2(x_j) = h_2(z)$ for some j. The probability of this event $\mathcal{E}$ is

$$\mathbf{Pr}(\mathcal{E}) = 1 - \left(1 - 1/p^2\right)^n = 1 - \left(1 - k^2/m^2\right)^n.$$

Notice that when $m/n = c$ is a constant and k is a constant, as is standard for a Bloom filter, we have $\mathbf{Pr}(\mathcal{E}) = o(1)$. Now since

$$\begin{aligned}\mathbf{Pr}(\mathcal{F}) &= \mathbf{Pr}(\mathcal{F} \mid \mathcal{E})\,\mathbf{Pr}(\mathcal{E}) + \mathbf{Pr}(\mathcal{F} \mid \neg\mathcal{E})\,\mathbf{Pr}(\neg\mathcal{E}) \\ &= o(1) + \mathbf{Pr}(\mathcal{F} \mid \neg\mathcal{E})(1 - o(1)),\end{aligned}$$

it suffices to consider $\mathbf{Pr}(\mathcal{F} \mid \neg\mathcal{E})$ to obtain the (constant) asymptotic false positive probability.

Conditioned on $\neg\mathcal{E}$ and $(h_1(z), h_2(z))$, the pair $(h_1(x_j), h_2(x_j))$ is uniformly distributed over the $p^2 - 1$ values in $V = \{0, \ldots, p-1\}^2 - \{(h_1(z), h_2(z))\}$. Of these, for each $i^* \in \{0, \ldots, k-1\}$, the $p-1$ pairs in

$$V_{i^*} = \{(a, b) \in V \,:\, a \equiv i^*(h_2(z) - b) + h_1(z) \bmod p,\ b \not\equiv h_2(z) \bmod p\}$$

are the ones such that if $(h_1(x_j), h_2(x_j)) \in V_{i^*}$, then i^* is the unique value of i such that $g_i(x_j) = g_i(z)$. We can therefore view the conditional probability as a variant of a balls-and-bins problem. There are n balls (each corresponding to some $x_j \in S$), and k bins (each corresponding to some $i^* \in \{0, \ldots, k-1\}$). With probability $k(p-1)/(p^2-1) = k/(p+1)$ a ball lands in a bin, and with the remaining probability it is discarded; when a ball lands in a bin, the bin it lands in is chosen uniformly at random. What is the probability that all of the bins have at least one ball?

This question is surprisingly easy to answer. By the Poisson approximation, the total number of balls that are not discarded has distribution $\mathrm{Bin}(n, k/(p+1)) \approx \mathrm{Po}(k^2/c)$, where $\mathrm{Bin}(\cdot,\cdot)$ and $\mathrm{Po}(\cdot)$ denote the binomial and Poisson distributions, respectively. Since each ball that is not discarded lands in a bin chosen at random, the joint distribution of the number of balls in the bins is asymptotically the same as the joint distribution of k independent $\mathrm{Po}(k/c)$ random variables, by a standard property of Poisson random variables. The probability that each bin has a least one ball now clearly converges to

$$\mathbf{Pr}(\mathrm{Po}(k/c) > 0)^k = (1 - \exp[-k/c])^k\,,$$

which is the asymptotic false positive probability for a standard Bloom filter, completing the analysis.

We make the above argument much more general and rigorous in Section 3, but for now we emphasize that we have actually characterized much more than just the false positive probability of our Bloom filter variant. In fact, we have characterized the asymptotic joint distribution of the number of items in S hashing to the locations used by some $z \notin S$ as being independent $\mathrm{Po}(k/c)$ random variables. Furthermore, from a technical perspective, this approach appears fairly robust. In particular, the above analysis uses only the facts that the probability that some $x \in S$ shares more than one of z's hash locations is $o(1)$, and that if some $x \in S$ shares exactly one of z's hash locations, then that hash location is nearly uniformly distributed over z's hash locations. These observations suggest that the techniques used in this section can be generalized to handle a much wider class of Bloom filter variants, and form the intuitive basis for the arguments in Section 3.

3 A General Framework

In this section, we introduce a general framework for analyzing Bloom filter variants, such as the one examined in Section 2. We start with some new notation. For any integer ℓ, we define the set $[\ell] = \{0, 1, \ldots, \ell - 1\}$ (note that this definition is slightly non-standard). We denote the support of a random variable X

by $\mathrm{Supp}(X)$. For a multi-set M, we use $|M|$ to denote the number of distinct elements of M, and $\|M\|$ to denote the number of elements of M with multiplicity. For two multi-sets M and M', we define $M \cap M'$ and $M \cup M'$ to be, respectively, the intersection and union of M' as *multi-sets*. Furthermore, in an abuse of standard notation, we define the statement $i, i \in M$ as meaning that i is an element of M of multiplicity at least 2.

We are now ready to define the framework. As before, U denotes the universe of items and $S \subseteq U$ denotes the set of n items for which the Bloom filter will answer membership queries. We define a *scheme* to be a method of assigning hash locations to every element of U. Formally, a scheme is specified by a joint distribution of discrete random variables $\{H(u) : u \in U\}$ (implicitly parameterized by n), where for $u \in U$, $H(u)$ represents the multi-set of hash-locations assigned to u by the scheme. We do not require a scheme to be defined for every value of n, but we do insist that it be defined for infinitely many values of n, so that we may take limits as $n \to \infty$. For example, for the class of schemes discussed in Section 2, we think of the constants k and c as being fixed to give a particular scheme that is defined for those values of n such that $p \stackrel{\text{def}}{=} m/k$ is a prime, where $m \stackrel{\text{def}}{=} cn$. Since there are infinitely many primes, the asymptotic behavior of this scheme as $n \to \infty$ is well-defined and is the same as in Section 2, where we let m be a free parameter and analyzed the behavior as $n, m \to \infty$ subject to m/n and k being fixed constants, and m/k being prime.

Having defined the notion of a scheme, we may now formalize some important concepts with new notation (all of which is implicitly parameterized by n). We define H to be the set of all hash locations that can be assigned by the scheme (formally, H is the set of elements that appear in some multi-set in the support of $H(u)$, for some $u \in U$). For $x \in S$ and $z \in U - S$, define $C(x, z) = H(x) \cap H(z)$ to be the multi-set of hash collisions of x with z. We let $\mathcal{F}(z)$ denote the *false positive event* for $z \in U - S$, which occurs when each of z's hash locations is also a hash location for some $x \in S$.

In the schemes that we consider, $\{H(u) : u \in U\}$ will always be independent and identically distributed. In this case, $\mathbf{Pr}(\mathcal{F}(z))$ is the same for all $z \in U - S$, as is the joint distribution of $\{C(x, z) : x \in S\}$. Thus, to simplify the notation, we may fix an arbitrary $z \in U - S$ and simply use $\mathbf{Pr}(\mathcal{F})$ instead of $\mathbf{Pr}(\mathcal{F}(z))$ to denote the false positive probability, and we may use $\{C(x) : x \in S\}$ instead of $\{C(x, z) : x \in S\}$ to denote the joint probability distribution of the multi-sets of hash collisions of elements of S with z.

The main technical result of this section is the following key theorem, which is a formalization and generalization of the analysis in Section 2.

Theorem 1. *Fix a scheme. Suppose that there are constants λ and k such that:*

1. *$\{H(u) : u \in U\}$ are independent and identically distributed.*
2. *For $u \in U$, $\|H(u)\| = k$.*

3. For $x \in S$, $\mathbf{Pr}(\|C(x)\| = i) = \begin{cases} 1 - \frac{\lambda}{n} + o(1/n) & i = 0 \\ \frac{\lambda}{n} + o(1/n) & i = 1 \\ o(1/n) & i > 1 \end{cases}$.

4. For $x \in S$, $\max_{i \in H} \left| \mathbf{Pr}(i \in C(x) \mid \|C(x)\| = 1,\ i \in H(z)) - \frac{1}{k} \right| = o(1)$.

Then $\lim_{n \to \infty} \mathbf{Pr}(\mathcal{F}) = \left(1 - \mathrm{e}^{-\lambda/k}\right)^k$.

Proof. For ease of exposition, we assign every element of $H(z)$ a unique number in $[k]$ (treating multiple instances of the same hash location as distinct elements). More formally, we define an arbitrary bijection f_M from M to $[k]$ for every multiset $M \subseteq H$ with $\|M\| = k$ (where f_M treats multiple instances of the same hash location in M as distinct elements), and label the elements of $H(z)$ according to $f_{H(z)}$. This convention allows us to identify the elements of $H(z)$ by numbers $i \in [k]$, rather than hash locations $i \in H$.

For $i \in [k]$ and $x \in S$, define $X_i(x) = 1$ if $i \in C(x)$ and 0 otherwise, and define $X_i \stackrel{\text{def}}{=} \sum_{x \in S} X_i(x)$. Note that $i \in C(x)$ is an abuse of notation; what we really mean is $f^{-1}_{H(z)}(i) \in C(x)$, although we will continue using the former since it is much less cumbersome. We show that $X^n \stackrel{\text{def}}{=} (X_0, \ldots, X_{k-1})$ converges in distribution to a vector $P \stackrel{\text{def}}{=} (P_0, \ldots, P_{k-1})$ of k independent $\mathrm{Po}(\lambda/k)$ random variables as $n \to \infty$. To do this, we make use of moment generating functions. For a random variable R, the moment generating function of R is defined by $M_R(t) \stackrel{\text{def}}{=} \mathbf{E}[\exp(tR)]$. We show that for any $t_0, \ldots, t_k$, $\lim_{n \to \infty} M_{\sum_{i=0}^{k-1} t_i X_i}(t_k) = M_{\sum_{i=0}^{k-1} t_i P_i}(t_k)$, which is sufficient by [1, Theorem 29.4 and p. 390], since $M_{\sum_{i=0}^{k-1} t_i P_i}(t_k) = \exp\left[\frac{\lambda}{k}\left(\sum_{i \in k} \mathrm{e}^{t_k t_i} - 1\right)\right] < \infty$, by an easy calculation. Proceeding, we write

$$M_{\sum_{i \in [k]} t_i X_i}(t_k) = M_{\sum_{i \in [k]} t_i \sum_{x \in S} X_i(x)}(t_k) = M_{\sum_{x \in S} \sum_{i \in [k]} t_i X_i(x)}(t_k)$$
$$= \left(M_{\sum_{i \in [k]} t_i X_i(x)}(t_k)\right)^n,$$

where the first two steps are obvious, and the third step follows from the fact that the $H(x)$'s are independent and identically distributed (for $x \in S$) conditioned on $H(z)$, so the $\sum_{i \in [k]} t_i X_i(x)$'s are too, since each is a function of the corresponding $H(x)$. Continuing, we have (as $n \to \infty$)

$$\left(M_{\sum_{i \in [k]} t_i X_i(x)}(t_k)\right)^n$$
$$= \Bigg(\mathbf{Pr}(\|C(x)\| = 0) + \sum_{j=1}^{k} \mathbf{Pr}(\|C(x)\| = j)$$
$$\times \sum_{T \subseteq [k] : |T| = j} \mathbf{Pr}(C(x) = f^{-1}_{H(z)}(T) \mid \|C(x)\| = j) \mathrm{e}^{t_k \sum_{i \in T} t_i} \Bigg)^n$$
$$= \left(1 - \frac{\lambda}{n} + \frac{\lambda \sum_{i \in [k]} \mathrm{e}^{t_k t_i}}{kn} + o(1/n)\right)^n$$
$$\to \mathrm{e}^{-\lambda + \frac{\lambda}{k} \sum_{i \in [k]} \mathrm{e}^{t_k t_i}} = \mathrm{e}^{\frac{\lambda}{k}\left(\sum_{i \in [k]} \left(\mathrm{e}^{t_k t_i} - 1\right)\right)} = M_{\sum_{i \in [k]} t_i \mathrm{Po}_i(\lambda_k)}(t_k).$$

The first step follows from the definition of the moment generating function. The second step follows from the assumptions on the distribution of $C(x)$ (the conditioning on $i \in H(z)$ is implicit in our convention that associates integers in $[k]$ with the elements of $H(z)$). The next two steps are obvious, and the last step follows from a previous computation.

We have now established that X^n converges to P in distribution as $n \to \infty$. Standard facts from probability theory [1] now imply that as $n \to \infty$,

$$\mathbf{Pr}(\mathcal{F}) = \mathbf{Pr}(\forall i \in [k], X_i > 0) \to \mathbf{Pr}(\forall i \in [k], P_i > 0) = \left(1 - \mathrm{e}^{-\lambda/k}\right)^k.$$

□

It turns out that the conditions of Theorem 1 can be verified very easily in many cases.

Lemma 1. *Fix a scheme. Suppose that there are constants λ and k such that:*

1. *$\{H(u) : u \in U\}$ are independent and identically distributed.*
2. *For $u \in U$, $\|H(u)\| = k$.*
3. *For $u \in U$, $\max_{i \in H} \left|\mathbf{Pr}(i \in H(u)) - \frac{\lambda}{kn}\right| = o(1/n)$.*
4. *For $u \in U$, $\max_{i_1, i_2 \in H} \mathbf{Pr}(i_1, i_2 \in H(u)) = o(1/n)$.*
5. *The set of all possible hash locations H satisfies $|H| = O(n)$.*

Then the conditions of Theorem 1 hold (with the same values for λ and k), and so the conclusion does as well.

Remark 1. Recall that, under our notation, the statement $i, i \in H(u)$ is true if and only if i is an element of $H(u)$ of multiplicity at least 2.

Proof. The proof is essentially just a number of applications of the first two Boole-Bonferroni inequalities. For details, see [10].

4 Some Specific Schemes

We are now ready to analyze some specific schemes. In particular, we examine a natural generalization of the scheme described in Section 2, as well as the double hashing and extended double hashing schemes introduced in [5, 6]. In both of these cases, we consider a Bloom filter consisting of an array of $m = cn$ bits and k hash functions, where $c > 0$ and $k \geq 1$ are fixed constants. The nature of the hash functions depends on the particular scheme under consideration.

4.1 Partition Schemes

First, we consider the class of *partition schemes*, where the Bloom filter is defined by an array of m bits that is partitioned into k disjoint arrays of $m' = m/k$ bits (we require that m be divisible by k), and an item $u \in U$ is hashed to location

$$h_1(u) + ih_2(u) \bmod m'$$

of array i, for $i \in [k]$, where h_1 and h_2 are independent fully random hash functions with codomain $[m']$. Note that the scheme analyzed in Section 2 is a partition scheme where m' is prime (and so is denoted by p in Section 2).

Unless otherwise stated, henceforth we do all arithmetic involving h_1 and h_2 modulo m'. We prove the following theorem concerning partition schemes.

Theorem 2. *For a partition scheme,* $\lim_{n\to\infty} \mathbf{Pr}(\mathcal{F}) = \left(1 - \mathrm{e}^{-k/c}\right)^k$.

Proof. We show that the $H(u)$'s satisfy the conditions of Lemma 1 with $\lambda = k^2/c$. For $i \in [k]$ and $u \in U$, define $g_i(u) = (i, h_1(u) + ih_2(u))$ and $H(u) = (g_i(u) : i \in [k])$. That is, $g_i(u)$ is u's ith hash location, and $H(u)$ is the multi-set of u's hash locations. This notation is obviously consistent with the definitions required by Lemma 1.

Since h_1 and h_2 are independent and fully random, the first two conditions are trivial. The last condition is also trivial, since there are $m = cn$ possible hash locations. For the remaining two conditions, fix $u \in U$. Observe that for $(i, r) \in [k] \times [m']$,

$$\mathbf{Pr}((i, r) \in H(u)) = \mathbf{Pr}(h_1(u) = r - ih_2(u)) = 1/m' = (k^2/c)/kn,$$

and that for distinct $(i_1, r_1), (i_2, r_2) \in [k] \times [m']$, we have

$$\begin{aligned}
&\mathbf{Pr}((i_1, r_1), (i_2, r_2) \in H(u)) \\
&= \mathbf{Pr}(i_1 \in H(u))\, \mathbf{Pr}(i_2 \in H(u) \mid i_1 \in H(u)) \\
&= \frac{1}{m'}\, \mathbf{Pr}(h_1(u) = r_2 - i_2 h_2(u) \mid h_1(u) = r_1 - i_1 h_2(u)) \\
&= \frac{1}{m'}\, \mathbf{Pr}((i_1 - i_2)h_2(u) = r_1 - r_2) \\
&\le \frac{1}{m'} \cdot \frac{\gcd(|i_2 - i_1|, m')}{m'} \le \frac{k}{(m')^2} = o(1/n),
\end{aligned}$$

where the fourth step is the only nontrivial step, and it follows from the standard fact that for any $r, s \in [m]$, there are at most $\gcd(r, m)$ values $t \in [m]$ such that $rt \equiv s \bmod m$ (see, for example, [9, Proposition 3.3.1]). Finally, since it is clear that from the definition of the scheme that $|H(u)| = k$ for all $u \in U$, we have that for any $(i, r) \in [k] \times [m']$, $\mathbf{Pr}((i, r), (i, r) \in H(u)) = 0$. □

4.2 (Extended) Double Hashing Schemes

Next, we consider the class of double hashing and extended double hashing schemes, which are analyzed empirically in [5, 6]. In these schemes, an item $u \in U$ is hashed to location

$$h_1(u) + ih_2(u) + f(i) \bmod m$$

of the array of m bits, for $i \in [k]$, where h_1 and h_2 are independent fully random hash functions with codomain $[m]$, and $f : [k] \to [m]$ is an arbitrary function.

When $f(i) \equiv 0$, the scheme is called a *double hashing scheme*. Otherwise, it is called an *extended double hashing scheme (with f)*. We show that the asymptotic false positive probability for an (extended) double hashing scheme is the same as for a standard Bloom filter. The proof is analogous to the proof of Theorem 2. For details, see the technical report version of this paper [10].

Theorem 3. *For any (extended) double hashing scheme,*

$$\lim_{n\to\infty} \mathbf{Pr}(\mathcal{F}) = \left(1 - \mathrm{e}^{-k/c}\right)^k.$$

5 Multiple Queries

In the previous sections, we analyzed the behavior of $\mathbf{Pr}(\mathcal{F}(z))$ for some fixed z and moderately sized n. Unfortunately, this quantity is not directly of interest in most applications. Instead, one is usually concerned with certain characteristics of the distribution of the number of elements in a sequence (of distinct elements) $z_1, \ldots, z_\ell \in U - S$ for which $\mathcal{F}(z)$ occurs. In other words, rather than being interested in the probability that a particular false positive occurs, we are concerned with, for example, the fraction of distinct queries on elements of $U - S$ posed to the filter for which it returns false positives. Since $\{\mathcal{F}(z) : z \in U - S\}$ are not independent, the behavior of $\mathbf{Pr}(\mathcal{F})$ alone does not directly imply results of this form. This section is devoted to overcoming this difficulty.

We start with a definition.

Definition 1. *Consider any scheme where $\{H(u) : u \in U\}$ are independent and identically distributed. Write $S = \{x_1, \ldots, x_n\}$. The* false positive rate *is defined to be the random variable $R = \mathbf{Pr}(\mathcal{F} \mid H(x_1), \ldots, H(x_n))$.*

The false positive rate gets its name from the fact that, conditioned on R, the events $\{\mathcal{F}(z) : z \in U - S\}$ are independent with common probability R. Thus, the fraction of a large number of queries on elements of $U - S$ posed to the filter for which it returns false positives is very likely to be close to R. In this sense, R, while a random variable, acts like a rate for $\{\mathcal{F}(z) : z \in U - S\}$.

It is important to note that in much of literature concerning standard Bloom filters, the false positive rate is not defined as above. Instead the term is often used as a synonym for the false positive probability. Indeed, for a standard Bloom filter, the distinction between the two concepts as we have defined them is unimportant in practice, since one can easily show that R is very close to $\mathbf{Pr}(\mathcal{F})$ with extremely high probability (see, for example, [13]). It turns out that this result generalizes very naturally to the framework presented in this paper, and so the practical difference between the two concepts is largely unimportant even in our very general setting. However, the proof is more complicated than in the case of a standard Bloom filter, and so we must be careful to use the terms as we have defined them.

We give only an outline of our results here, deferring the details to [10]. First, we use a standard Doob martingale argument to apply the Azuma-Hoeffding

inequality to R, which tells us that R is concentrated around $\mathbf{E}[R] = \mathbf{Pr}(\mathcal{F})$. We then use that result to prove versions of the strong law of large numbers, the weak law of large numbers, Hoeffding's inequality, and the central limit theorem.

6 Experiments

In this section, we evaluate the theoretical results of the previous sections empirically for small values of n. We are interested in the following specific schemes: the standard Bloom filter scheme, the partition scheme, the double hashing scheme, and the extended double hashing schemes where $f(i) = i^2$ and $f(i) = i^3$.

For $c \in \{4, 8, 12, 16\}$, we do the following. First, compute the value of $k \in \{\lfloor c \ln 2 \rfloor, \lceil c \ln 2 \rceil\}$ that minimizes $p = (1 - \exp[-k/c])^k$. Next, for each of the schemes under consideration, repeat the following procedure 10,000 times: instantiate the filter with the specified values of n, c, and k, populate the filter with a set S of n items, and then query $\lceil 10/p \rceil$ elements not in S, recording the number Q of those queries for which the filter returns a false positive. We then approximate the false positive probability of the scheme by averaging the results over all 10,000 trials. We use the standard Java pseudorandom number generator to simulate independent hash values.

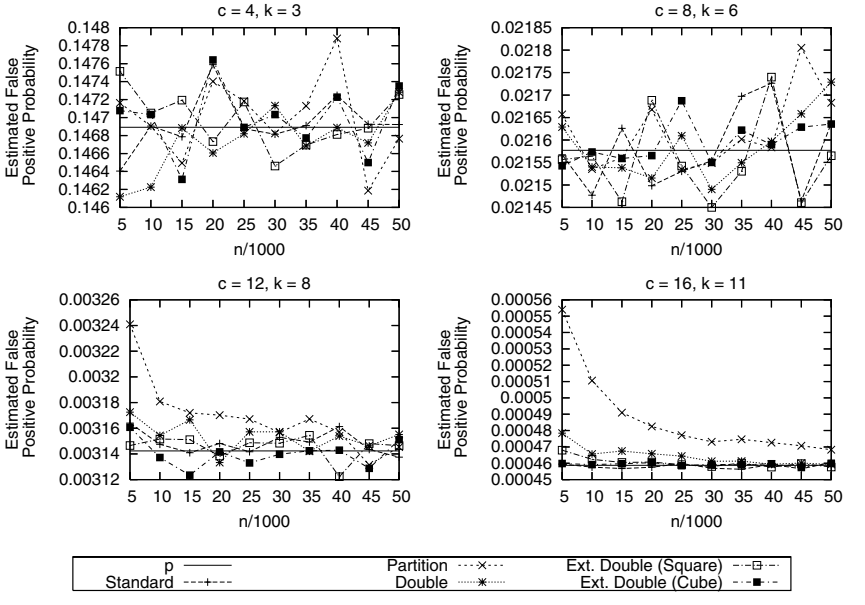


Fig. 1. Estimates of the false positive probability for various schemes and parameters

The results are shown in Figure 1. In Figure 1, we see that for small values of c, the different schemes are essentially indistinguishable from each other, and simultaneously have a false positive probability/rate close to p. This result is

particularly significant since the filters that we are experimenting with are fairly small, supporting our claim that these schemes are useful even in settings with very limited space. However, we also see that for the slightly larger values of $c \in \{12, 16\}$, the partition scheme is no longer particularly useful for small values of n, while the other schemes are. This result is not particularly surprising, since we know from [10, Section 6] that all of these schemes are unsuitable for small values of n and large values of c. Furthermore, we expect that the partition scheme is the least suited to these conditions, given the standard fact that the partitioned version of a standard Bloom filter never performs better than the original version. Nevertheless, the partition scheme might still be useful in certain settings, since it gives a substantial reduction in the range of the hash functions.

7 Conclusion

Bloom filters are simple randomized data structures that are extremely useful in practice. In fact, they are so useful that any significant reduction in the time required to perform a Bloom filter operation immediately translates to a substantial speedup for many practical applications. Unfortunately, Bloom filters are so simple that they do not leave much room for optimization.

This paper focuses on modifying Bloom filters to use less of the only resource that they traditionally use liberally: (pseudo)randomness. Since the only nontrivial computations performed by a Bloom filter are the constructions and evaluations of pseudorandom hash functions, any reduction in the required number of pseudorandom hash functions yields a nearly equivalent reduction in the time required to perform a Bloom filter operation (assuming, of course, that the Bloom filter is stored entirely in memory, so that random accesses can be performed very quickly).

We have shown that a Bloom filter can be implemented with only two pseudorandom hash functions without any increase in the asymptotic false positive probability. We have also shown that the asymptotic false positive probability acts, for all practical purposes and reasonable settings of a Bloom filter's parameters, like a false positive rate. This result has enormous practical significance, since the analogous result for standard Bloom filters is essentially the theoretical justification for their extensive use.

More generally, we have given a framework for analyzing modified Bloom filters, which we expect will be used in the future to refine the specific schemes that we analyzed in this paper. We also expect that the techniques used in this paper will be usefully applied to other data structures, as demonstrated by our modification to the Count-Min sketch (in [10]).

Acknowledgements

We are very grateful to Peter Dillinger and Panagiotis Manolios for introducing us to this problem, providing us with advance copies of their work, and also for many useful discussions.

References

1. P. Billingsley. Probability and Measure, Third Edition. John Wiley & Sons, 1995.
2. P. Bose, H. Guo, E. Kranakis, A. Maheshwari, P. Morin, J. Morrison, M. Smid, and Y. Tang. On the false-positive rate of Bloom filters. Submitted. `http://cg.scs.carleton.ca/~morin/publications/ds/bloom-submitted.pdf`
3. A. Broder and M. Mitzenmacher. Network Applications of Bloom Filters: A Survey. *Internet Mathematics*, 1(4):485-509, 2004.
4. G. Cormode and S. Muthukrishnan. Improved Data Stream Summaries: The Count-Min Sketch and its Applications. DIMACS Technical Report 2003-20, 2003.
5. P. C. Dillinger and P. Manolios. Bloom Filters in Probabilistic Verification. In *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2004)*, pp. 367-381, 2004.
6. P. C. Dillinger and P. Manolios. Fast and Accurate Bitstate Verification for SPIN. In *Proceedings of the 11th International SPIN Workshop on Model Checking of Software (SPIN 2004)*, pp. 57-75, 2004.
7. D. P. Dubhashi and D. Ranjan. Balls and Bins: A Case Study in Negative Dependence. *Random Structures and Algorithms*, 13(2):99-124, 1998.
8. L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area Web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281-293, 2000.
9. K. Ireland and M. Rosen. A Classical Introduction to Modern Number Theory, Second Edition. Springer-Verlag, 1990.
10. A. Kirsch and M. Mitzenmacher. Building a Better Bloom Filter. Harvard University Computer Science Technical Report TR-02-05, 2005. `ftp://ftp.deas.harvard.edu/techreports/tr-02-05.pdf`.
11. D. Knuth. The Art of Computer Programming, Volume 3: Sorting and Searching. Addison-Wesley, 1973.
12. G. Lueker and M. Molodowitch. More analysis of double hashing. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing (STOC '88)*, pp. 354-359, 1988.
13. M. Mitzenmacher. Compressed Bloom Filters. *IEEE/ACM Transactions on Networking*, 10(5):613-620, 2002.
14. M. Mitzenmacher and E. Upfal. Probability and Computing: Randomized Algorithms and Probabilistic Analysis. Cambridge University Press, 2005.
15. A. Pagh, R. Pagh, and S. Srinivas Rao. An Optimal Bloom Filter Replacement. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '05)*, pp. 823-829, 2005.
16. M. V. Ramakrishna. Practical performance of Bloom filters and parallel free-text searching. *Communications of the ACM*, 32(10):1237-1239, 1989.
17. J. P. Schmidt and A. Siegel. The analysis of closed hashing under limited randomness. In *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing (STOC '90)*, pp. 224-234, 1990.

A Unified Approach to Approximating Partial Covering Problems

Jochen Könemann[1,⋆], Ojas Parekh[2], and Danny Segev[3]

[1] Department of Combinatorics and Optimization, University of Waterloo, Canada
jochen@math.uwaterloo.ca
[2] Department of Mathematics and Computer Science, Emory University, USA
ojas@mathcs.emory.edu
[3] School of Mathematical Sciences, Tel-Aviv University, Israel
segevd@post.tau.ac.il

Abstract. An instance of the *generalized partial cover* problem consists of a ground set U and a family of subsets $\mathcal{S} \subseteq 2^U$. Each element $e \in U$ is associated with a profit $p(e)$, whereas each subset $S \in \mathcal{S}$ has a cost $c(S)$. The objective is to find a minimum cost subcollection $\mathcal{S}' \subseteq \mathcal{S}$ such that the combined profit of the elements covered by $\mathcal{S}'$ is at least P, a specified profit bound. In the *prize-collecting* version of this problem, there is no strict requirement to cover any element; however, if the subsets we pick leave an element $e \in U$ uncovered, we incur a penalty of $\pi(e)$. The goal is to identify a subcollection $\mathcal{S}' \subseteq \mathcal{S}$ that minimizes the cost of $\mathcal{S}'$ plus the penalties of uncovered elements.

Although problem-specific connections between the partial cover and the prize-collecting variants of a given covering problem have been explored and exploited, a more general connection remained open. The main contribution of this paper is to establish a formal relationship between these two variants. As a result, we present a unified framework for approximating problems that can be formulated or interpreted as special cases of generalized partial cover. We demonstrate the applicability of our method on a diverse collection of covering problems, for some of which we obtain the first non-trivial approximability results.

1 Introduction

For over three decades the *set cover* problem and its ever-growing list of generalizations, variants, and special cases have attracted the attention of researchers in the fields of discrete optimization, complexity theory, and combinatorics. Essentially, these problems are concerned with identifying a minimum cost collection of sets that covers a given set of elements, possibly with additional side constraints. While such settings may appear to be very simple at first glance, they still capture computational tasks of great theoretical and practical importance, as the reader may verify by consulting directly related surveys [2, 13, 18, 28] and the references therein.

In the present paper we focus our attention on the *generalized partial cover* problem, whose input consists of a ground set of elements U and a family $\mathcal{S}$ of subsets of U. In addition, each element $e \in U$ is associated with a profit $p(e)$, whereas each subset $S \in \mathcal{S}$

⋆ Research supported by NSERC grant no. 288340-2004.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 468–479, 2006.

has a cost $c(S)$. The objective is to find a minimum cost subcollection $\mathcal{S}' \subseteq \mathcal{S}$ such that the combined profit of the elements covered by $\mathcal{S}'$ is at least P, a specified profit bound. When all elements are endowed with unit profits, we obtain the well-known *partial cover* problem, in which the goal is to cover a given number of elements by picking subsets of minimum total cost.

Numerous computational problems can be formulated or interpreted as special cases of generalized partial cover, although this fact may be well-hidden. For most of these problems, novel techniques in the design of approximation algorithms have emerged over the years, and it is clearly beyond the scope of this writing to present an exhaustive overview. However, from the abundance of greedy schemes, local-search heuristics, randomized methods, and LP-based algorithms a simple observation is revealed: There is currently no unified approach to approximating partial covering problems.

The suggested method: Preliminaries. The main contribution of this paper is to establish a formal relationship between the partial cover and the prize-collecting versions of a given covering problem. In the *prize-collecting set cover* problem there is no strict requirement to cover any element; however, if the subsets we pick leave an element $e \in U$ uncovered, we incur a penalty of $\pi(e)$. The objective is to find a subcollection $\mathcal{S}' \subseteq \mathcal{S}$ that minimizes the cost of $\mathcal{S}'$ plus the penalties of the uncovered elements. A polynomial-time algorithm for this problem is said to be *Lagrangian multiplier preserving with factor r* (henceforth, r-LMP) if for every instance I it constructs a solution that satisfies $C + r\Pi \leq r \cdot \mathrm{OPT}(I)$, where C is the total cost of the subsets picked, and Π is the sum of penalties over all uncovered elements. We further denote by $\mathcal{I}_r$ the family of weighted set systems $(U, \mathcal{S}, c)$ that possess the following property: There is an r-LMP algorithm for all prize-collecting instances $(U, \mathcal{S}, c, \pi), \pi : U \to \mathbb{Q}_+$. In other words, for every penalty function π the corresponding instance admits an r-LMP approximation.

The main result. At the heart of our method is an algorithm for the generalized partial cover problem that computes an approximate solution by making use of an r-LMP prize-collecting algorithm in a black-box fashion. Specifically, in Section 2 we prove the following theorem.

Theorem 1. *Let I be a generalized partial cover instance defined on an underlying weighted set system $(U, \mathcal{S}, c)$, and suppose that $(U, \mathcal{S}, c) \in \mathcal{I}_r$ for some $r \geq 1$. Then, for any $\epsilon > 0$, we can find a feasible solution to I whose cost is at most $(\frac{4}{3} + \epsilon)r$ times the optimum, within time polynomial in $|U|$, $|\mathcal{S}|^{1/\epsilon}$ and the input length of I.*

Here is a rough outline of how the proof of Theorem 1 will proceed. We begin by formulating the generalized partial cover problem as an integer program. Next, we dualize the complicating constraint that places a lower bound of P on the total profit. More precisely, we lift this constraint to the objective function multiplied by an auxiliary variable λ, and obtain its corresponding Lagrangian relaxation. For any fixed $\lambda \geq 0$, the new program describes, up to a constant term, a prize-collecting set cover instance with non-uniform penalties. We now conduct a binary search, using the r-LMP prize-collecting algorithm as a subroutine, to find sufficiently close $\lambda_1 \geq \lambda_2$ that satisfy: For λ_1, the algorithm constructs a solution $\mathcal{S}_1 \subseteq \mathcal{S}$ such that the total profit of the elements covered by $\mathcal{S}_1$ is at least P; For λ_2, it constructs a solution $\mathcal{S}_2 \subseteq \mathcal{S}$ with a total profit of at most P.

Although we can exploit the r-LMP property to show that the cost of $\mathcal{S}_2$ is within factor r of optimum, this solution is not necessarily feasible. The situation is quite the opposite with respect to $\mathcal{S}_1$, which is a feasible solution whose cost may be arbitrarily large. Having observed these facts, we create an additional feasible solution $\mathcal{S}_3$ by augmenting $\mathcal{S}_2$ with a carefully chosen subset of $\mathcal{S}_1$. The cost of this subset is bounded by extending the arguments used by Levin and Segev [24] and independently by Golovin, Nagarajan and Singh [17] for approximating the *k-multicut* problem. Finally, we establish Theorem 1 by proving that the cost of the cheaper of $\mathcal{S}_1$ and $\mathcal{S}_3$ is at most $(\frac{4}{3}+\epsilon)r$ times the cost of an optimal solution.

Designing LMP algorithms. At this point in time, the reader should bear in mind that the performance guarantee of our algorithm, as stated in Theorem 1, depends on the existence of an LMP prize-collecting algorithm for a given covering problem. Indeed, this dependence appears to be the primary factor limiting the employment of Lagrangian relaxations in most problems of interest. The latter drawback was pointed out by Chudak, Roughgarden and Williamson [6], who asked whether it is possible to devise more general variants of the Lagrangian relaxation framework that apply to a broader class of problems. We answer this question in the affirmative, by developing prize-collecting algorithms with the LMP property for some of the most fundamental integer covering problems. These results, along with a detailed description of previous work, are formally presented in Section 3.

2 The Generalized Partial Cover Algorithm

The main result of this section is a constructive proof of Theorem 1. Recall that a generalized partial cover instance I is defined with respect to an underlying weighted set system, consisting of a ground set U and a family of subsets $\mathcal{S} \subseteq 2^U$, where each $S \in \mathcal{S}$ has a cost $c(S)$. The additional ingredients of I are profits $p(e)$, specified for each element $e \in U$, and a requirement parameter P. Now suppose that $(U, \mathcal{S}, c) \in \mathcal{I}_r$ for some $r \geq 1$, meaning that there is an r-LMP algorithm $\mathcal{A}$ for all prize-collecting instances $(U, \mathcal{S}, c, \pi)$, $\pi : U \to \mathbb{Q}_+$.

2.1 Preliminaries

The method we suggest and its analysis will be based on a natural integer programming formulation of the generalized partial cover problem. In the following, let $\mathcal{S}_e \subseteq \mathcal{S}$ be the collection of sets that contain $e \in U$, and let $P_U = \sum_{e \in U} p(e)$.

$$\text{minimize} \quad \sum_{S \in \mathcal{S}} c(S)x_S \tag{GC}$$

$$\text{subject to} \quad \sum_{S \in \mathcal{S}_e} x_S + z_e \geq 1 \qquad \forall\, e \in U \tag{2.1}$$

$$\sum_{e \in U} p(e) z_e \leq P_U - P \tag{2.2}$$

$$x_S,\, z_e \in \{0,1\} \qquad \forall\, S \in \mathcal{S},\, e \in U \tag{2.3}$$

In this formulation, the variable x_S indicates whether we pick the set S, whereas z_e indicates whether the element e is uncovered. Constraint (2.1) guarantees that we either pick at least one set that contains e, or specify that this element is uncovered by setting $z_e = 1$. Constraint (2.2) forces any feasible solution to cover elements with a total profit of at least P.

Essential to the subsequent analysis will be the fact that the LP-relaxation of (GC), obtained by replacing constraint (2.3) with $x_S \geq 0$ and $z_e \geq 0$, has an integrality gap of $O(r)$. Unfortunately, this prerequisite is not satisfied even in the case of unit profits, as the next example illustrates. Consider an instance in which the ground set U consists of n elements, and the family $\mathcal{S}$ contains a single set $S = U$ with cost n. When we are required to cover at least one element, the integral optimum is clearly n. However, by setting $x_S = \frac{1}{n}$ and $z_e = 1 - \frac{1}{n}$ for every $e \in U$, we define a feasible fractional solution whose cost is 1. This example, as well as additional constructions of similar nature, demonstrate that an unbounded integrality gap may arise whenever a small number of sets in the optimal solution contribute a large fraction of its cost.

Therefore, an inevitable part of our algorithm is a preprocessing step in which, given a fixed accuracy parameter $\epsilon > 0$, we "guess" the $\lfloor \frac{1}{\epsilon} \rfloor$ most expensive sets in the optimal solution, whose cost we denote by OPT. More precisely, we enumerate all $O(|\mathcal{S}|^{1/\epsilon})$ subsets $\mathcal{S}' \subseteq \mathcal{S}$ of cardinality at most $\lfloor \frac{1}{\epsilon} \rfloor$, test each such subset as the correct guess, and return the best solution we find. For a given subset $\mathcal{S}'$, we include it as part of the solution to be constructed, eliminate the sets in $\mathcal{S}'$ from $\mathcal{S}$, remove all covered elements from U and from the remaining sets, and update the profit requirement. Any set whose cost is greater than $\min_{S \in \mathcal{S}'} c(S)$ is also eliminated. Consequently, the cost of each remaining set is at most $\epsilon \cdot \mathrm{OPT}$.

In the remainder of this section we will bypass the preprocessing step, and assume that the maximum cost of a set in $\mathcal{S}$ is at most $\epsilon \cdot \mathrm{OPT}$. For ease of presentation, we also assume that $c(S) > 0$ for every $S \in \mathcal{S}$ and that $p(e) > 0$ for every $e \in U$, since zero-cost sets can be picked in advance and zero-profit elements can be discarded.

2.2 Obtaining $\mathcal{S}_1$ and $\mathcal{S}_2$

We now dualize the profit constraint (2.2), and lift it to the objective function multiplied by $\lambda \geq 0$. The resulting Lagrangian relaxation is:

$$\begin{aligned} \mathrm{LR}(\lambda) = \text{minimize} \quad & \sum_{S \in \mathcal{S}} c(S) x_S + \lambda \left(\sum_{e \in U} p(e) z_e - (P_U - P) \right) \\ \text{subject to} \quad & \sum_{S \in \mathcal{S}_e} x_S + z_e \geq 1 \qquad \forall\, e \in U \\ & x_S,\, z_e \in \{0, 1\} \qquad \forall\, S \in \mathcal{S},\, e \in U \end{aligned}$$

We remark that, excluding the constant term of $-\lambda(P_U - P)$ in the objective function, $\mathrm{LR}(\lambda)$ is an integer programming formulation of the prize-collecting set cover problem, in which each element $e \in U$ is associated with a penalty $\lambda p(e)$. We refer to this instance as I_λ, and use $\mathrm{OPT}(I_\lambda)$ to denote its optimum value. It is not difficult to verify that $\mathrm{LR}(\lambda) = \mathrm{OPT}(I_\lambda) - \lambda(P_U - P)$ is at most OPT for any $\lambda \geq 0$, by observing that an

optimal solution to (GC) is also a feasible solution to LR(λ), whose cost is at most OPT.

Since the underlying weighted set system of I_λ is identical to that of I, we may apply the prize-collecting algorithm $\mathcal{A}$ to approximate I_λ. Let x^λ indicate which sets in $\mathcal{S}$ were picked by the algorithm, and let z^λ indicate which elements were left uncovered. In terms of (x^λ, z^λ), the r-LMP property of $\mathcal{A}$ is equivalent to

$$\sum_{S\in\mathcal{S}} c(S)x_S^\lambda + r\sum_{e\in U}\lambda p(e)z_e^\lambda \le r\cdot \mathrm{OPT}(I_\lambda)\ , \tag{2.4}$$

an inequality that, in particular, leads to the following observation.

Lemma 2. *When $\lambda > \frac{1}{\min_{e\in U} p(e)}\sum_{S\in\mathcal{S}} c(S)$, the solution (x^λ, z^λ) covers all elements. On the other hand, (x^0, z^0) does not cover any element.*

Proof. Let $\lambda > \frac{1}{\min_{e\in U} p(e)}\sum_{S\in\mathcal{S}} c(S)$, and suppose that there is an element $\bar{e}\in U$ for which $z_{\bar{e}}^\lambda = 1$, that is, $\bar{e}$ is not covered by any set the algorithm $\mathcal{A}$ picks when we approximate I_λ. Then (x^λ, z^λ) no longer satisfies inequality (2.4), as

$$\sum_{S\in\mathcal{S}} c(S)x_S^\lambda + r\sum_{e\in U}\lambda p(e)z_e^\lambda \ge r\lambda p(\bar{e}) > r\frac{p(\bar{e})}{\min_{e\in U} p(e)}\sum_{S\in\mathcal{S}} c(S) \ge r\sum_{S\in\mathcal{S}} c(S) \ge r\cdot\mathrm{OPT}(I_\lambda)\ ,$$

where the last inequality holds since $\mathcal{S}$ is a feasible solution to I_λ.

Now let $\lambda = 0$, and note that each element of the instance I_0 has a zero penalty. Therefore, by deciding not to pick any set and instead pay all penalties we obtain a feasible solution with zero cost, implying that $\mathrm{OPT}(I_0) = 0$. Since (x^0, z^0) satisfies inequality (2.4), it follows that this solution cannot pick any set, as all sets in $\mathcal{S}$ have strictly positive costs by assumption. □

This observation allows us to conduct a binary search over $[0, \frac{2}{\min_e p(e)}\sum_{S\in\mathcal{S}} c(S)]$, consisting of a polynomially-bounded number of calls to the prize-collecting algorithm $\mathcal{A}$, as a result of which we find $\lambda_1 \ge \lambda_2$ that satisfy:

1. $\lambda_1 - \lambda_2 \le \frac{\epsilon c_{\min}}{P_U}$, where $c_{\min} = \min_{S\in\mathcal{S}} c(S) > 0$.
2. The elements covered by $(x^{\lambda_1}, z^{\lambda_1})$ have a total profit of $P_1 \ge P$, and at the same time those covered by $(x^{\lambda_2}, z^{\lambda_2})$ have a total profit of $P_2 \le P$.

For ease of notation, we designate by $\mathcal{S}_1$ and $\mathcal{S}_2$ the subsets of $\mathcal{S}$ that were picked by the solutions $(x^{\lambda_1}, z^{\lambda_1})$ and $(x^{\lambda_2}, z^{\lambda_2})$, respectively. Without loss of generality, $P_1 > P$, or otherwise $\mathcal{S}_1$ is already a feasible solution whose cost is at most $r\cdot\mathrm{LR}(\lambda_1) \le r\cdot\mathrm{OPT}$. Similarly, we assume that $P_2 < P$. The analysis of our algorithm crucially depends on the next lemma, which is a consequence of the r-LMP property.

Lemma 3. *Let $\alpha = \frac{P-P_2}{P_1-P_2} \in (0,1)$. Then, $\alpha c(\mathcal{S}_1) + (1-\alpha)c(\mathcal{S}_2) \le r(1+\epsilon)\mathrm{OPT}$.*

Proof. By combining inequality (2.4) with the fact that $\mathrm{LR}(\lambda) = \mathrm{OPT}(I_\lambda) - \lambda(P_U - P) \le \mathrm{OPT}$ for every $\lambda \ge 0$, we have

$$\begin{aligned} c(\mathcal{S}_1) &= \sum_{S\in\mathcal{S}} c(S)x_S^{\lambda_1} \le r\left(\mathrm{OPT}(I_{\lambda_1}) - \lambda_1\sum_{e\in U} p(e)z_e^{\lambda_1}\right) = r\left(\mathrm{OPT}(I_{\lambda_1}) - \lambda_1\left(P_U - P_1\right)\right) \\ &= r(\mathrm{LR}(\lambda_1) + \lambda_1(P_1 - P)) \le r(\mathrm{OPT} + \lambda_1(P_1 - P))\ . \end{aligned} \tag{2.5}$$

A similar argument shows that $c(\mathcal{S}_2) \leq r(\mathrm{OPT} + \lambda_2(P_2 - P))$. Therefore,

$$\begin{aligned}
\alpha c(\mathcal{S}_1) + (1-\alpha)c(\mathcal{S}_2) &\leq \alpha r(\mathrm{OPT} + \lambda_1(P_1 - P)) + (1-\alpha)r(\mathrm{OPT} + \lambda_2(P_2 - P)) \\
&\leq r \cdot \mathrm{OPT} + \alpha r\left(\lambda_2 + \frac{\epsilon c_{\min}}{P_U}\right)(P_1 - P) + (1-\alpha)r\lambda_2(P_2 - P) \\
&= r \cdot \mathrm{OPT} + r\lambda_2\left(\alpha(P_1 - P) + (1-\alpha)(P_2 - P)\right) + r\alpha\epsilon c_{\min} \cdot \frac{P_1 - P}{P_U} \\
&\leq r \cdot \mathrm{OPT} + r\epsilon c_{\min} \\
&\leq r(1+\epsilon)\mathrm{OPT} \ .
\end{aligned}$$

The second inequality follows from observing that $P_1 > P$ and $\lambda_1 \leq \lambda_2 + \frac{\epsilon c_{\min}}{P_U}$. The third inequality holds since $\alpha(P_1 - P) + (1-\alpha)(P_2 - P) = 0$, $\alpha < 1$ and $P_1 - P \leq P_U$. □

2.3 Composing an Additional Solution

Up until now, the only feasible solution we have at our possession is $\mathcal{S}_1$, as this subset of $\mathcal{S}$ covers elements with an overall profit of $P_1 > P$. Inequality (2.5) places an upper bound of $r \cdot \mathrm{OPT} + r\lambda_1(P_1 - P)$ on the cost of $\mathcal{S}_1$. However, the latter term may be arbitrarily large in comparison to OPT, implying that $\mathcal{S}_1$ cannot approximate the instance I by itself. The situation is quite the opposite with respect to $\mathcal{S}_2$: Although this solution covers elements with an insufficient profit of $P_2 < P$, a similar bound of $r \cdot \mathrm{OPT} + r\lambda_2(P_2 - P)$ on its cost actually yields the inequality $c(\mathcal{S}_2) \leq r \cdot \mathrm{OPT}$, since in this case $r\lambda_2(P_2 - P) \leq 0$.

At this point, we are concerned with creating an additional feasible solution $\mathcal{S}_3$, by augmenting $\mathcal{S}_2$ with a carefully chosen subset $\mathcal{S}' \subseteq \mathcal{S}_1$. To attain feasibility, we must ensure that of the elements that were left uncovered by $\mathcal{S}_2$, a subcollection with a total profit of at least $P - P_2$ is covered by $\mathcal{S}'$. We construct this augmenting subset as follows. Let $U' \subseteq U$ be the collection of elements that are covered by $\mathcal{S}_1$ but not by $\mathcal{S}_2$. We assign each element $e \in U'$ to an arbitrary set in $\mathcal{S}_1 \setminus \mathcal{S}_2$ that contains it, and denote by $\varphi(S)$ the total profit of the elements assigned to S. Without loss of generality, we assume that $\mathcal{S}_1 \setminus \mathcal{S}_2 = \{S_1, \ldots, S_k\}$, where these sets are indexed by non-decreasing order of the ratio $\frac{c(S_i)}{\varphi(S_i)}$. Finally, let $\mathcal{S}' = \{S_1, \ldots, S_q\}$, where q is the minimal index for which $\sum_{i=1}^{q} \varphi(S_i) \geq P - P_2$. Note that such an index exists, since $\sum_{i=1}^{k} \varphi(S_i) \geq P_1 - P_2$ and $P_1 > P$. The next lemma bounds the cost of $\mathcal{S}_3 = \mathcal{S}_2 \cup \mathcal{S}'$.

Lemma 4. $c(\mathcal{S}_3) \leq c(\mathcal{S}_2) + \alpha c(\mathcal{S}_1 \setminus \mathcal{S}_2) + \epsilon \cdot \mathrm{OPT}$.

Proof. By assumption, the cost of each set in $\mathcal{S}$ is at most $\epsilon \cdot \mathrm{OPT}$. Therefore, it is sufficient to prove that $c(\mathcal{S}' \setminus \{S_q\}) = \sum_{i=1}^{q-1} c(S_i) \leq \alpha c(\mathcal{S}_1 \setminus \mathcal{S}_2)$. To this end, consider a random variable K that takes the values $1, \ldots, k$, such that $\mathbb{P}(K = i) = \frac{\varphi(S_i)}{\sum_{l=1}^{k} \varphi(S_l)}$, and let $R = \frac{c(S_K)}{\varphi(S_K)}$. Since the sets in $\mathcal{S}_1 \setminus \mathcal{S}_2$ are indexed by non-decreasing order of $\frac{c(S_i)}{\varphi(S_i)}$, we have $\mathbb{E}(R | K \leq q-1) \leq \mathbb{E}(R)$. As $\alpha = \frac{P - P_2}{P_1 - P_2}$, this inequality implies $\sum_{i=1}^{q-1} c(S_i) \leq \alpha c(\mathcal{S}_1 \setminus \mathcal{S}_2)$, since

$$\mathbb{E}(R) = \sum_{i=1}^{k} \frac{c(S_i)}{\varphi(S_i)} \cdot \frac{\varphi(S_i)}{\sum_{l=1}^{k} \varphi(S_l)} = \frac{1}{\sum_{l=1}^{k} \varphi(S_l)} \sum_{i=1}^{k} c(S_i) \leq \frac{1}{P_1 - P_2} c(\mathcal{S}_1 \setminus \mathcal{S}_2)$$

and

$$\mathbb{E}(R|K \leq q-1) = \sum_{i=1}^{q-1} \frac{c(S_i)}{\varphi(S_i)} \cdot \frac{\varphi(S_i)}{\sum_{l=1}^{q-1} \varphi(S_l)} = \frac{1}{\sum_{l=1}^{q-1} \varphi(S_l)} \sum_{i=1}^{q-1} c(S_i) \geq \frac{1}{P-P_2} \sum_{i=1}^{q-1} c(S_i) \ .$$

The last inequality holds since $\sum_{l=1}^{q-1} \varphi(S_l) < P - P_2$, by the minimality of q. □

2.4 Deriving the Approximation Factor

We now conclude the proof of Theorem 1, by demonstrating that the cost of the cheaper of $\mathcal{S}_1$ and $\mathcal{S}_3$ is within factor $(\frac{4}{3} + O(\sqrt{\epsilon}))r$ of optimum. An appropriate choice of ϵ restores the original form of the theorem.

Lemma 5. $\min\{c(\mathcal{S}_1), c(\mathcal{S}_3)\} \leq (\frac{4}{3} + O(\sqrt{\epsilon}))r \cdot \mathrm{OPT}$.

Proof. To simplify the analysis, we begin by introducing a new parameter, $\beta = \frac{c(\mathcal{S}_2)}{\mathrm{OPT}} \in [0, r]$, and bound the cost of $\mathcal{S}_1$ and $\mathcal{S}_3$ in terms of OPT, α and β. We first observe that

$$c(\mathcal{S}_1) = \frac{\alpha c(\mathcal{S}_1)}{\alpha} \leq \frac{r(1+\epsilon)\mathrm{OPT} - (1-\alpha)c(\mathcal{S}_2)}{\alpha} = \frac{r(1+\epsilon) - (1-\alpha)\beta}{\alpha}\mathrm{OPT} \ ,$$

where the first inequality follows from Lemma 3, and the last equation is obtained by substituting $c(\mathcal{S}_2) = \beta \cdot \mathrm{OPT}$. In addition, Lemma 4 implies that

$$\begin{aligned} c(\mathcal{S}_3) &\leq c(\mathcal{S}_2) + \alpha c(\mathcal{S}_1 \setminus \mathcal{S}_2) + \epsilon \cdot \mathrm{OPT} \leq (1-\alpha)c(\mathcal{S}_2) + \alpha c(\mathcal{S}_1) + \alpha c(\mathcal{S}_2) + \epsilon \cdot \mathrm{OPT} \\ &\leq r(1+\epsilon)\mathrm{OPT} + \alpha c(\mathcal{S}_2) + \epsilon \cdot \mathrm{OPT} = (r(1+\epsilon) + \alpha\beta + \epsilon)\mathrm{OPT} \ , \end{aligned}$$

where the third inequality and the last equation follow from Lemma 3 and the definition of β, respectively. Finally, we bound the resulting approximation factor by considering the worst possible choice for the parameters α and β, to conclude that

$$\begin{aligned} \min\{c(\mathcal{S}_1), c(\mathcal{S}_3)\} &\leq \min\left\{\frac{r(1+\epsilon)-(1-\alpha)\beta}{\alpha}, r(1+\epsilon) + \alpha\beta + \epsilon\right\}\mathrm{OPT} \\ &\leq \max_{\substack{\alpha\in(0,1)\\ \beta\in[0,r]}} \min\left\{\frac{r(1+\epsilon)-(1-\alpha)\beta}{\alpha}, r(1+\epsilon) + \alpha\beta\right\}\mathrm{OPT} + \epsilon \cdot \mathrm{OPT} \\ &= \left(\frac{4}{3} + O(\sqrt{\epsilon})\right) r \cdot \mathrm{OPT} \ . \end{aligned}$$

□

3 Applications

In what follows, we demonstrate the applicability of our method on a diverse collection of covering problems, which is by no means exhaustive. Rather, the problems we have chosen to study are only meant to illustrate that the LMP property is applicable in a variety of settings. For the vast majority of these problems, we propose the first algorithm that approximates their generalized partial cover version. For others, our algorithms offer approximation guarantees that compete with the currently best known results. Due to space limitations, we defer the description of problem-specific prize-collecting subroutines to the full version of this paper [23].

3.1 Set Cover, in Terms of Δ

Kearns [21, Thm. 5.15] seems to have been the first to study the partial cover problem, showing that the greedy set cover algorithm [20, 25] can be adapted to provide an approximation factor of $2H(|U|) + 3$. A slightly different algorithm was suggested by Slavík [32], who obtained a factor of $H(\min\{\Delta, k\})$, where Δ is the maximum size of a set in $\mathcal{S}$ and k is the coverage requirement. We remark that the partial cover problem contains set cover as a special case, implying that it cannot be approximated within a factor of $(1-\epsilon)\ln|U|$ for any $\epsilon > 0$, unless NP $\subset$ TIME($n^{O(\log\log n)}$) [10].

To the best of our knowledge, the greedy heuristic has not been studied in the context of generalized partial cover, and in fact no algorithm is currently known for this problem. In the full version of this paper [23], we prove that every weighted set system $(U, \mathcal{S}, c)$ is in $\mathcal{I}_{H(\Delta)}$, where $\Delta = \max_{S\in\mathcal{S}}|S|$. The next theorem follows.

Theorem 6. *The generalized partial set cover problem can be approximated within a factor of* $(\frac{4}{3} + \epsilon)H(\Delta)$*, for any fixed* $\epsilon > 0$.

3.2 Set Cover, in Terms of f

Let f_e be the number of sets in $\mathcal{S}$ that contain the element $e \in U$; f_e is also known as the *frequency* of e. A recent line of work, that was initiated by Bshouty and Burroughs [4] and Hochbaum [19] in the context of *partial vertex cover*, is approximating partial cover in terms of f, the maximum frequency of any element. Based on the local-ratio method, Bar-Yehuda [3] devised an algorithm for generalized partial cover whose approximation guarantee is f, a result that was independently obtained by Fujito [11] using a prima-dual algorithm. Gandhi, Khuller and Srinivasan [12] achieved a similar ratio for partial cover.

In the full version of this paper [23], we present a combinatorial f-LMP algorithm for the prize-collecting set cover problem, showing that every weighted set system $(U, \mathcal{S}, c)$ is in $\mathcal{I}_f$, where $f = \max_{e\in U} f_e$. Combined with Theorem 1, this result allows us to approximate the generalized partial set cover problem within a factor of $(\frac{4}{3}+\epsilon)f$, which is slightly worse than the currently best.

3.3 Laminar Cover

Let $G = (V, E)$ be an undirected graph, in which each edge $e \in E$ has a non-negative cost $c(e)$, and let $\mathcal{F} = \{V_1, \ldots, V_k\} \subseteq 2^V$ be a *laminar family* of vertex sets, meaning that $V_i \cap V_j \in \{\emptyset, V_i, V_j\}$ for every $i \neq j$. We say that an edge e *covers* V_i if it has exactly one endpoint in V_i. The objective is to find a minimum cost set of edges that collectively cover all sets in $\mathcal{F}$. Note that every instance of this problem induces a weighted set system $(\mathcal{F}, \mathcal{S}, c)$, where for each edge $e \in E$ there is an analogous subset $S_e \in \mathcal{S}$, consisting of all vertex sets $V_i \in \mathcal{F}$ covered by e. Laminar cover can be approximated by applying various techniques, most of which actually deal with the more general *tree augmentation* problem, and produce solutions whose cost is within factor 2 of optimum. We refer the reader to a short survey of these results [9, Sec. 1]. For the unweighted case, Nagamochi [27] proposed a $(1.875 + \epsilon)$-approximation for any fixed $\epsilon > 0$, a ratio that was later improved to $\frac{3}{2}$ by Even, Feldman, Kortsarz and Nutov [9].

In the *generalized partial laminar cover* problem, each $V_i \in \mathcal{F}$ is associated with a profit $p(V_i)$. The goal is to identify a minimum cost set of edges $E' \subseteq E$ such that the overall profit of the sets in $\mathcal{F}$ covered by E' is at least P, a specified profit bound. We are not aware of any approximability result for this problem, even for the seemingly simple case of unit profits. In the full version of this paper [23], we prove that $(\mathcal{F}, \mathcal{S}, c) \in \mathcal{I}_2$ for every weighted set system induced by a laminar cover instance, to obtain the following theorem.

Theorem 7. *The generalized partial laminar cover problem can be approximated within a factor of* $\frac{8}{3} + \epsilon$*, for any fixed* $\epsilon > 0$.

3.4 Totally Unimodular Cover and k-Interval Cover

The *element-set incidence matrix* $\mathcal{M}_U^{\mathcal{S}}$ of a set system $(U, \mathcal{S})$ has a row for every element $e \in U$ and a column for every set $S \in \mathcal{S}$; its entry in row e and column S is 1 when $e \in S$ and 0 otherwise. Totally unimodular cover (TUC) is a special case of the set cover problem in which $\mathcal{M}_U^{\mathcal{S}}$ is totally unimodular, that is, every square submatrix of this matrix has determinant 0, 1 or -1. We remark that although TUC is known to have integral LP solutions (see, for example, [7, Sec. 6.5]), this property does not extend to its partial covering version, which has not been explicitly studied yet. A particularly interesting problem captured by the latter variant is *partial bipartite vertex cover*: While the approximability of the unit-profit case is still open, arbitrary profits render the problem NP-hard, since it generalizes *minimum knapsack* even when the given graph is a star. We omit the straightforward reduction.

As illustrated in the full version of this paper [23], the prize-collecting set cover problem can be formulated as an integer program whose constraint matrix is $[\mathcal{M}_U^{\mathcal{S}}, I]$. Simple linear algebra arguments show that whenever $\mathcal{M}_U^{\mathcal{S}}$ is totally unimodular then so is $[\mathcal{M}_U^{\mathcal{S}}, I]$, implying that we obtain a 1-LMP algorithm by solving prize-collecting TUC to optimality as a linear program. The next theorem follows.

Theorem 8. *The generalized partial TUC problem can be approximated within a factor of* $\frac{4}{3} + \epsilon$*, for any fixed* $\epsilon > 0$.

We say that $\mathcal{M}_U^{\mathcal{S}}$ is a *k-interval matrix* if it contains at most k blocks of consecutive 1's in each row. The k-interval cover problem (k-IC) is a special case of set cover in which $\mathcal{M}_U^{\mathcal{S}}$ is a k-interval matrix. In the full version of this paper [23], we present a k-LMP rounding algorithm for the prize-collecting k-IC problem, that makes use of our 1-LMP algorithm for the corresponding variant of totally unimodular cover. We derive the following result as a corollary of Theorem 1.

Theorem 9. *The generalized partial k-IC problem can be approximated within a factor of* $(\frac{4}{3} + \epsilon)k$*, for any fixed* $\epsilon > 0$.

This provides, for instance, the first algorithm that approximates *partial rectangle stabbing* in $\mathbb{R}^d$, noting that the resulting factor of $(\frac{4}{3}+\epsilon)d$ nearly matches the d-approximation of Gaur, Ibaraki and Krishnamurti [16] for the full coverage version of this problem. In addition, we obtain an alternative, albeit non-combinatorial, $(\frac{4}{3}+\epsilon)f$-approximation for partial set cover with maximum element frequency f.

3.5 Edge Cover

Given an undirected graph $G = (V, E)$ with non-negative edge costs, *edge cover* is the problem of finding a minimum cost set of edges that contains at least one edge incident to each vertex. Clearly, this problem is equivalent to the special case of set cover in which each subset consists of exactly two elements. We note that edge cover is actually a matching problem in disguise, implying its polynomial time solvability [8, 26]. Plesník [30] proved that unit-profit partial edge cover, which is also known as the *k-edge cover* problem, can be solved to optimality by reducing it to standard edge cover. However, when arbitrary profits are allowed, this problem becomes NP-hard, as it generalizes minimum knapsack. Since Parekh [29, Sec. 2.3] suggested a polynomial-time algorithm for prize-collecting edge cover, we obtain the following theorem.

Theorem 10. *Generalized partial edge cover can be approximated within a factor of* $\frac{4}{3} + \epsilon$*, for any fixed* $\epsilon > 0$.

3.6 Multicut

On trees. The input to this problem consists of an edge-weighted tree $T = (V, E)$ and a collection of k distinct pairs of vertices, $\{s_1, t_1\}, \ldots, \{s_k, t_k\}$. The objective is to find a minimum cost set of edges whose removal from T disconnects each of the given pairs. It is important to note that, once again, we are facing a special case of set cover: The elements to cover are the input pairs, and an edge $e \in E$ covers $\{s_i, t_i\}$ if it resides on the unique path in T connecting s_i and t_i. Garg, Vazirani and Yannakakis [15] presented a primal-dual 2-approximation for this problem, which was also shown to be at least as hard to approximate as vertex cover.

The corresponding partial cover problem, in which we are required to disconnect a specified number of pairs, has recently been studied by Levin and Segev [24] and independently by Golovin et al. [17], who achieved an approximation guarantee of $\frac{8}{3}+\epsilon$, for any fixed $\epsilon > 0$. Since the former authors provide a 2-LMP algorithm for the prize-collecting multicut problem, we immediately obtain the following theorem, extending the factor of $\frac{8}{3} + \epsilon$ to the case of arbitrary profits.

Theorem 11. *When the underlying graph is a tree, the generalized partial multicut problem can be approximated within a factor of* $\frac{8}{3} + \epsilon$*, for any fixed* $\epsilon > 0$.

General graphs. When the input graph is no longer restricted to be a tree, the multicut problem becomes significantly harder to approximate. While Garg et al. [14] devised an $O(\log k)$-approximation using the region growing method, a hardness result of $\Omega(\log \log n)$ was given by Chawla, Krauthgamer, Kumar, Rabani and Sivakumar [5], assuming a stronger version of the Unique Games Conjecture [22]. Based on Räcke's hierarchical decomposition method [31], Alon, Awerbuch, Azar, Buchbinder and Naor [1] have shown how to simulate multicuts in general graphs by multicuts in the corresponding decomposition tree. As observed by Golovin et al. [17], this method extends to approximate the partial multicut problem within factor $O(\alpha \log^2 n \log \log n)$, given an α-approximation for the more restricted tree case. Their arguments can be easily combined with Theorem 11 to derive the next result for arbitrary profits.

Theorem 12. *On arbitrary graphs, the generalized partial multicut problem can be approximated within a factor of* $O(\log^2 n \log\log n)$.

References

1. N. Alon, B. Awerbuch, Y. Azar, N. Buchbinder, and J. Naor. A general approach to online network optimization problems. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 577–586, 2004.
2. E. Balas and M. Padberg. Set partitioning: A survey. *SIAM Review*, 18(4):710–760, 1976.
3. R. Bar-Yehuda. Using homogeneous weights for approximating the partial cover problem. *Journal of Algorithms*, 39(2):137–144, 2001.
4. N. H. Bshouty and L. Burroughs. Massaging a linear programming solution to give a 2-approximation for a generalization of the vertex cover problem. In *Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science*, pages 298–308, 1998.
5. S. Chawla, R. Krauthgamer, R. Kumar, Y. Rabani, and D. Sivakumar. On the hardness of approximating multicut and sparsest-cut. In *Proceedings of the 20th Annual IEEE Conference on Computational Complexity*, pages 144–153, 2005.
6. F. A. Chudak, T. Roughgarden, and D. P. Williamson. Approximate k-MSTs and k-Steiner trees via the primal-dual method and Lagrangean relaxation. *Mathematical Programming*, 100(2):411–421, 2004.
7. W. J. Cook, W. H. Cunningham, W. R. Pulleyblank, and A. Schrijver. *Combinatorial Optimization*. John Wiley and Sons, New York, 1997.
8. J. Edmonds and E. L. Johnson. Matching: A well-solved class of integer linear programs. In *Combinatorial Structures and their Applications*, pages 89–92. Gordon and Breach, New York, 1970.
9. G. Even, J. Feldman, G. Kortsarz, and Z. Nutov. A 3/2-approximation algorithm for augmenting the edge-connectivity of a graph from 1 to 2 using a subset of a given edge set. In *Proceedings of the 4th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems*, pages 90–101, 2001.
10. U. Feige. A threshold of $\ln n$ for approximating set cover. *Journal of the ACM*, 45(4):634–652, 1998.
11. T. Fujito. On approximation of the submodular set cover problem. *Operations Research Letters*, 25(4):169–174, 1999.
12. R. Gandhi, S. Khuller, and A. Srinivasan. Approximation algorithms for partial covering problems. *Journal of Algorithms*, 53(1):55–84, 2004.
13. R. S. Garfinkel and G. L. Nemhauser. Optimal set covering: A survey. In A. M. Geoffrion, editor, *Perspectives on Optimization*, pages 164–183, 1972.
14. N. Garg, V. V. Vazirani, and M. Yannakakis. Approximate max-flow min-(multi)cut theorems and their applications. *SIAM Journal on Computing*, 25(2):235–251, 1996.
15. N. Garg, V. V. Vazirani, and M. Yannakakis. Primal-dual approximation algorithms for integral flow and multicut in trees. *Algorithmica*, 18(1):3–20, 1997.
16. D. R. Gaur, T. Ibaraki, and R. Krishnamurti. Constant ratio approximation algorithms for the rectangle stabbing problem and the rectilinear partitioning problem. *Journal of Algorithms*, 43(1):138–152, 2002.
17. D. Golovin, V. Nagarajan, and M. Singh. Approximating the k-multicut problem. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 621–630, 2006.
18. D. S. Hochbaum. Approximating covering and packing problems: Set cover, vertex cover, independent set, and related problems. In D. S. Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, chapter 3, pages 94–143. PWS Publishing Company, 1997.

19. D. S. Hochbaum. The *t*-vertex cover problem: Extending the half integrality framework with budget constraints. In *Proceedings of the 1st International Workshop on Approximation Algorithms for Combinatorial Optimization*, pages 111–122, 1998.
20. D. S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9(3):256–278, 1974.
21. M. J. Kearns. *The Computational Complexity of Machine Learning*. MIT Press, 1990.
22. S. Khot. On the power of unique 2-prover 1-round games. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, pages 767–775, 2002.
23. J. Könemann, O. Parekh, and D. Segev. A unified approach to approximating partial covering problems, 2006. Available at `http://www.math.tau.ac.il/~segevd`.
24. A. Levin and D. Segev. Partial multicuts in trees. In *Proceedings of the 3rd International Workshop on Approximation and Online Algorithms*, pages 320–333, 2005.
25. L. Lovász. On the ratio of optimal integral and fractional covers. *Discrete Mathematics*, 13:383–390, 1975.
26. K. G. Murty and C. Perin. A 1-matching blossom type algorithm for edge covering problems. *Networks*, 12:379–391, 1982.
27. H. Nagamochi. An approximation for finding a smallest 2-edge-connected subgraph containing a specified spanning tree. *Discrete Applied Mathematics*, 126(1):83–113, 2003.
28. M. W. Padberg. Covering, packing and knapsack problems. *Annals of Discrete Mathematics*, 4:265–287, 1979.
29. O. Parekh. *Polyhedral Techniques for Graphic Covering Problems*. PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, 2002.
30. J. Plesník. Constrained weighted matchings and edge coverings in graphs. *Discrete Applied Mathematics*, 92(2–3):229–241, 1999.
31. H. Räcke. Minimizing congestion in general networks. In *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science*, pages 43–52, 2002.
32. P. Slavík. Improved performance of the greedy algorithm for partial cover. *Information Processing Letters*, 64(5):251–254, 1997.

Navigating Low-Dimensional and Hierarchical Population Networks*

Ravi Kumar[1], David Liben-Nowell[2], and Andrew Tomkins[1]

[1] Yahoo! Research, Sunnyvale, CA 94089, USA
[2] Department of Computer Science, Carleton College, Northfield, MN 55057, USA
{ravikumar, atomkins}@yahoo-inc.com, dlibenno@carleton.edu

Abstract. Social networks are *navigable small worlds*, in which two arbitrary people are likely connected by a short path of intermediate friends that can be found by a "decentralized" routing algorithm using only local information. We develop a model of social networks based on an arbitrary metric space of points, with population density varying across the points. We consider *rank-based friendships*, where the probability that person u befriends person v is inversely proportional to the number of people who are closer to u than v is. Our main result is that greedy routing can find a short path (of expected polylogarithmic length) from an arbitrary source to a randomly chosen target, independent of the population densities, as long as the doubling dimension of the metric space of locations is low. We also show that greedy routing finds short paths with good probability in tree-based metrics with varying population distributions.

1 Introduction

The last few years have witnessed increased interest in measuring, modeling, and exploiting *social networks*—collections of people connected by edges representing acquaintance, friendship, or other social relationships. Numerous internet startups have arisen predicating that one's social network requires the same careful husbandry as one's credit rating or investment portfolio. A common focus of scientific studies of social networks is the *small-world phenomenon*, the observation that most pairs of people are connected through short chains of friends. A remarkable experiment of Stanley Milgram [24] in the 1960s empirically validated this hypothesis, showing that two typical people in the United States were connected by a chain of acquaintances with an average length of six, thereby introducing the concept of "six degrees of separation" into popular culture. It is surprising that short paths exist, but it is remarkable that members of the network are able to discover these short paths with only information about their local neighborhood and some scant information about the destination [16].

* Part of this work was done while the second author was visiting Yahoo! Research. Thanks to David Barbella, Erik Demaine, George Kachergis, David Karger, Jon Kleinberg, Danny Krizanc, Jasmine Novak, Prabhakar Raghavan, Anna Sallstrom, and Ben Sowell for helpful comments and suggestions.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 480–491, 2006.

Thus, Milgram's experiment suggests not only that social networks have small diameter but also that they admit efficient "decentralized" search.

There have been a number of recent models of social networks that attempt to include an account of observed social-network properties like the small-world phenomenon. Watts and Strogatz [31] proposed a model consisting of two superimposed sets of edges, a structured set of edges and a smaller number of random edges. The former is meant to capture "typical" social friendships created, say, by geographic proximity. The latter is meant to capture long-range connections; these edges reduce the network's diameter, but do not explain navigability. Kleinberg [15, 16] proposed a simple model that suffices to produce a navigable small world. The underlying network is a k-dimensional grid, and each person occupies a unique grid location. As in the Watts–Strogatz model, the network has two kinds of links. Each person is connected via short-range links to her immediate neighbors in the grid, and she has one long-range link, chosen randomly so that the probability that a person u befriends a person v is proportional to $d(u, v)^{-\alpha}$, where $d(u, v)$ is the lattice distance between u and v, and $\alpha \geq 0$ is a parameter of the model. Kleinberg studied *greedy routing*—to route a message from s to t, person s sends the message to the neighbor of s who is closest in lattice distance to t—and showed that with high probability this simple algorithm finds paths of length polylogarithmic in the population size if and only if $\alpha = k$.

As with most models of complex real-world phenomena, Kleinberg's mathematically appealing model makes certain simplifying assumptions. In particular, it postulates that each grid point is occupied by a single individual, and hence the grid exhibits uniform population density—a significant deviation from most real-world populations. This issue has been addressed by two subsequent models, both designed to handle nonuniform population distributions without compromising analytical tractability. The first is Kleinberg's *group-structure* model, based on set systems [17]. The second is *rank-based friendship*, which we proposed in joint work with Jasmine Novak and Prabhakar Raghavan [21]. We showed empirically that the geographic distribution of friendships in the LiveJournal blogging community closely matches rank-based friendship when the population is modeled in two-dimensional Euclidean space, and we proved that short paths can be found in two-dimensional grids by greedy routing [19, 21].

In this paper, we focus on rigorous analysis of rank-based friendship in a wide variety of social-network settings. For intuition on this model, consider two people u and v who live 500 meters apart. In rural Minnesota, say, u and v are probably next-door neighbors, and are very likely to know each other; in Manhattan, there may be more than 10,000 people who live closer to u than v does, and u and v have probably never met. This discrepancy suggests why distance alone is insufficient as the basis for a model of real friendships. Instead, we model long-range links using a notion of "rank": the *rank* of a person v with respect to u is the number of people who live at least as close to u as v does. Thus, the closest candidate friend to u has rank 1, the next one has rank 2, and so forth. In rank-based friendship, the probability that u befriends v is inversely proportional to the rank of v with respect to u. (Intuitively, to be befriended by

u, person v will have to compete with all of the more "convenient" candidate friends for u, i.e., all people w who live closer to u than v does.) An important feature of this model is that it implicitly accounts for the dimensionality of the space in which people live. For example, in a k-dimensional grid of uniform population density, for any k, the rank-based formulation induces exactly the unique value of α proven by Kleinberg to generate a navigable network.

Our contributions. In this paper, we study the properties of rank-based friendship. Our main theorem shows that greedy routing discovers short paths for arbitrary (not necessarily uniform) population densities on any metric space of low doubling dimension. We extend the main theorem in two directions. First, we present a recursive formulation of a population in which it is possible to route a message to the city of Manhattan, then to the appropriate block of the city, and finally to the floor of the building where the target person lives. We show that the theorem holds even for recursive structures of polynomial depth. Next, we analyze greedy routing and rank-based friendship under tree metrics, which have been proposed to capture non-geographic proximity between individuals in a social network [17, 30]. (One can naturally model the proximity of two hobbies or occupations, e.g., through a tree.) Through a more detailed analysis, we are able to remove the notion of "local neighbors" entirely in this case.

In Sect. 2, we present background material on metric spaces, population networks, and routing algorithms, and we define a notion of a social structure, which can be formed from an arbitrary metric space. Points in the metric space correspond to locations, at which many people may reside. In Sect. 3, we formally define rank-based friendship, and we construct social networks with long-range links generated via rank-based friendship. We also define a general notion of short-range links based on the metric space. In Sect. 4, we present our main result on greedy routing in these social networks. Specifically, we show that greedy routing finds a path from an arbitrary source person to the location of a target person chosen uniformly at random from the population. The expected length of this path is polylogarithmic in the size of the population and exponential in the doubling dimension of the metric space. Thus, if the underlying metric space has low doubling dimension—like a constant-dimensional grid—greedy routing yields expected polylogarithmic paths. We then turn to our two extensions. In Sect. 5, we describe recursive population networks and analyze greedy routing in these networks; due to potentially polynomial-depth leaves in the tree of locations, we must adopt a more complex measure of progress towards the target. Finally, in Sect. 6, we investigate tree social structures, in which the points of the population network are leaves of a tree.

Other related work. There have been several relevant extensions to Kleinberg's original model, which we review here. In k-dimensional grids, there has also been considerable work on upper and lower bounds for the diameter and the length of the greedy path (e.g., [4, 23, 26]), and partially decentralized algorithms other than greedy routing have also been considered [9, 20, 22, 23, 27]. Kleinberg has extended his model to tree-based structures and group structures [17]. Fraigniaud

analyzes circumstances under which a graph can be augmented to be navigable, and gives a positive answer in the case of bounded treewidth or bounded chordality [8]. Analysis of navigability in (uniform-population) networks of low doubling dimension has been performed by Duchon et al. [7] and Slivkins [28]. Broadly speaking, these papers give stronger bounds on network navigability than the present work, but are limited to uniform populations; the treatment of nonuniform population distributions is the major contribution of this paper.

In previous work with J. Novak and P. Raghavan, we defined rank-based friendship and referenced a technical report that includes a theorem regarding population networks on two-dimensional grids [19, 21]. The current paper gives a significantly more general analysis of rank-based friendship, and subsumes the particular theorem contained in that technical report. Notions similar to rank-based friendships can also be found in geometric data structures [5, 10].

The question of searching in social networks was also considered by Adamic et al. [1–3] and Kim et al. [13], and Milgram's experiment was replicated in a larger-scale email setting by Dodds et al. [6]. For a comprehensive treatment of social networks, good sources include the book by Wasserman and Faust [29] and a recent survey by Kleinberg [18].

2 Preliminaries

Background on metric spaces. Let $\langle X, d\rangle$ be a metric space. Denote by $B_r(x) := \{y \in X : d(x,y) < r\}$ the open radius-r ball around point $x \in X$. Define the *aspect ratio* as $\Delta := \max_{x,y\in X} d(x,y)/\min_{x,y\in X, x\neq y} d(x,y)$. The *doubling dimension* of $\langle X, d\rangle$ is the smallest α such that, for every $r > 0$ and every $Y \subseteq X$ of diameter $2r$, Y can be covered by at most 2^α subsets of diameter r.

Social structures and population networks. A *social network* is a graph $\langle P, E\rangle$, where a node represents a person and an edge represents a friendship between its endpoints. Edges are directed to allow nonreciprocal friendships. Let $\Gamma(u)$ denote the out-neighbors of $u \in P$.

A *social structure* is a quadruple $\langle L, d, P, \mathsf{loc}\rangle$, where L is a finite set of *points*; $d : L \times L \to \mathbb{R}^{\geq 0}$ is a *distance metric* on the points (so $\langle L, d\rangle$ is a metric space); P is an ordered finite set of *people*; and $\mathsf{loc} : P \to L$ is the *location function*, which maps people to the point in which they live. For convenience, we assume that d is scaled so that $\min_{\ell,\ell'\in L,\ell\neq\ell'} d(\ell,\ell') = 1$. Extend d to $d : (P \cup L) \times (P \cup L) \to \mathbb{R}^{\geq 0}$ where $d(u,\cdot) := d(\mathsf{loc}(u),\cdot)$ and $d(\cdot, v) := d(\cdot, \mathsf{loc}(v))$ for all people $u, v \in P$. We use the ordering on P to break ties in comparing distances: for people $u, v, v' \in P$, write $d(u,v) < d(u,v')$ as shorthand for $\langle d(u,v), v\rangle \prec_{lexicographic} \langle d(u,v'), v'\rangle$, where the ordering on the second component is given by the ordering on P. This tie-breaking role is the only purpose of the ordering on people.

A *population network* is a quintuple $\langle L, d, P, \mathsf{loc}, E\rangle$ where $\langle L, d, P, \mathsf{loc}\rangle$ is a social structure and $E \subseteq P \times P$ is a set of *friendships*. (Thus $\langle P, E\rangle$ is a social network.) Let $\mathsf{pop}(\ell) := |\{u \in P : \mathsf{loc}(u) = \ell\}|$ denote the *population* of $\ell \in L$. Extend pop so that $\mathsf{pop}(L') := \sum_{\ell\in L'} \mathsf{pop}(\ell)$ for a subset $L' \subseteq L$ of the points.

Write $n := \mathsf{pop}(L) = |P|$ for the total population. Let $\mathsf{dens} : L \to [0,1]$ be a probability distribution denoting the *population density* of each point $\ell \in L$, so that $\mathsf{dens}(\ell) := \mathsf{pop}(\ell)/n$. As before, we extend dens to allow us to write $\mathsf{dens}(L') := \sum_{\ell \in L'} \mathsf{dens}(\ell)$ for $L' \subseteq L$.

Routing algorithms. Given a population network $\langle L, d, P, \mathsf{loc}, E\rangle$, a *source individual* $s \in P$, and a *target individual* $t \in P$, a *routing algorithm* seeks a path $\rho = \langle u_0, u_1, \ldots, u_k\rangle$ from $s = u_0$ to $t = u_k$ in the graph $\langle P, E\rangle$.

We are interested in routing algorithms that compute the next step u_{i+1} from the current person u_i without taking the entire graph $\langle P, E\rangle$ as input. The algorithm is *decentralized* if, when computing the next step u_{i+1} in the path, the only information used is u_i, t, the social structure $\langle L, d, P, \mathsf{loc}\rangle$, and the set of neighbors $\Gamma(u_i)$ of the current node u_i. (That is, the edges in E excluding those incident to u_i are not available as input to the decentralized algorithm.) In this paper, we focus on one particular decentralized algorithm: the *greedy algorithm* Greedy selects $u_{i+1} := \mathrm{argmin}_{v \in \Gamma(u_i)} d(v, t)$.

3 Rank-Based Friendship

For two people $u, v \in P$, the *rank of v with respect to u* is the number of people $w \in P$ who are closer to u than v is. Formally, this quantity is given by $\mathsf{rank}_u(v) := |\{w \in P : d(u,w) < d(u,v)\}|$, where we break ties in distance from person $u \in P$ using the linear ordering on P so that, for any $i \in \{1, \ldots, n\}$ and any person $u \in P$, there is exactly one person v such that $\mathsf{rank}_u(v) = i$.

A *rank-based* friendship for a person $u \in P$ is one generated as follows: a friend v is chosen randomly for u according to the probability distribution $\mathtt{Pr}[u \text{ links to } v] \propto 1/\mathsf{rank}_u(v)$. For any person $u \in P$, we have $\sum_v 1/\mathsf{rank}_u(v) = \sum_{i=1}^{n} 1/i = H_n$, the nth harmonic number. Therefore, by normalizing, we have

$$\mathtt{Pr}[\text{a particular rank-based link from } u \text{ links to } v] = 1/(H_n \cdot \mathsf{rank}_u(v)). \quad (1)$$

Up to constant factors, this rank-based formulation gives the same link probabilities as Kleinberg's distance-based model for a uniform-population k-dimensional mesh. Thus Kleinberg's results [16] immediately imply that rank-based friendship produces a navigable grid for a uniformly distributed population:

Theorem 1. *Let $\langle L, d, P, \mathsf{loc}\rangle$ be a social structure where L is a k-dimensional mesh for $k = \Theta(1)$, d is the Manhattan (L_1) distance, and we have a uniform population P in which exactly one person lives at each point on the grid. Endow each person in the network with $2k$ "local" friends (the immediate neighbors in each cardinal direction) and one "long-range" friend, chosen according to rank-based friendship. Then, with high probability, the length of the* Greedy *path from any $s \in L$ to any $t \in L$ is $O(\log^2 n)$.*

In this paper, we will consider networks with more complicated metrics on the points. To do so, we will need a generalization of the $2k$ "local" neighbors from Theorem 1. For a social structure $\langle L, d, P, \mathsf{loc}\rangle$, construct a population network $\langle L, d, P, \mathsf{loc}, E\rangle$ by generating friendships as follows:

- Endow each person $p \in P$ with δ rank-based links, chosen according to (1).
- Endow each person $p \in P$ with "local neighbors," as follows. Let $G = \langle L, E_G \rangle$ be a graph where shortest paths correspond to the metric d—i.e., the shortest ℓ-to-ℓ' path in G has length $d(\ell, \ell')$. For $\ell \in L$, let $\Gamma_G(\ell)$ be the neighbors of ℓ in G. For every person $p \in P$ with $\mathsf{loc}(p) = \ell$ and for every $\ell' \in \Gamma_G(\ell)$, choose an arbitrary q such that $\mathsf{loc}(q) = \ell'$ and add the edge $\langle p, q \rangle$ to E.

We refer to a network satisfying the latter condition as a *neighbor-connected* network. Neighbor connectivity ensures that, for any s and any t, the first step taken by $\mathsf{Greedy}(s, t)$ will be to a person u such that $d(u, t) < d(s, t)$. Among other things, this condition guarantees that every person encountered by Greedy is encountered only once. Thus we can invoke the Principle of Deferred Decisions in our analysis (see [25]): we proceed as if the long-range links of each person are generated only once the greedy algorithm encounters that person. Furthermore, the greedy algorithm never gets "stuck"; a person u fails to link to a person v such that $d(v, t) < d(u, t)$ only if $\mathsf{loc}(u) = \mathsf{loc}(t)$. (Notice also that neighbor connectivity requires that every point has strictly positive population.)

4 Routing in Networks with Low Doubling Dimension

Let $\langle L, d, P, \mathsf{loc} \rangle$ be an arbitrary social structure, where $n := |P|$. Let α and Δ, respectively, be the doubling dimension and aspect ratio of $\langle L, d \rangle$. We derive a neighbor-connected degree-δ rank-based population network $\langle L, d, P, \mathsf{loc}, E \rangle$ by endowing each person $p \in P$ with "local" neighbors as required to achieve neighbor connectivity and δ rank-based friends. In this section, we show that greedy routing finds a short path to a target location whenever α is small.

Lemma 2 (Greedy quickly (in expectation) halves distance to target). *For arbitrary $s \in P$ and $t \in P$ chosen uniformly at random from P, the expected number of rank-based links examined before $\mathsf{Greedy}(s, t)$ reaches a person in $B_{d(s,t)/2}(\mathsf{loc}(t))$ is $O(\log n \cdot \log \Delta \cdot 2^{O(\alpha)})$, where the expectation is taken over both the random construction of the network and the random choice of t.*

Proof sketch. An *r-net*, for any $r > 0$, is a set $S \subseteq L$ such that (i) for all $x \in L$, there is some $s \in S$ with $d(x, s) < r$; and (ii) for all distinct $s, s' \in S$, we have $d(s, s') \geq r$. An r-net can be greedily constructed for any $r > 0$. Let $\mathscr{R} := \{1, 2, 4, \ldots, 2^{2+\lceil \log \Delta \rceil}\}$. For every $r \in \mathscr{R}$, we define a set of balls of radius r, where the set $\mathscr{C}_r$ of ball centers forms an $(r/2)$-net. Let r_t denote the minimum $r \in \mathscr{R}$ such that $s, t \in B_r(s')$ for some $s' \in \mathscr{C}_r$. We show that $2r_t \geq d(s, t)/2 \geq r_t/8$; thus it will suffice to show that the expected number of links examined before $\mathsf{Greedy}(s, t)$ lands in $B_{r_t/8}(t) \subseteq B_{d(s,t)/2}(t)$ is $O(\log n \cdot \log \Delta \cdot 2^{O(\alpha)})$.

Suppose that $\mathsf{Greedy}(s, t)$ has generated a partial path from s, where the last element of the path so far is some person $u \in P$. Each step taken by Greedy decreases the distance from the current point to the target t, so we have that $d(u, t) \leq d(s, t) \leq 2r_t$.

We refer to a link from u as *good$_t$* if it connects u to any person living in the ball $B_{r_t/8}(t)$. Let $\beta_{u,t}$ denote the probability that a particular link from u is

good$_t$. We show that there is a point $z_t \in \mathscr{C}_{16r_t}$ such that $B_{8r_t}(t) \subseteq B_{16r_t}(z_t)$, and that $\beta_{u,t} \geq \mathsf{pop}(B_{r_t/8}(t))/(\mathsf{pop}(B_{16r_t}(z_t)) \cdot H_n)$, independent of u. Define $\beta_t := \mathsf{pop}(B_{r_t/8}(t))/(\mathsf{pop}(B_{16r_t}(z_t)) \cdot H_n)$. Thus the probability that a particular link from u is good$_t$ is at least β_t for every person u along the Greedy path, and is independent at each step. Therefore, the expected number of links examined by Greedy before we reach a good$_t$ link (or t itself) is at most $1/\beta_t$, where the expectation is taken over the random construction of the network.

We now examine the expected value of $1/\beta_t$ when t is chosen uniformly at random from the population. We show that

$$\begin{aligned}\mathtt{E}_t[1/\beta_t] &\leq H_n \cdot \sum\nolimits_{x \in L} \mathsf{dens}(x) \cdot \mathsf{dens}(B_{16r_x}(z_x))/\mathsf{dens}(B_{r_x/16}(z'_x)) \\ &\leq H_n \cdot \sum_{r \in \mathscr{R}, z \in \mathscr{C}_{16r}} \sum_{z' \in \mathscr{C}_{r/16} : z' \in B_{16r}(z)} \frac{\mathsf{dens}(B_{16r}(z))}{\mathsf{dens}(B_{r/16}(z'))} \sum_{x \in B_{r/16}(z')} \mathsf{dens}(x),\end{aligned}$$

where the second line follows by reindexing the summation to be over radii and ball centers from the appropriate r-nets rather than over target locations x. From this, we obtain

$$\mathtt{E}_t[1/\beta_t] \leq H_n \cdot \sum\nolimits_{r \in \mathscr{R}} \sum\nolimits_{z \in \mathscr{C}_{16r}} \mathsf{dens}(B_{16r}(z)) \cdot |\{z' \in \mathscr{C}_{r/16} : z' \in B_{16r}(z)\}|.$$

Using properties of r-nets, we are able to show that the inner summation is upper bounded by $2^{O(\alpha)}$, independent of r. Thus, the expectation is upper bounded by $H_n \cdot |\mathscr{R}| \cdot 2^{O(\alpha)}$, which is $O(\log n \cdot \log \Delta \cdot 2^{O(\alpha)})$ by definition of $\mathscr{R}$. □

Theorem 3. *Let $\langle L, d, P, \mathsf{loc}, E\rangle$ be a neighbor-connected degree-δ rank-based population network. Let $s \in P$ be arbitrary, and let $t \in P$ be chosen uniformly at random. Then the expected length of the* Greedy(s,t) *path from s to* $\mathsf{loc}(t)$ *is $O(\max\{\log \Delta, \log n \cdot \log^2 \Delta \cdot 2^{O(\alpha)}/\delta\})$, where the expectation is taken both over the random construction of the network and over the random choice of t.*

As a corollary, in the k-dimensional mesh under L_1 distance, where each person has δ rank-based friends and $2k$ local friends, for an arbitrary source s and a uniformly chosen target t, the expected length of Greedy$(s, \mathsf{loc}(t))$ is $O(\log^3 n \cdot 2^{O(k)}/\delta)$, which is just $O(\log^3 n)$ when $\delta = \Omega(1)$ and $k = O(1)$.

5 Recursive Population Networks

In this section, we describe a recursive model of population networks that allows higher resolution of location, and that allows the routing of messages to an *individual*, rather than just to that individual's city or town.

Recursive social structures. In the model described previously, a point $\ell \in L$ represents a collection of collocated individuals. Here, we extend the model so that each $\ell \in L$ represents either a single individual or a substructure refining distances between ℓ's inhabitants.

A *recursive social structure (RSS)* is the following: we have a social structure consisting of people living at various points, with a distance function describing the separation between points. For each point ℓ in which strictly more than one person lives, we have, recursively, a social structure for the people living in point ℓ. Formally, an RSS σ on a nonempty set P of people is given as follows:

- If $|P| = 1$, then σ is simply the lone individual in P.
- If $|P| \geq 2$, then $\sigma = \langle L, d, P, \mathsf{loc}, M \rangle$, where $\langle L, d, P, \mathsf{loc} \rangle$ is a social structure with $|L| \geq 2$ and $\mathsf{pop}(\ell) \geq 1$ for every $\ell \in L$, and, for every $\ell \in L$, we have that $M(\ell) = \sigma_\ell$ is an RSS on the set of people $P_\ell := \{u \in P : \mathsf{loc}(u) = \ell\}$.

For an RSS σ, define a tree $\mathscr{T}(\sigma)$ of social structures, where each social structure $\langle L, d, P, \mathsf{loc} \rangle$ contained in σ has "child structures" for each point $\ell \in L$ with $|P_\ell| \geq 2$. The leaves of the tree are the points with a single resident. Let $\mathscr{M}(\sigma)$ denote the internal nodes in $\mathscr{T}(\sigma)$. For $N \in \mathscr{T}(\sigma)$, let $\mathsf{depth}(N)$ denote the depth of N in the tree $\mathscr{T}(\sigma)$, and let $\mathsf{depth}(\mathscr{T}(\sigma))$ denote the depth of the deepest leaf in $\mathscr{T}(\sigma)$. (The root of $\mathscr{T}(\sigma)$ has depth one.)

For $u \in P$, let $N_u = u$ denote the leaf of $\mathscr{T}(\sigma)$ where u is the lone person. For a structure $N \in \mathscr{M}(\sigma)$, we write $u \in N$ to denote that N_u is in the subtree of $\mathscr{T}(\sigma)$ rooted at N—i.e., that $u \in P_N$ where $N = \langle L_N, d_N, P_N, \mathsf{loc}_N \rangle$. Write $\mathsf{depth}(u) := \mathsf{depth}(N_u)$, and for any $1 \leq i \leq \mathsf{depth}(u)$, write $\mathsf{structure}_i(u)$ to denote the unique structure at depth i in $\mathscr{T}(\sigma)$ such that $u \in \mathsf{structure}_i(u)$. Finally, for two individuals $u, v \in P$, let $\mathsf{LCA}(u, v)$ denote the least common ancestor of u and v in $\mathscr{T}(\sigma)$—i.e., the smallest-population structure N in $\mathscr{T}(\sigma)$ such that $u, v \in N$.

From an RSS σ on a set P of people, we derive a (standard) social structure $\mathscr{S}(\sigma)$, where the distances between people are derived from σ. Because the leaves of $\mathscr{T}(\sigma)$ are just the people of P, there will be a unique location in $\mathscr{S}(\sigma)$ for each person of P. To derive distances $d_\sigma(u, v)$ in σ, we consider only the coarsest-resolution structure N in which u and v live in distinct points. Formally, let $N := \mathsf{LCA}(u, v)$, where $N = \langle L_N, d_N, P_N, \mathsf{loc}_N \rangle$. (Note that $u, v \in P_N$ and that $\mathsf{loc}_N(u) \neq \mathsf{loc}_N(v)$.) We define $d(u, v) := \langle -\mathsf{depth}(N), d_N(\mathsf{loc}_N(u), \mathsf{loc}_N(v)) \rangle$, and we use standard lexicographic ordering on pairs to compare distances.

Recursive population networks. Given an RSS σ on a set P of people, we can generate a *recursive population network (RPN)* $\rho = \langle \sigma, E \rangle$ by endowing the people of P with friendships. Let $d = d_\sigma$ be the derived distance function as described above. (We will abuse notation and write $\mathscr{T}(\rho) := \mathscr{T}(\sigma)$, etc.) In a degree-$\delta$ rank-based RPN, we endow each person in P with δ long-range links, chosen according to (1). We assume that ties in distance are broken randomly for the purposes of generating rank-based friendships.

As before, we introduce local neighbors to guarantee (minute) progress from any source to any target t. (The condition is similar to the one introduced in Section 4, but slightly more complicated.) Let $h : P \to \mathbb{R}$ be a function assigning a "social height" to the people in the network. Consider any $N \in \mathscr{T}(\sigma)$ where $N = \langle L_N, d_N, P_N, \mathsf{loc}_N \rangle$, and let P_ℓ denote the set of people living in point $\ell \in L_N$. For a person $p \in P_\ell$, consider the following conditions:

1. Person p has a local link to a person $q \in P_\ell$ so that $h(q) > h(p)$.
2. Suppose that the metric d_N on L_N is a shortest-path metric in a graph $G = \langle L_N, E_N \rangle$. For every point ℓ' for which the edge $\langle \ell, \ell' \rangle \in E_N$, there exists a q where $\mathsf{loc}_N(q) = \ell'$ such that p has a local link to person q.

If every person in an RPN ρ satisfies one of these two conditions for every structure in $\mathscr{M}(\sigma)$, then we say that ρ is *neighbor connected*.

We also add a tie-breaking rule to Greedy using the social-height function. Suppose that source s seeks a greedy path to a target $t \neq s$, and there is no friend u of s such that $d(u,t) < d(s,t)$. (Thus s cannot fall into Case 2 of the definition of neighbor connectivity for the structure $N = \mathsf{LCA}(s,t)$.) The next step in the Greedy path is a neighbor u of s such that $d(s,t) = d(u,t)$ and $h(u) > h(s)$. This tie-breaking rule guarantees that Greedy can "lift" itself out of a substructure to reach a target in a different structure.

For *non-local* ties in distance—i.e., s has two distinct friends u, v such that $d(u,t) = d(v,t) < d(s,t)$—we assume that ties are broken uniformly at random.

Routing on rank-based RPNs. We will prove that Greedy finds short paths in expectation in any neighbor-connected rank-based RPN derived from an RSS σ as long as the maximum doubling dimension of $N \in \mathscr{M}(\sigma)$ is small.

Notice the following fact, which follows immediately by definition of $d = d_\sigma$: for any $u \in P$ and any depth $i \leq \mathsf{depth}(u)$, all people in $\mathsf{structure}_i(u)$ are closer to u than any person outside $\mathsf{structure}_i(u)$ is to u. An immediate consequence of this fact is that the path found by Greedy aiming for a target t will never leave $\mathsf{structure}_i(t)$ once it enters this subtree.

The expected time required to reach a target t drawn uniformly from the population P is bounded by $O(\max\{\log \Delta, \log n \cdot \log^2 \Delta \cdot 2^{O(\alpha)}/\delta\} \cdot \mathsf{depth}(\mathscr{T}(\sigma)))$, by Theorem 3: in expectation we reach the target point in any particular structure in $O(\max\{\log \Delta, \log n \cdot \log^2 \Delta \cdot 2^{O(\alpha)}/\delta\})$ steps, and we must find the correct point $\mathsf{depth}(t)$ times before we have arrived at the target person herself. In the following, we remove the dependence on $\mathsf{depth}(\mathscr{T}(\sigma))$.

Theorem 4. *Let ρ be an arbitrary degree-δ rank-based neighbor-connected RPN with $n = |P|$ people, maximum doubling dimension α, and maximum aspect ratio Δ. For an arbitrary source person $s \in P$ and a target person $t \in P$ chosen uniformly at random from P, we have that the expected length of the* Greedy *path from s to t is $O(\max\{\log \Delta, \log^2 \Delta \cdot \log n \cdot 2^{O(\alpha)}/\delta\} \cdot \min\{\mathsf{depth}(\mathscr{T}(\rho)), \log n\})$.*

Proof sketch. Our proof proceeds by showing that within a polylogarithmic number of steps we will reduce by a factor of two the number of people closer to the target than the current person on the greedy path is. Let $N_{\mathsf{LCA}} := \mathsf{LCA}(s,t)$, and let $P_{\mathsf{LCA}} := \mathsf{pop}(N_{\mathsf{LCA}})$ be its population. In the structure N_{LCA}, we begin at some point ℓ_s and we wish to reach some point ℓ_t. There are two cases to consider. If $\mathsf{pop}(\ell_t) \leq |P_{\mathsf{LCA}}|/2$ (i.e., the subpopulation containing the target is not too big), then simply reaching ℓ_t as per Theorem 3 constitutes considerable progress towards the target. If $\mathsf{pop}(\ell_t) > |P_{\mathsf{LCA}}|/2$, then any node encountered on the Greedy path has a probability $\Omega(1/H_n)$ of linking to one of the $|P_{\mathsf{LCA}}|/2$

people closest to t. Thus in $O(\log n)$ steps with high probability we reach one of the $|P_{\mathsf{LCA}}|/2$ people closest to t, which is also considerable progress towards the target. In either case, we have reduced by a factor of two the number of people closer to the target than the current person on the greedy path; a logarithmic number of repetitions of this process will find the target individual herself.

To formalize the above intuitive argument, consider running Greedy starting from person s until the completion of the following two-phase operation:

Phase 1 ("Halfway there"): Run Greedy starting from s until we reach a person v such that either (i) $v \in \mathsf{structure}_{\mathsf{depth}(t)-1}(t)$—i.e., the structure that directly contains the target t—or (ii) $\mathsf{rank}_t(v) \leq \mathsf{pop}(\mathsf{LCA}(s,t))/2$.

Phase 2 ("One level deeper"): Run Greedy starting from v until we reach a person w such that either $w = t$ or $\mathsf{depth}(\mathsf{LCA}(w,t)) > \mathsf{depth}(\mathsf{LCA}(v,t))$.

We show the following:

- After Phase 2 has ended, either $w = t$ or $\mathsf{pop}(\mathsf{LCA}(w,t)) \leq \mathsf{pop}(\mathsf{LCA}(s,t))/2$.
- The expected number of steps before we complete a single two-phase operation is $O(\max\{\log \Delta, \log^2 \Delta \cdot \log n \cdot 2^{O(\alpha)}/\delta\})$.

Thus after a logarithmic number of repetitions of the two-phase process—or $\mathsf{depth}(\mathscr{T}(\rho))$ repetitions, if that quantity is smaller—we reach the target t. □

6 Routing in Trees

We now turn to *tree social structures* $\langle L, d, P, \mathsf{loc}\rangle$, where the elements of L are the leaves of a k-ary tree T. We abuse notation and also let T denote the nodes of this tree. Let $T[r]$ denote the subtree of T rooted at $r \in T$. We restrict d so that, for every point $x \in L$ and every node r that is an ancestor of x in T, the point x is closer to every node in $T[r]$ than it is to any node outside of $T[r]$. The population P consists of an arbitrary set of n people, and $\mathsf{loc} : P \to L$ is an arbitrary location function. In particular, we do not impose the condition that $\mathsf{pop}(\ell)$ be strictly positive for every $\ell \in L$; we can simply treat zero-population leaves as not appearing in the tree. If each person in a k-ary tree social structure is endowed with δ edges chosen according to rank-based friendship, then we refer to the resulting population network as a *rank-based δ-degree k-ary tree population network*. Proofs of the following are omitted due to space constraints.

Lemma 5. *Fix an arbitrary $s \in P$. Fix any internal node $r \in T$ such that $\mathsf{loc}(s) \in T[r]$. Choose $t \in P$ uniformly at random from $\{t : \mathsf{loc}(t) \in T[r]\}$. Let r_t denote the child of r such that $\mathsf{loc}(t) \in T[r_t]$. Then, with probability at least $1 - \frac{(k-1)\cdot H_n}{e\cdot\delta}$, within one step the path from s to t found by* Greedy *reaches $T[r_t]$.*

Theorem 6. *Let $\langle L, d, P, \mathsf{loc}, E\rangle$ be a rank-based δ-degree k-ary tree population network. Fix an arbitrary $\eta \geq 1$. If the degree δ satisfies $\delta \geq \eta \cdot k \cdot H_n \cdot \mathsf{depth}(T)/e$, then the following holds with probability at least $1 - 1/\eta$: for an arbitrary source person $s \in P$ and a target person $t \in P$ chosen uniformly at random from P, the* Greedy *path from s to $\mathsf{loc}(t)$ has length at most $\mathsf{depth}(T)$.*

As a corollary, consider a rank-based population network derived from a binary tree with depth $O(\log^k n)$ and with degree $\delta = \Omega(\eta \cdot \log^{k+1} n)$. Then with probability at least $1 - 1/\eta$, for arbitrary $s \in P$ and uniformly chosen $t \in P$, the length of the Greedy path from s to $\mathsf{loc}(t)$ has length $O(\log^k n)$.

7 Discussion and Future Work

Here we highlight some interesting open questions for future study, focusing on the model described in Sections 4 and 5. We have shown that $\mathsf{E}_t[|\mathsf{Greedy}(s,t)|] = \text{polylog}(|P|)$ for any $s \in P$ in rank-based networks. In contrast, Kleinberg has shown that, for uniform populations, with high probability, $\mathsf{Greedy}(s,t)$ has polylogarithmic length for any s and for any t when link probabilities are chosen according to the correct distance-based distribution. There may be population distributions for which the "for all t" condition cannot be achieved in our context, perhaps if there is a recluse who is very unlikely to be reached by long-range links. It remains open whether Greedy finds a short expected path for *any* target.

We use the assumption that there are no empty locations in our network to guarantee that Greedy never gets "stuck" at a person u without a local neighbor closer to the target than u herself is. Investigating the limitations of Greedy in a model with zero-population locations (like lakes and deserts in the real world) is an intriguing direction, and would eliminate the most unrealistic limitation in our model. Geographic routing via local-information algorithms in general, and geographic routing around obstacles in particular, has been previously considered in the wireless-networking community [11, 12, 14]. It is an interesting question as to whether these results, where there is typically a technologically inspired threshold on the geographic distance that a message can traverse in a single hop, can be adapted to the social-network setting.

A number of partially decentralized algorithms (e.g., [9, 20, 22, 23, 27]) have been shown to outperform Greedy theoretically or experimentally; it would be interesting to analyze them in rank-based networks. More generally, our results can be viewed as extending Kleinberg's theorem to a dimension-independent model that allows varying population density (and one that holds in real networks [21]). There have been some recent theoretical results extending and refining Kleinberg's result—for example, considering routing on other types of underlying graphs [7, 8, 28], among other results [4, 23, 26]—and we might hope to be able to make analogous improvements to our results.

References

1. L. Adamic, E. Adar. How to search a social network. *Social Networks*, 27(3):187–203, 2005.
2. L. Adamic, R. Lukose, B. Huberman. Local search in unstructured networks. In *Handbook of Graphs and Networks*. Wiley-VCH, 2002.
3. L. Adamic, R. Lukose, A. Puniyani, B. Huberman. Search in power-law networks. *Physical Review Letters E*, 64(046135), 2001.

4. L. Barrière, P. Fraigniaud, E. Kranakis, D. Krizanc. Efficient routing in networks with long range contacts. In *Proc. Intl. Conf. on Distr. Comp.*, 2001.
5. E. Demaine, J. Iacono, S. Langerman. Proximate point searching. *Computational Geometry: Theory and Applications*, 28(1):29–40, 2004.
6. P. Dodds, R. Muhamad, D. Watts. An experimental study of search in global social networks. *Science*, 301:827–829, 2003.
7. P. Duchon, N. Hanusse, E. Lebhar, N. Schabanel. Could any graph be turned into a small world? *Theoretical Computer Science*, 355(1):96–103, 2006.
8. P. Fraigniaud. Greedy routing in tree-decomposed graphs. In *Proc. Eur. Symp. Alg.*, 2005.
9. P. Fraigniaud, C. Gavoille, C. Paul. Eclecticism shrinks even small worlds. In *Proc. Symp. on Princ. of Distr. Comp.*, 2004.
10. J. Iacono, S. Langerman. Proximate planar point location. In *Proc. Symp. on Comp. Geom.*, 2003.
11. B. Karp. *Geographic Routing for Wireless Networks.* PhD thesis, Harvard, 2000.
12. B. Karp, H. Kung. GPSR: Greedy perimeter stateless routing for wireless networks. In *Proc. Intl. Conf. on Mobile Computing and Networking*, 2000.
13. B. Kim, C. Yoon, S. Han, H. Jeong. Path finding strategies in scale-free networks. *Physical Review Letters E*, 65(027103), 2002.
14. Y. Kim, R. Govindan, B. Karp, S. Shenker. Geographic routing made practical. In *Proc. Symp. on Networked Systems Design and Impl.*, 2005.
15. J. Kleinberg. Navigation in a small world. *Nature*, 406:845, 2000.
16. J. Kleinberg. The small-world phenomenon: An algorithmic perspective. In *Proc. Symp. Theory of Comp.*, 2000.
17. J. Kleinberg. Small-world phenomena and the dynamics of information. In *Advances in Neural Information Processing*, 2001.
18. J. Kleinberg. Complex networks and decentralized search algorithms. In *Proc. International Congress of Mathematicians*, 2006.
19. R. Kumar, D. Liben-Nowell, J. Novak, P. Raghavan, A. Tomkins. Theoretical analysis of geographic routing in social networks. TR MIT-CSAIL-TR-2005-040.
20. E. Lebhar, N. Schabanel. Close to optimal decentralized routing in long-range contact networks. In *Proc. Intl. Colloq. on Automata, Lang., and Prog.*, 2004.
21. D. Liben-Nowell, J. Novak, R. Kumar, P. Raghavan, A. Tomkins. Geographic routing in social networks. *Proc. Natl. Acad. Sciences*, 102(33):11623–11628, 2005.
22. G. Manku, M. Naor, U. Wieder. Know thy neighbor's neighbor: the power of lookahead in randomized P2P networks. In *Proc. Symp. Theory of Comp.*, 2004.
23. C. Martel, V. Nguyen. Analyzing Kleinberg's (and other) small-world models. In *Proc. Symp. on Princ. of Distr. Comp.*, 2004.
24. S. Milgram. The small world problem. *Psychology Today*, 1:61–67, 1967.
25. R. Motwani, P. Raghavan. *Randomized Algorithms.* Cambridge Univ. Press, 1995.
26. V. Nguyen, C. Martel. Analyzing and characterizing small-world graphs. In *Proc. Symp. on Disc. Alg.*, 2005.
27. O. Şimşek, D. Jensen. A probabilistic framework for decentralized search in networks. In *Proc. Intl. Joint Conf. on AI*, 2005.
28. A. Slivkins. Distance estimation and object location via rings of neighbors. In *Proc. Symp. on Princ. of Distr. Comp.*, 2005.
29. S. Wasserman, K. Faust. *Social Network Analysis.* Cambridge Univ. Press, 1994.
30. D. Watts, P. Dodds, M. Newman. Identity and search in social networks. *Science*, 296:1302–1305, 2002.
31. D. Watts, S. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, 1998.

Popular Matchings in the Capacitated House Allocation Problem

David F. Manlove* and Colin T.S. Sng

Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, UK
Fax: +44(0) 141 330 4913
{davidm, sngts}@dcs.gla.ac.uk

Abstract. We consider the problem of finding a popular matching in the *Capacitated House Allocation problem* (CHA). An instance of CHA involves a set of agents and a set of houses. Each agent has a preference list in which a subset of houses are ranked in strict order, and each house may be matched to a number of agents that must not exceed its capacity. A matching M is *popular* if there is no other matching M' such that the number of agents who prefer their allocation in M' to that in M exceeds the number of agents who prefer their allocation in M to that in M'. Here, we give an $O(\sqrt{C}n_1 + m)$ algorithm to determine if an instance of CHA admits a popular matching, and if so, to find a largest such matching, where C is the total capacity of the houses, n_1 is the number of agents and m is the total length of the agents' preference lists. For the case where preference lists may contain ties, we give an $O((\sqrt{C} + n_1)m)$ algorithm for the analogous problem.

1 Introduction

An instance I of the *Capacitated House Allocation problem* (CHA) comprises a bipartite graph $G = (A, H, E)$, where $A = \{a_1, a_2, ..., a_{n_1}\}$ is the set of *agents*, $H = \{h_1, h_2, ..., h_{n_2}\}$ is the set of *houses* and E is the set of edges in G. We let $n = n_1 + n_2$ and $m = |E|$. Each agent $a_i \in A$ ranks in strict order a subset of the set of houses (the *acceptable* houses for a_i) represented by his/her *preference list*. Each house $h_j \in H$ has a *capacity* $c_j \geq 1$ which indicates the maximum number of agents that may be matched to it. We assume that $m \geq \max\{n_1, n_2\}$, i.e. no agent has an empty preference list and each house is acceptable to at least one agent. We also assume that $c_j \leq n_1$ for each $h_j \in H$. Let $C = \sum_{j=1}^{n_2} c_j$ denote the sum of the capacities of the houses.

A *matching* M in I is a subset of E such that (i) each agent is matched to at most one house in M, and (ii) each house $h_j \in H$ is matched to at most c_j agents in M. If an agent $a_i \in A$ is matched in M, we denote by $M(a_i)$ the house that a_i is matched to in M. We define $M(h_j)$ to be the set of agents matched to h_j in M (thus $M(h_j)$ could be empty). Given two matchings M and M' in I, we say that an agent a_i *prefers* M' to M if either (i) a_i is matched in M' and

* Supported by EPSRC grant GR/R84597/01 and RSE/Scottish Executive Personal Research Fellowship.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 492–503, 2006.

(a)
a_1: h_1 h_2 h_3
a_2: h_1 h_2 h_3
a_3: h_1 h_2 h_3

(b)
a_1: h_1 h_2
a_2: h_1

Fig. 1. Two instances of HA

unmatched in M, or (ii) a_i is matched in both M' and M and prefers $M'(a_i)$ to $M(a_i)$. Let $P(M', M)$ denote the set of agents who prefer M' to M. Then, M' is *more popular than* M if $|P(M', M)| > |P(M, M')|$, i.e. the number of agents who prefer M' to M is greater than the number of agents who prefer M to M'. Furthermore, a matching M in I is *popular* if there is no other matching M' in I that is more popular than M.

CHA is an example of a bipartite matching problem with one-sided preferences [1, 2, 7, 3]. These problems have applications in areas such as campus housing allocation in US universities [1], hence the problem name; in assigning probationary teachers to their first posts in Scotland; and in Amazon's DVD rental service. A variety of optimality criteria have been defined for such problems. Gärdenfors [6] first introduced the notion of a popular matching (also known as a *majority assignment*) in the context of voting theory. Alternatively, *Pareto optimality* [1, 2] is often regarded by economists as a fundamental property to be satisfied. A matching M is *Pareto optimal* if there is no matching M' such that some agent prefers M' to M, and no agent prefers M to M'. Finally, a matching is *rank maximal* [7] if it assigns the maximum number of agents to their first-choice houses, and subject to this, the maximum number of agents to their second-choice houses, and so on. However, Pareto optimal matchings and rank maximal matchings need not be popular.

Popular matchings were considered by Abraham et al. [3] in the context of the *House Allocation problem* (HA) – the special case of CHA in which each house has capacity 1. They gave an instance of HA in which no popular matching exists (see Figure 1(a)) and also noted that popular matchings can have different sizes (see Figure 1(b); in this HA instance the matchings $M_1 = \{(a_1, h_1)\}$ and $M_2 = \{(a_1, h_2), (a_2, h_1)\}$ are both popular). Abraham et al. [3] described an $O(n+m)$ algorithm for finding a maximum cardinality popular matching (henceforth a maximum popular matching) if one exists, given an instance of HA. They also described an $O(\sqrt{n}m)$ counterpart for the *House Allocation problem with Ties* (HAT) – the generalisation of HA in which agents' preferences may include ties.

Several other recent papers have also focused on popular matchings. Mahdian [8] gave some probabilistic results with respect to the existence of popular matchings in a random instance of HA. Abraham and Kavitha [4] considered popular matchings in a dynamic matching market in which agents and houses can enter and leave the market, and showed that there exists a 2-step *voting path* to compute a new popular matching from some initial matching after every such change, provided some popular matching exists. Also Mestre [10] studied a generalisation of the problem in which agents have a weight indicating their priority, and the objective is to compute a *weighted popular matching* M (i.e. there is no other matching M' such that the weighted majority of the agents prefer M' to M.)

In this paper, we consider popular matchings in instances of CHA and CHAT, where CHAT denotes the *Capacitated House Allocation problem with Ties* – the generalisation of CHA in which agents' preference lists may contain ties. Both CHA and CHAT are natural generalisations of the one-one HA and HAT models considered in [3] to the case where houses may have non-unitary capacity. We extend the characterisations and algorithms for popular matchings from [3] to these many-one settings. In particular, in Section 2, we develop a characterisation of popular matchings in a CHA instance I, and then use it to construct an $O(\sqrt{C}n_1 + m)$ algorithm for finding a maximum popular matching in I if one exists. In Section 3, we build a new characterisation of popular matchings in a CHAT instance I, and then use it to construct an $O((\sqrt{C}+n_1)m)$ algorithm for finding a maximum popular matching in I if one exists.

We finally remark that a straightforward solution to each of the problems of finding a maximum popular matching, given an instance of CHA or CHAT, may be to use "cloning". Informally, this entails creating c_j clones for each house h_j, to obtain an instance $C(I)$ of HAT (i.e. each house has capacity 1), and then applying the HAT algorithm of [3] to $C(I)$. However, we will show in Sections 2 and 3 that this method in general leads to slower algorithms than the direct approach that we will be using in each case.

2 Popular Matchings in CHA

Characterising Popular Matchings. Let I be an instance of CHA. For each agent $a_i \in A$, let $f(a_i)$ denote the first-ranked house on a_i's preference list. Any such house h_j is called an *f-house*. For each $h_j \in H$, let $f(h_j) = \{a_i \in A : f(a_i) = h_j\}$ and $f_j = |f(h_j)|$ (possibly $f_j = 0$). Now let M be a matching in I. We say that a house $h_j \in H$ is *full* if $|M(h_j)| = c_j$, and *undersubscribed* if $|M(h_j)| < c_j$. We also create a unique *last resort* house $l(a_i)$ with capacity 1 for each agent $a_i \in A$, and append $l(a_i)$ to a_i's preference list. The following lemma is a vital first step in characterising popular matchings in I.

Lemma 1. *Let M be a popular matching in I. Then for every f-house h_j, $|M(h_j) \cap f(h_j)| = \min\{c_j, f_j\}$.*

Proof. We consider the following two cases.

– *Case (i):* Suppose $f_j \leq c_j$. We will show that $f(h_j) \subseteq M(h_j)$. For, suppose not. Then choose any $a_r \in f(h_j) \backslash M(h_j)$. We consider the subcases that (a) h_j is undersubscribed and (b) h_j is full. In subcase (a), promote a_r to h_j to obtain a more popular matching than M. In subcase (b), choose any $a_s \in M(h_j) \backslash f(h_j)$. Let $h_k = f(a_s)$. Then $h_k \neq h_j$. If h_k is undersubscribed, promote a_r to h_j and promote a_s to h_k to obtain a more popular matching than M. Otherwise, choose any $a_t \in M(h_k)$. We then promote a_r to h_j, promote a_s to h_k and demote a_t to $l(a_t)$ to obtain a more popular matching than M.

– *Case (ii):* Suppose $f_j > c_j$. If h_j is undersubscribed, then $f(h_j) \not\subseteq M(h_j)$ so there exists some $a_r \in f(h_j) \backslash M(h_j)$ that we can promote to h_j to obtain a more popular matching as in Case (i)(a). Hence, h_j is full. Now, suppose for a

contradiction that $M(h_j) \not\subseteq f(h_j)$. Then there exists some $a_s \in M(h_j) \backslash f(h_j)$. As $f_j > c_j$, it follows that $f(h_j) \not\subseteq M(h_j)$ so there exists some $a_r \in f(h_j) \backslash M(h_j)$. The remainder of the argument follows Case (i)(b).

Hence the following properties hold for the new matching. If $f_j \leq c_j$, then $f(h_j) \subseteq M(h_j)$. Otherwise, $M(h_j) \subseteq f(h_j)$ and $|M(h_j)| = c_j$. Thus, the condition in the statement of the lemma is now satisfied. □

For each agent a_i, we next define $s(a_i)$ to be the most-preferred house h_j on a_i's preference list such that either (i) h_j is a non-f-house, or (ii) h_j is an f-house such that $h_j \neq f(a_i)$ and $f_j < c_j$. Note that $s(a_i)$ must exist in view of $l(a_i)$. We refer to such a house h_j as an *s-house*. We remark that the set of f-houses need not be disjoint from the set of s-houses. It may be shown that a popular matching M will only match an agent a_i to either $f(a_i)$ or $s(a_i)$, as indicated by the next two lemmas (see [9] for the proofs).

Lemma 2. *Let M be a popular matching in I. Then no agent $a_i \in A$ can be matched in M to a house between $f(a_i)$ and $s(a_i)$ on a_i's preference list.*

Lemma 3. *Let M be a popular matching in I. Then no agent $a_i \in A$ can be matched in M to a house worse than $s(a_i)$ on a_i's preference list.*

Let $G = (A, H, E)$ be the underlying graph of I. We form a subgraph G' of G by letting G' contain only two edges for each agent a_i, that is, one to $f(a_i)$ and the other to $s(a_i)$. We say that a matching M is *agent-complete* in a given graph if it matches all agents in the graph. Clearly, in view of last resort houses, all popular matchings must be agent-complete in G'. However, G' need not admit an agent-complete matching if $s(a_i) \neq l(a_i)$ for some agent a_i. In conjunction with Lemmas 1-3, the graph G' gives rise to the following characterisation of popular matchings in I.

Theorem 1. *A matching M is popular in I if and only if*

1. *for every f-house h_j,*
   (a) *if $f_j \leq c_j$, then $f(h_j) \subseteq M(h_j)$;*
   (b) *if $f_j > c_j$, then $|M(h_j)| = c_j$ and $M(h_j) \subseteq f(h_j)$.*
2. *M is an agent-complete matching in the reduced graph G'.*

Proof. By Lemmas 1-3, any popular matching necessarily satisfies Conditions 1 and 2. We now show that these conditions are sufficient.

Let M by any matching satisfying Conditions 1 and 2 and suppose for a contradiction that M' is a matching that is more popular than M. Let a_i be any agent that prefers M' to M and let $h_k = M'(a_i)$. Since M is an agent-complete matching in G', and since G' contains only edges from a_i to $f(a_i)$ and $s(a_i)$, then $M(a_i) = s(a_i)$. Hence either (i) $h_k = f(a_i)$ or (ii) h_k is an f-house such that $h_k \neq f(a_i)$ and $f_k \geq c_k$, by definition of $s(a_i)$.

In Case (i), if $f_k < c_k$ then by Condition 1(a), $a_i \in M(h_k)$, a contradiction. Hence in both Cases (i) and (ii), $f_k \geq c_k$. In each of the cases that $f_k = c_k$ and $f_k > c_k$, it follows by Conditions 1(a) and 1(b) that $|M(h_k)| = c_k$ and $M(h_k) \subseteq$

```
1.    M := ∅;
2.    for each f-house h_j
3.        c'_j := c_j;
4.        if f_j ≤ c_j
5.            for each a_i ∈ f(h_j)
6.                M := M ∪ {(a_i, h_j)};
7.                delete a_i and its incident edges from G';
8.            c'_j := c_j − f_j;
9.    remove all isolated and full houses, and their incident edges, from G';
10.   compute a maximum matching M' in G' using capacities c'_j;
11.   if M' is not agent-complete in G'
12.       output "no popular matching exists"
13.   else
14.       M := M ∪ M';
15.       for each a_i ∈ A
16.           h_j := f(a_i);
17.           if f_j > c_j and |M(h_j)| < c_j and h_j ≠ M(a_i)
18.               promote a_i from M(a_i) to h_j in M;
```

Fig. 2. Algorithm Popular-CHA for finding a popular matching in CHA

$f(h_k)$. Since h_k is full in M, it follows that $|M(h_k)\backslash M'(h_k)| \geq |M'(h_k)\backslash M(h_k)|$. Hence for every a_i who prefers $M'(a_i) = h_k$ to $M(a_i)$, there is a unique $a_j \in M(h_k)\backslash M'(h_k)$. But as $a_j \in M(h_k)$, it follows that $h_k = f(a_j)$. Hence a_j prefers $M(a_j)$ to $M'(a_j)$. Therefore, M is popular in I. □

Finding a Popular Matching. Theorem 1 leads to Algorithm Popular-CHA for finding a popular matching in a CHA instance I, or reporting that none exists, as shown in Figure 2. The algorithm begins by using a pre-processing step (lines 2-9) on G' that matches agents to their first-choice house h_j whenever $f_j \leq c_j$, so as to satisfy Condition 1(a) of Theorem 1.

Our next step computes a maximum cardinality matching M' (henceforth a maximum matching) in G', according to the adjusted house capacities c'_j that are defined following pre-processing. The subgraph G' can be viewed as an instance of the Upper Degree-Constrained Subgraph problem (UDCS) [5]. (An instance of UDCS is essentially the same as an instance of CHA, except that agents have no explicit preferences in the UDCS case; the definition of a matching is unchanged.) We use Gabow's algorithm [5] to compute M' in G' and then test whether M' is agent-complete. The pre-allocations are then added to M' to give M. As a last step, we ensure that M also meets Condition 1(b) of Theorem 1. For, suppose that $h_j \in H$ is an f-house such that $f_j > c_j$. Then by definition, h_j cannot be an s-house. Thus if $a_k \in M(h_j)$ prior to the third for loop, it follows that $a_k \in f(h_j)$. At this stage, if h_j is undersubscribed in M, we repeatedly promote any agent $a_i \in f(h_j)\backslash M(h_j)$ from $M(a_i)$ (note that $M(a_i)$ must be $s(a_i)$ and hence cannot be an f-house h_l such that $f_l > c_l$) to h_j until h_j is full, ensuring that $M(h_j) \subseteq f(h_j)$.

It is clear that the reduced graph G' of G can be constructed in $O(m)$ time.

The graph G' has $O(n_1)$ edges since each agent has degree 2 in G'. Clearly each of the pre- and post-processing steps involving the three for loop phases takes $O(n_1 + n_2)$ time. The complexity of Gabow's algorithm [5] for computing M' in G' is $O(\sqrt{C}n_1)$. Hence we obtain the following result concerning the complexity of Algorithm Popular-CHA.

Lemma 4. *Given an instance of CHA, we can find a popular matching, or determine that none exists, in* $O(\sqrt{C}n_1 + m)$ *time.*

It remains to consider the problem of finding a maximum popular matching in I. We begin by dividing the set of all agents into disjoint sets. Let A_1 be the set of all agents a_i with $s(a_i) = l(a_i)$, and let $A_2 = A - A_1$. We aim to find a matching M that satisfies the conditions of Theorem 1, and that minimises the number of A_1-agents who are matched to their last resort house.

We begin by constructing G', and carrying out the pre-processing step in lines 2-9 of Algorithm Popular-CHA on all agents in $A_1 \cup A_2$. We then try to find a maximum matching M' in G' that only involves the A_2-agents that remain after pre-processing and their incident edges. If M' is not an agent-complete matching of the agents in A_2 that remain after pre-processing, then G admits no popular matching by Theorem 1. Otherwise, we remove all edges in G' that are incident to a last resort house, and try to match A_1-agents to their first-choice houses. At each step, we try to match an additional A_1-agent to his/her first-choice house by finding an augmenting path with respect to M' using Gabow's algorithm for UDCS [5], so that we have a maximum matching of agents in $A_1 \cup A_2$ in G' at the end of this process. If any A_1-agent remains unmatched, we simply assign him/her to his/her last resort house, to obtain an agent-complete matching in G'. We also ensure that Condition 1(b) of Theorem 1 is met by executing the third for loop in Algorithm Popular-CHA. Clearly then, the matching so obtained, together with the pre-assignments from earlier, is a maximum popular matching, giving the following theorem.

Theorem 2. *Given an instance of CHA, we can find a maximum popular matching, or determine that none exists, in* $O(\sqrt{C}n_1 + m)$ *time.*

An alternative approach to our algorithm would be to use cloning. Given an instance I of CHA, we may obtain an instance J of HAT by creating c_j clones $h_j^1, h_j^2, ..., h_j^{c_j}$ of each house h_j in I, where each clone has a capacity of 1. In addition, we replace each occurrence of h_j in a given agent's preference list with the sequence $h_j^1, h_j^2, ..., h_j^{c_j}$, the elements of which are listed in a single tie at the point where h_j appears. We can then apply the $O(\sqrt{n}m)$ algorithm for HAT given by [3] to J in order to find a maximum popular matching in I.

We now compare the worst-case complexity of the above cloning approach with that of our algorithm. The underlying graph G_J of J contains $n' = n_1 + C$ nodes. Let $c_{min} = \min\{c_j : h_j \in H\}$, and for $a_i \in A$, let A_i denote the set of acceptable houses for a_i. Then the number of edges in G_J is $m' = \sum_{a_i \in A} \sum_{h_j \in A_i} c_j \geq mc_{min}$. Hence the complexity of applying the algorithm given by [3] to J is $\Omega(\sqrt{C}mc_{min})$. Recall that the complexity of Algorithm

Popular-CHA is $O(\sqrt{C}n_1+m)$. It follows that the cloning method is slower by a factor of $\Omega(\sqrt{C}c_{min})$ or $\Omega(mc_{min}/n_1)$ (note that $m \geq n_1$ and $c_{min} \geq 1$) according as $\sqrt{C}n_1 \leq m$ or $\sqrt{C}n_1 > m$ respectively. In the case that $c_{min} = \Omega(n_1)$, our approach offers an improvement by a factor of $\Omega(n_1^{3/2}n_2^{1/2})$ or $\Omega(m)$ respectively.

3 Popular Matchings in CHAT

In this section, we generalise the characterisation of popular matchings together with Algorithm Popular-CHA as given in the previous section to the case that I is an instance of CHAT.

Characterising Popular Matchings. Let M be a popular matching in I. For each agent $a_i \in A$, let $f(a_i)$ denote the set of first-ranked houses on a_i's preference list (clearly it is possible that $|f(a_i)| > 1$ in view of ties in the preference lists). We refer to all such houses h_j as *f-houses* and we let $f(h_j) = \{a_i \in A : h_j \in f(a_i)\}$. Let $G = (A, H, E)$ be the underlying graph of I. Define $E_1 = \{(a_i, h_j) : a_i \in A \wedge h_j \in f(a_i)\}$ to be the set of *first-choice edges*. We define the *first-choice graph* of G as $G_1 = (A, H, E_1)$. For instances with strict preference lists, Lemma 1 implies that $M \cap E_1$ is a maximum matching in G_1. As the next lemma indicates (see [9] for the proof), this latter condition also extends to the CHAT case.

Lemma 5. *Let M be a popular matching in I. Then $M \cap E_1$ is a maximum matching in G_1.*

As Lemma 1 no longer holds in general in a CHAT instance, we work towards a new definition of s-houses by using some concepts from the theory of bipartite matching. Let M be a maximum matching in some bipartite graph G where all nodes have capacity 1. According to the Edmonds-Gallai Decomposition (see [11]), then the nodes of G can be partitioned into three disjoint sets: $\mathcal{E}$, $\mathcal{O}$ and $\mathcal{U}$. Nodes in $\mathcal{E}$, $\mathcal{O}$ and $\mathcal{U}$ are called *even, odd,* and *unreachable* respectively. A node v is even (odd) if there exists an alternating path of even (odd) length from an unmatched node in G to v. If no such alternating path exists, v is unreachable. Some fundamental properties of this node labelling (henceforth referred to as the *EOU labelling*) in relation to a maximum matching in G are summarised in Lemma 3.2 of [3].

Our aim is to obtain an EOU labelling of G_1 relative to a maximum matching M_1 of G_1 (as obtained by Gabow's algorithm [5], for example). However Lemma 3.2 of [3] applies directly only to the case where each node in the given bipartite graph has capacity 1. We obtain an EOU labelling of nodes in G_1 by a cloning process, as follows. The *cloned graph* $C(G_1)$ can be constructed from G_1 by replacing every house $h_j \in H$ with the clones $h_j^1, h_j^2, \ldots, h_j^{c_j}$. We then divide the capacity of each house among its clones by allowing each clone to have capacity 1. In addition, if $(a_i, h_j) \in G_1$, then we add $(a_i, h_j^k) \in C(G_1)$ for all k $(1 \leq k \leq c_j)$. We then adapt the maximum matching M_1 in G_1 to obtain a matching $C(M_1)$ in $C(G_1)$, as follows. If a house h_j in G_1 is matched to x_j agents $a_{i_1}, \ldots a_{i_{x_j}}$ in

M_1, then we add (a_{i_k}, h_j^k) to $C(M_1)$ for $1 \leq k \leq x_j$, so that $|C(M_1)| = |M_1|$ and $C(M_1)$ is a maximum matching in $C(G_1)$.

We next use $C(M_1)$ and $C(G_1)$ to obtain an EOU labelling of the nodes in $C(G_1)$, and hence G_1. Clearly, such a labelling in $C(G_1)$ is useful only if it can give a well-defined characterisation of EOU labels in G_1. Crucial to this is the need for the clones corresponding to each house $h_j \in H$ to have the same EOU label in $C(G_1)$, as stated by the next lemma (see [9] for the proof).

Lemma 6. *Let G_1 be the first-choice graph in I and let M_1 be a maximum matching in G_1. Define the cloned graph $C(G_1)$ and its corresponding maximum matching $C(M_1)$ as above. Then, given any house $h_j \in H$, any two clones of h_j in $C(G_1)$ have the same EOU label.*

We now use Lemma 6 to obtain an EOU labelling of the nodes in G_1. Clearly, in view of Lemma 6, a well-defined EOU labelling of $h_j \in H$ can be obtained by letting h_j inherit its EOU label from those of its clones. That is, we say that h_j is even, odd or unreachable in G_1 if its clones are even, odd or unreachable in $C(G_1)$ respectively. It is immediate that each agent can inherit its EOU label in G_1 from its corresponding label in $C(G_1)$. The next result is a consequence of Lemma 6 (see [9] for the proof).

Lemma 7. *Let M be a popular matching in I. Then every odd or unreachable house $h_j \in H$ satisfies $|M(h_j)| = c_j$ and $M(h_j) \subseteq f(h_j)$.*

Lemmas 6 and 7 give us the following analogue of Lemma 3.2 from [3] for CHAT.

Lemma 8. *Let G_1 be the first-choice graph in I and let M_1 be a maximum matching in G_1. Define $\mathcal{E}$, $\mathcal{O}$ and $\mathcal{U}$ to be the node sets corresponding to even, odd and unreachable nodes in an EOU labelling of G_1 with respect to M_1. Then:*

(a) The sets $\mathcal{E}$, $\mathcal{O}$ and $\mathcal{U}$ are pairwise disjoint. Every maximum matching in G_1 partitions the nodes into the same sets of even, odd and unreachable nodes.
(b) Every maximum matching M in G_1 satisfies the following properties:
  (i) every odd agent is matched to an even house in M;
  (ii) every odd house is full in M and matched only to even agents in M;
  (iii) every unreachable agent is matched to an unreachable house in M;
  (iv) every unreachable house is full in M and matched only to unreachable agents in M;
  (v) $|M| = |\mathcal{O}_A| + |\mathcal{U}_A| + \sum_{h_j \in \mathcal{O}_H} c_j$, where $\mathcal{U}_A$ is the set of unreachable agents, $\mathcal{O}_A$ is the set of odd agents and $\mathcal{O}_H$ is the set of odd houses.
(c) No maximum matching in G_1 contains an edge between two nodes in $\mathcal{O}$ or a node in $\mathcal{O}$ with a node in $\mathcal{U}$. There is no edge in G_1 connecting a node in $\mathcal{E}$ with a node in $\mathcal{U}$, or between two nodes of $\mathcal{E}$.

We are now in a position to define $s(a_i)$, the set of houses such that, in a popular matching M, if $a_i \in A$ is matched in M and $M(a_i) \notin f(a_i)$, then $M(a_i) \in s(a_i)$. We will ensure that any odd or unreachable house h_j is not a member of $s(a_i)$, since $|M(h_j)| = c_j$ and $M(h_j) \subseteq f(h_j)$ by Lemma 7. Hence,

we define $s(a_i)$ to be the set of highest-ranking houses in a_i's preference list that are even in G_1. Any such house is called an *s-house*. Clearly, it is possible that $|s(a_i)| > 1$, however, a_i is indifferent between all houses in $s(a_i)$. Furthermore, $s(a_i) \neq \emptyset$ due to the existence of last resort houses which are of degree 0 in G_1 (and thus even). However, $f(a_i)$ and $s(a_i)$ need not be disjoint. It turns out that Lemmas 2 and 3 also extend to CHAT as established by the following lemmas (see [9] for the proofs).

Lemma 9. *Let M be a popular matching in I. Then no agent $a_i \in A$ can be matched in M to a house between $f(a_i)$ and $s(a_i)$ on a_i's preference list.*

Lemma 10. *Let M be a popular matching in I. Then no agent $a_i \in A$ can be matched in M to a house worse than $s(a_i)$ on a_i's preference list.*

As was the case with CHA, we can also define a subgraph G' for the CHAT instance I by this time letting G' contain only edges from each agent a_i to houses in $f(a_i) \cup s(a_i)$. Clearly, all popular matchings must be agent-complete in G' in view of last resort houses. However, an agent-complete matching need not exist if $s(a_i) \neq \{l(a_i)\}$ for some agent a_i. Lemmas 5, 9 and 10 give rise to the following characterisation of popular matchings in I.

Theorem 3. *A matching M is popular in I if and only if*

1. *$M \cap E_1$ is a maximum matching in G_1, and*
2. *M is an agent-complete matching in the subgraph G'.*

Proof. By Lemmas 5, 9 and 10, any popular matching necessarily satisfies Conditions 1 and 2. We now show that these conditions are sufficient.

Let M be any matching satisfying Conditions 1 and 2. Suppose for a contradiction that M' is a matching that is more popular than M. Let a_i be any agent that prefers M' to M. Since a_i prefers $M'(a_i)$ to $M(a_i)$, M is an agent-complete matching in G', and G' only contains edges from a_i to $f(a_i) \cup s(a_i)$, it follows that $M(a_i) \in s(a_i)$, and $f(a_i)$ and $s(a_i)$ are disjoint. Hence, $M'(a_i)$ must be an odd or unreachable house in G_1, as $M(a_i)$ is the highest-ranked even house in a_i's preference list.

Let $h_{j_1} = M'(a_i)$. Since h_{j_1} is odd or unreachable, it follows by Condition 1 and Lemma 8(b) that $|M(h_{j_1})| = c_{j_1}$ and $M(h_{j_1}) \subseteq f(h_{j_1})$. Now since $a_i \in M'(h_{j_1}) \backslash M(h_{j_1})$, there exists a distinct agent $a_{k_1} \in M(h_{j_1}) \backslash M'(h_{j_1})$. If a_{k_1} is unmatched in M' or $M'(a_{k_1}) \notin f(a_{k_1})$, then a_{k_1} prefers M to M'. Otherwise, suppose $M'(a_{k_1}) \in f(a_{k_1})$. Let $h_{j_2} = M'(a_{k_1})$. Clearly, a_{k_1} is even or unreachable so that h_{j_2} must be odd or unreachable. It follows by Condition 1 and Lemma 8(b) that $|M(h_{j_2})| = c_{j_2}$ and $M(h_{j_2}) \subseteq f(h_{j_2})$. Hence, there exists an agent $a_{k_2} \neq a_{k_1}$ such that $a_{k_2} \in M(h_{j_2}) \backslash M'(h_{j_2})$ and $h_{j_2} \in f(a_{k_2})$. If a_{k_2} is unmatched in M' or $M'(a_{k_2}) \notin f(a_{k_2})$, then a_{k_2} prefers M to M'. Otherwise, suppose that $M'(a_{k_2}) \in f(a_{k_2})$. Let $h_{j_3} = M'(a_{k_2})$. Then there exists an agent $a_{k_3} \in M(h_{j_3}) \backslash M'(h_{j_3})$ by a similar argument for a_{k_2}. Note that possibly $h_{j_3} = h_{j_1}$, but we must be able to choose $a_{k_3} \neq a_{k_1}$, for otherwise $|M'(h_{j_1})| > |M(h_{j_1})|$, which is a contradiction since $|M(h_{j_1})| = c_{j_1}$. Thus, a_{k_3} is a distinct agent, so that

1. Build subgraph $G_1=(A, H, E_1)$, where $E_1=\{(a_i, h_j) : a_i \in A \wedge h_j \in f(a_i)\}$.
2. Compute a maximum matching M_1 of first-choice edges in G_1.
3. Obtain an EOU labelling of G_1 using $C(G_1)$ and $C(M_1)$.
4. Build subgraph $G'=(A, H, E')$, where $E'=\{(a_i, h_j) : a_i \in A \wedge h_j \in f(a_i) \cup s(a_i)\}$.
5. Delete all edges in G' connecting two odd nodes, or connecting an odd node with an unreachable node. (This step does not delete an edge of M_1.)
6. Find a maximum matching M in the reduced graph G' by augmenting M_1.
7. If M is not agent-complete in G', then output "No popular matching exists", otherwise return M as a popular matching in I.

Fig. 3. Algorithm Popular-CHAT for finding a popular matching in CHAT

we can repeat the above argument to identify an alternating path P in which houses need not be distinct, but agents are distinct. Clearly, P must terminate at some agent a_{k_r} as the number of agents are finite. Furthermore, it must be the case that a_{k_r} is unmatched in M' or $M'(a_{k_r}) \notin f(a_{k_r})$ so that for every a_i that prefers M' to M, there must exist a distinct a_{k_r} that prefers M to M'.

Finally, we note the uniqueness of a_{k_r}. If there exists another agent a'_i who prefers M' to M, then we can build another alternating path – it is possible that some of the houses are those already used in previous alternating paths such as P. However, it must be the case (from our argument that a_{k_3} is a distinct agent) that we are always able to identify distinct agents not already used in previous alternating paths, as each house on the path is odd or unreachable, and thus full in M. Hence, M is popular in I. □

Finding a Popular Matching. Theorem 3 leads to Algorithm Popular-CHAT for finding a popular matching in I of CHAT or reporting that none exists, as shown in Figure 3. The next lemma is an important step in establishing the correctness of the algorithm.

Lemma 11. *Algorithm Popular-CHAT constructs a matching M such that $M \cap E_1$ is a maximum matching of G_1.*

Proof. (Sketch – see [9] for the full proof.) Firstly, we claim that only first-choice edges are incident to odd nodes and unreachable houses in G' at the end of Step 4, using our definition of s-houses and Lemma 8(b). Define a *second-choice* edge as belonging to the edge set $\{(a_i, h_j) \in E' : h_j \in s(a_i) \wedge s(a_i) \not\subseteq f(a_i)\}$. By the claim, and by Lemma 8(c), the only first-choice edges in G' after Step 5 are those between (i) odd agents and even houses, (ii) even agents and odd houses, and (iii) unreachable agents and unreachable houses; the only second-choice edges are those between (i) even agents and even houses, and (ii) unreachable agents and even houses. Moreover no edge of M_1 is deleted by Step 5 of the algorithm. It follows that in Step 6, odd agents must remained matched to their first-choice houses, and by an argument involving alternating paths, unreachable agents cannot become worse off. Only even agents may become worse off, so that at least $|\mathcal{O}_A| + |\mathcal{U}_A| + \sum_{h_j \in \mathcal{O}_H} c_j$ first-choice edges are matched in the matching M. It thus follows by Lemma 8(b) that $M \cap E_1$ is a maximum matching of G_1. □

Hence if Algorithm Popular-CHAT returns a matching M, then M is both an agent-complete matching in G' and $M \cap E_1$ is a maximum matching of G_1 by Lemma 11. Hence M is a popular matching in I by Theorem 3.

We now consider the complexity of Algorithm Popular-CHAT. Let F be the number of first-choice edges in G, and let $c_{max} = \max\{c_j : h_j \in H\}$; then $c_{max} \leq n_1$. Clearly G_1 can be constructed in $O(F+n_2)$ time. We use Gabow's algorithm [5] to compute a maximum matching M_1 in G_1 in $O(\sqrt{C}F)$ time. We next use $C(M_1)$ in $C(G_1)$ to compute an EOU labelling of G_1. The total number of edges in $C(G_1)$ is $O(c_{max}F)$. We first use a pre-processing step to label each unmatched agent and each undersubscribed house as even. Clearly, this step takes $O(n)$ time. Next, breadth-first search may be used on $C(G_1)$ to search for alternating paths with respect to $C(M_1)$, building up odd or even labels for every node encountered. This step labels all odd and even (matched) agents, and all odd and even (full) houses and takes $O(c_{max}F + n_2)$ time. Any remaining unlabelled nodes must be unreachable and we can directly label these nodes in G_1 in $O(n)$ time. Thus, the total time complexity of this step is $O(c_{max}F + n_2) = O(n_1F + n_2)$. The EOU labelling of G_1 is then used to construct G' and to delete certain edges from G' at Steps 4 and 5 of the algorithm, both of which take $O(m)$ time overall.

Finally, we use Gabow's algorithm again to obtain the maximum matching M in G' in $O(\sqrt{C}(F + S))$ time, where S is the number of second-choice edges in G'. The following result gives the overall run-time of Algorithm Popular-CHAT.

Lemma 12. *Given an instance of CHAT, we can find a popular matching, or determine that none exists, in* $O((\sqrt{C} + n_1)m)$ *time.*

It now remains to consider the problem of finding a maximum popular matching in I. The aim is to find a matching that satisfies the conditions of Theorem 3 and that minimises the number of agents who are matched to their last resort houses. We begin by firstly using Algorithm Popular-CHAT to compute a popular matching M in I, assuming such a matching exists. Then $M \cap E_1$ is a maximum matching in G_1. We remove all edges in G' (and thus from M) that are incident to a last resort house. Clearly, M still satisfies the property that $M \cap E_1$ is a maximum matching in G_1, but M need not be maximum in G' if agents become unmatched as a result of the edge removals. Thus, we obtain a new maximum matching M' from M by using Gabow's algorithm on G' again. If M' is not agent-complete in G', we simply assign any agent who remains unmatched in M' to their last resort house to obtain an agent-complete matching. Using an argument similar to that in the proof of Lemma 11, it follows that $M' \cap E_1$ is a maximum matching of G_1. Thus, M' is a maximum popular matching in I. Clearly the overall complexity of this approach is as for Algorithm Popular-CHAT, giving the following result.

Theorem 4. *Given an instance of CHAT, we can find a maximum popular matching, or report that no such matching exists, in* $O((\sqrt{C} + n_1)m)$ *time.*

We may compare the complexity of our direct approach for CHAT to that obtained using cloning on I together with the algorithm of [3] on the cloned instance of I. As in Section 2, the latter approach takes $\Omega(\sqrt{C}m' + C)$ time,

where $m' = \sum_{a_i \in A} \sum_{h_j \in A_i} c_j$. The complexity of Algorithm Popular-CHAT may be rewritten as $O(\sqrt{C}m + m_F + C)$, where $m_F = \sum_{a_i \in A} \sum_{h_j \in f(a_i)} c_j$. Clearly $m_F \leq m'$. Since $m' \geq mc_{min}$, the first term in the complexity function of the cloning method is slower than the first term in that of Algorithm Popular-CHAT by a factor of $\Omega(c_{min})$, which is $\Omega(n_1)$ if $c_j = \Omega(n_1)$ for each $h_j \in H$.

4 Concluding Remarks

We conclude with the following open problem. Suppose that we are presented with an instance J of CHA or CHAT in which the houses have preferences over the agents. Real-life applications of such a problem exist in many centralised matching markets such as the National Resident Marketing Program (NRMP) [12] and counterpart schemes in Canada and Scotland. Then, what is the complexity of finding a maximum popular matching in J if one exists?

Acknowledgement

We would like to thank Rob Irving for helpful discussions concerning this paper.

References

1. A. Abdulkadiroğlu and T. Sönmez. Random serial dictatorship and the core from random endowments in house allocation problems. *Econometrica*, 66:689–701, 1998.
2. D.J. Abraham, K. Cechlárová, D.F. Manlove, and K. Melhorn. Pareto optimality in house allocation problems. In *Proceedings of ISAAC '04*, vol. 3341 of *Lecture Notes in Computer Science*, pp. 3–15. Springer, 2004.
3. D.J. Abraham, R.W. Irving, T. Kavitha, and K. Melhorn. Popular matchings. In *Proceedings of SODA '05*, pp. 424–432. ACM-SIAM, 2005.
4. D.J. Abraham and T. Kavitha. Dynamic matching markets and voting paths. *To appear in Proceedings of SWAT '06*. Springer, 2006.
5. H.N. Gabow. An efficient reduction technique for degree-constrained subgraph and bidirected network flow problems. In *Proceedings of STOC '83*, pp. 448–456, 1983.
6. P. Gärdenfors. Match making: assignments based on bilateral preferences. *Behavioural Sciences*, 20:166–173, 1975.
7. R.W. Irving, T. Kavitha, K. Melhorn, D. Michail, and K. Paluch. Rank-maximal matchings. In *Proceedings of SODA '04*, pp. 68–75. ACM-SIAM, 2004.
8. M. Mahdian. Random popular matchings. In *Proceedings of EC '06*, pp. 238–242. ACM-SIAM, 2006.
9. D.F. Manlove and C.T.S Sng. Popular matchings in the Capacitated House Allocation Problem. Technical Report TR-2006-222, University of Glasgow, Department of Computing Science, June 2006.
10. J. Mestre. Weighted popular matchings. To appear in *Proceedings of ICALP '06*. Springer, 2006.
11. W.R. Pulleyblank. *Matchings and Extensions*. In R.L. Graham, M. Grötschel, and L. Lovász, editors, *The Handbook of Combinatorics, volume 1, chapter 3, pages 179-232*. North Holland, 1995.
12. A.E. Roth. The evolution of the labor market for medical interns and residents: a case study in game theory. *Journal of Political Economy*, 92:991–1016, 1984.

Inner-Product Based Wavelet Synopses for Range-Sum Queries

Yossi Matias and Daniel Urieli

School of Computer Science, Tel Aviv University,
Tel Aviv 69978, Israel
{matias, daniel1}@tau.ac.il

Abstract. In recent years wavelet based synopses were shown to be effective for approximate queries in database systems. The simplest wavelet synopses are constructed by computing the Haar transform over a vector consisting of either the raw-data or the prefix-sums of the data, and using a greedy-heuristic to select the wavelet coefficients that are kept in the synopsis. The greedy-heuristic is known to be optimal for point queries w.r.t. the mean-squared-error, but no similar efficient optimality result was known for range-sum queries, for which the effectiveness of such synopses was only shown experimentally.

We construct an operator that defines a norm that is equivalent to the mean-squared error over all possible *range-sum queries*, where the norm is measured on the *prefix-sums vector*. We show that the Haar basis (and in fact *any wavelet basis*) is orthogonal w.r.t. the inner product defined by this novel operator. This allows us to use Parseval-based thresholding, and thus obtain the first linear time construction of a provably optimal wavelet synopsis for range-sum queries. We show that the new thresholding is very similar to the greedy-heuristic that is based on point queries.

For the case of range-sum queries over the raw data, we define a similar operator, and show that Haar basis is not orthogonal w.r.t. the inner product defined by this operator.

1 Introduction

In recent years there has been increasing attention to the development and study of data synopses, as effective means for addressing performance issues in massive data sets. Data synopses are concise representations of data sets, that are meant to effectively support approximate queries to the represented data sets [5]. A primary constraint of a data synopsis is its size. The effectiveness of a data synopsis is measured by the accuracy of the answers it provides, as well as by its response time and its construction time. Several different synopses were introduced and studied, including random samples, sketches, and different types of histograms. Recently, wavelet-based synopses were introduced and shown to be a powerful tool for building effective data synopses for various applications, including selectivity estimation for query optimization in DBMS, approximate query processing in OLAP applications and more (see [13, 21, 19, 20, 1, 2, 4, 3], and references therein).

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 504–515, 2006.

The general idea of wavelet-based approximations is to transform a given data vector of size N into a representation with respect to a wavelet basis (this is called a *wavelet transform*), and approximate it using only $M \ll N$ wavelet basis vectors, by retaining only M coefficients from the linear combination that spans the data vector (*coefficients thresholding*). The linear combination that uses only M coefficients (and assumes that all other coefficients are zero) defines a new vector that approximates the original vector, using less space. This is called *M-term approximation*, which defines a *wavelet synopsis* of size M.

Wavelet Synopses. Wavelets were traditionally used to compress some data set where the purpose was to reconstruct, in a later time, an approximation of the *whole* data using the set of retained coefficients. The situation is a little different when using wavelets for building synopses in database systems [13, 21]: in this case different *portions* of the data are reconstructed each time, in response to user queries, and same portions of the data can be built several times in response to different queries. Thus, when building wavelet synopses in database systems, the approximation error is measured over *queries*, in contrast to the standard wavelet-based approximation techniques, where the error is measured over the data. Another aspect of the use of wavelet-based synopses is that due to the large data-sizes in modern DBMS (giga-, tera- and peta-bytes), the efficiency of building wavelet synopses is of primary importance. Disk I/Os should be minimized and non-linear-time algorithms may be unacceptable. Wavelet synopses suggested in the database literature typically used the Haar wavelet basis due to its simplicity.

Optimal Wavelet Synopses. The main advantage of transforming the data into a representation with respect to a wavelet basis is that for data vectors containing similar values, many wavelet coefficients tend to have very small values. Thus, eliminating such small coefficients introduces only small errors when reconstructing the original data, resulting in a very effective form of lossy data compression.

After the wavelet transform is done, the selection of coefficients that are retained in the wavelet synopsis may have significant impact on the approximation error. The goal is therefore to select a subset of M coefficients that minimizes the approximation error. A subset that minimizes the approximation error for a given error metric w.r.t. the given basis is called an *optimal wavelet synopsis*.

While there has been a considerable work on wavelet synopses and their applications [13, 21, 19, 20, 1, 2, 4, 3, 12], so far most known optimal wavelet synopses are with respect to *point queries* [13, 21, 2, 4, 12, 15]. Additionally, some of them are built in non-linear time.

Wavelet Synopses for Range-Sum Queries. A primary use of wavelet-based synopses in DBMS is answering range-sum queries. For such synopses, the approximation error is measured over the set of all possible range queries.

In the database literature (e.g., [13, 21, 19, 20]), two types of wavelet synopses for range-sum queries were presented. One over raw data and the other one over

the vector of prefix-sums of the data. In both cases, a range-sum query can be expressed using point queries. In the prefix-sums case, the answer to a range query is a difference between two point queries; in the raw-data case the answer is a sum of all point queries in the range, or using a formula that depends on about $2 \log N$ queries for pre-specified hierarchical ranges.

Thus, the basic thresholding algorithms suggested in [13, 21] were based on a greedy heuristic, that optimizes the synopsis w.r.t. point queries, based on Parseval's theorem. As for range-queries no efficient optimality result has been known, the greedy-heuristic was selected for a lack of a better choice, and due to the simplicity and efficiency of its implementation.

It seems that optimality over points would give especially good results for the case of prefix-sums, where the answer to a range-query is a difference between only two point queries. Moreover, note that if we are interested only in range queries of the form $d_{0:i}$, that is, $\sum_{i=0}^{i} d_i$, then the greedy-heuristic is optimal, as a point query in this case is exactly a range query of the form $d_{0:i}$. However, it turns out that when using the greedy heuristic over prefix-sums for general queries $d_{i:j}$, the mean-squared-error could be larger than the optimal error by a factor of $\Theta\left(\sqrt{N}\right)$, as shown in this paper (Thm. 4). Nevertheless, we show here that a slight variation of the greedy heuristic is indeed an optimal thresholding for the case of prefix-sums.

One optimality result for range-sum queries is mentioned in [6]; the authors introduce an algorithm that computes optimal M-term wavelet synopses for range-sum queries over prefix-sums in $O\left(N\left(M \log N\right)^{O(1)}\right)$ time.

1.1 Contributions

As pointed out above, the greedy heuristic is based on applying the Parseval-based thresholding, which is optimal for point queries, for the case of range-sum queries. The reason we can rely on Parseval's formula and get an (efficient) optimal thresholding in the case of point queries, is because in this case the Haar basis is orthogonal.

Our main goal is to be able to use a Parseval-based thresholding that is *optimal* for *range-sum* queries. The main idea is to express the mean-squared-error measured over $(\Theta\left(N^2\right))$ *range-sum* queries using an inner product defined on the raw-data / prefix-sums vector (vectors of size N), and check whether the Haar basis is still orthogonal in these cases. So far inner products were used in a more conventional way, to define an Euclidean error between two vectors, or a generalized Euclidean error (weigthed norm, see [12]). The main technical contributions with respect to this approach are:

- We construct an operator that defines a norm that is equivalent to the mean-squared error over all possible *range-sum queries*, where the norm is measured on the *prefix-sums vector*.
- We show that the Haar basis (and in fact *any wavelet basis*) is orthogonal w.r.t. the inner product defined by this novel operator. This allows us to

use Parseval-based thresholding, and thus obtain the first linear time, I/O optimal, construction of a provably optimal wavelet synopsis.
- We show that the new thresholding is very similar to the greedy-heuristic that is based on point queries.
- The synopsis is also an optimal *enhanced wavelet synopsis*. Enhanced wavelet synopses are synopses that allow changing the values of their coefficients to *arbitrary values*.
- For the case of range-sum queries over the raw data, we define a similar operator, and show that Haar basis is not orthogonal w.r.t. the inner product defined by this operator, both analytically and experimentally. For non-orthogonal bases no efficient optimal thresholding algorithm is known. Additionally, our empirical proof demonstrates an anomaly when using a non-orthogonal basis, where a larger synopsis may result with an increased error.

The problem that is at the heart of the subject is the representation of a symmetric operator (the X matrix in this paper) in a wavelet basis. The deep mathematical question is: what kind of matrices are diagonal in a wavelet basis. This is connected to the theory of Calderon-Zygmund operators which are sparse in wavelet bases. For additional references, see [7, 14].

1.2 Paper Outline

In Sec. 2 we describe some basics regarding Parseval Formula and its use. In Sec. 3 we describe the development of the optimal synopsis for prefix-sums. We build the inner-product for the case of prefix-sums, and then construct the optimal synopsis resulted from it. We then discuss the similarity and difference between our optimal wavelet synopsis and the greedy-heuristic given in [13, 21]. In Sec. 4 we show the non-orthogonality of Haar basis for range queries in the raw-data case. Conclusions are given in Sec. 5.

2 Optimal Thresholding in Orthonormal Bases

The efficient construction of optimal wavelet-synopses is commonly based on Parseval's formula.

2.1 Parseval's Formula

Let V be a vector space, where $v \in V$ is a vector and $\{u_0, ..., u_{N-1}\}$ is an orthonormal basis of V. We can express v as $v = \sum_{i=0}^{N-1} \alpha_i u_i$. Then

$$\|v\|^2 = \sum_{i=0}^{N-1} \alpha_i^2 \tag{1}$$

An M-term approximation is achieved by representing v using a subset of coefficients $S \subset \{\alpha_0, ..., \alpha_{N-1}\}$ where $|S| = M$. The error vector is then $e =$

$\sum_{i \notin S} \alpha_i u_i$. By Parseval's formula, $\|e\|^2 = \sum_{i \notin S} \alpha_i^2$. This proves the following theorem.

Theorem 1 (Parseval-based optimal thresholding). *Let V be a vector space, where $v \in V$ is a vector and $\{u_0, ..., u_{N-1}\}$ is an orthonormal basis of V. We can represent v by $\{\alpha_0, ..., \alpha_{N-1}\}$ where $v = \sum_{i=0}^{N-1} \alpha_i u_i$. Suppose we want to approximate v using a subset $S \subset \{\alpha_0, ..., \alpha_{N-1}\}$ where $|S| = M \ll N$. Picking the M largest (in absolute value) coefficients to S minimizes the L_2 norm of the error vector, over all possible subsets of M coefficients.*

Given an inner-product, based on this theorem one can easily find an optimal synopsis by choosing the largest M coefficients. In fact, we can use the Parseval-based optimal thresholding even when using an *orthogonal* basis, as we can normalize coefficients by multiplying them by the corresponding basis vectors norms. Parseval-based thresholding can then be applied on the normalized coefficients (see [13] for such usage). Thus, the main technical question with which we are concerned in the rest of the paper is the *orthogonality* of the Haar basis in specific cases of interest.

Note that in order to show a negative result, that is, that Thm. 1 *cannot* be applied during the thresholding w.r.t. a given basis, it is sufficient to find an inner product that defines the desired L_2 norm, and show that the given basis is not orthogonal w.r.t. this inner product. This relies on the fact that if a norm is defined by some inner-product, then this inner-product is unique; that is, no other inner-product defines the same norm.

Thus, if a basis is shown to be non-orthogonal w.r.t. an inner product $\langle \cdot, \cdot \rangle$ whose norm is $\| \cdot \| = \sqrt{\langle \cdot, \cdot \rangle}$, then it can be said to be *non-orthogonal w.r.t. the norm* $\| \cdot \|$.

2.2 Optimality over Enhanced Wavelet Synopses

As our synopses are built w.r.t. an orthonormal basis, they are also *enhanced* wavelet synopses (see [12]).

3 The Synopsis Construction

In this section we describe the development of our optimal synopsis. First we define the MSE error metrics by which we measure the approximation error over range-sum queries (Sec. 3.1), denoted here as MSE_{range}. Our goal is to efficiently build a synopsis that minimizes MSE_{range}. Our main idea is to define our problem in terms of an inner product space by constructing a range-sum-based inner product (Sec. 3.2), and to show that this inner product defines an L_2 norm that is equivalent, up to a constant positive factor, to MSE_{range}, when approximating a prefix-sums vector (Sec. 3.3). We then show that the Haar basis (and in fact *any* wavelet basis) is orthogonal with respect to this inner product and normalize it (Sec. 3.4). Next, we discuss the complexity of the algorithm (Sec. 3.5). Finally we show the similarity to the greedy heuristic (Sec. 3.6).

3.1 The Error Metrics for Range-Sum Queries

We define the error metrics by which the approximation error is measured. This is the mean-squared-error (MSE), measured over all possible *range-sum* queries.

Let $D = (d_0, ..., d_{N-1})$ be a sequence with $N = 2^j$ values. Let d_i be the i'th data value, and let $q_{l:r}$ be the range query $\sum_{i=l}^{r} d_i$. Let $d_{l:r}$ be the answer to the range query $q_{l:r}$ and let $\hat{d}_{l:r}$ be an approximated answer to the query $q_{l:r}$. The *absolute error* of the approximated answer is defined as $|e_{l:r}| = |d_{l:r} - \hat{d}_{l:r}|$. We can now define the mean-squared-error of any approximation that approximates all range-queries in some way. Such approximation defines a vector $\hat{R} = \left(\hat{d}_{1:1}, ..., \hat{d}_{1:N}, \hat{d}_{2:2}, ..., \hat{d}_{2:N}, ..., \hat{d}_{N:N}\right)$. A vector of approximated answers defines a vector of errors $E = (e_{1:1}, ..., e_{1:N}, e_{2:2}, ..., e_{2:N}, ..., e_{N:N})$. The MSE is defined as:

$$MSE_{range}\left(\hat{R}\right) = \frac{1}{(N+1)\,N/2} \sum_{i=1,...,N j=i,...,N} e_{i:j}^2$$

which is the sum of squared errors divided by the number of possible range-sum queries. Note that typically the sum of squared errors was measured only over point queries.

3.2 The Prefix-Sum Based (PSB) Inner Product

We want to approximate a data vector $v \in R^N$ where $N = 2^j$. Our inner product, called *PSB inner product*, would be defined by the following positive symmetric bilinear form:

$$\mathbf{X} = \begin{pmatrix} N & -1 & \dots & -1 \\ -1 & N & \ddots & \vdots \\ \vdots & \ddots & \ddots & -1 \\ -1 & \dots & -1 & N \end{pmatrix} \tag{2}$$

That is, $\langle v, u \rangle_X := v^T X u$ where $v, u \in R^N$.

3.3 Defining a Norm Based on the PSB Inner Product

Based on the PSB inner product we define the norm:

$$\|v\|_{\mathrm{X}} = \sqrt{\langle v, v \rangle_X} \tag{3}$$

Lemma 1. *Let $P = (p_1, ..., p_N)$ be a vector of prefix sums of the data. Let $\hat{P} = (\hat{p_1}, ..., \hat{p_N})$ be a vector that approximates it, by which we answer range-sum queries. Let $E_p = (p_1 - \hat{p_1}, ..., p_N - \hat{p_N}) = (e_{p_1}, ..., e_{p_N})$ be the error vector. Let $\hat{R}$ be the vector of approximations of all range-sum queries, answered using $\hat{P}$.*

Then,

$$\|E_p\|_{\mathrm{X}}^2 = \frac{N\,(N+1)}{2} MSE_{range}\left(\hat{R}\right).$$

Proof. Let $v \in R^N$ where $v = (v_1, \ldots, v_N)$. Then

$$\|v\|_{\mathrm{X}}^2 = \langle v, v \rangle_X = v^T X v = \sum_{i,j} v_i \cdot X_{ij} \cdot v_j = ... =$$

$$\sum_{i=1}^{N} v_i^2 + \sum_{i=2}^{N} (v_i - v_1)^2 + \sum_{i=3}^{N} (v_i - v_2)^2 + \cdots + (v_N - v_{N-1})^2$$

Now, let $D = (d_1, ..., d_N)$ be a vector of data values, which $P = (p_1, \ldots, p_N)$ is the vector of its prefix-sums ($p_i = \sum_{i=1}^{i} d_i$). Each range-sum query $d_{l:r}$ is computed by $d_{l:r} = p_r - p_{l-1}$ (p_{-1} is defined as 0 and is not part of the vector). Therefore the absolute error of a specific range sum query approximation is:

$$|e_{l:r}| = |d_{l:r} - \hat{d}_{l:r}| = |\,(p_r - p_{l-1}) - (\hat{p}_r - \hat{p}_{l-1})\,| =$$

$$|\,(p_r - \hat{p}_r) - (p_{l-1} - \hat{p}_{l-1})\,| = |e_{p_r} - e_{p_{(l-1)}}|$$

As a result

$$\|E_p\|_{\mathrm{X}}^2 = \langle E_p, E_p \rangle_X =$$

$$\sum_{i=1}^{N} e_{p_i}^2 + \sum_{i=2}^{N} (e_{p_i} - e_{p_1})^2 + \sum_{i=3}^{N} (e_{p_i} - e_{p_2})^2 + \cdots + \left(e_{p_N} - e_{p_{(N-1)}}\right)^2 =$$

$$\sum_{i=1}^{N} e_{1:i}^2 + \sum_{i=2}^{N} e_{2:i}^2 + \sum_{i=3}^{N} e_{3:i}^2 + \cdots + e_{N:N}^2 = \frac{N\,(N+1)}{2} MSE_{range}\left(\hat{R}\right)$$

This concludes our proof.

Minimizing $\|E_p\|_{\mathrm{X}}$ is equivalent to minimizing the $MSE_{range}\left(\hat{R}\right)$ norm, since $\frac{N(N+1)}{2}$ is always positive and constant. By proving the Haar basis is orthogonal with respect to the PSB inner product, we would be able to use Thm. 1: choosing the M largest normalized coefficients to our synopses (where M is the space limitation) would minimize $\|E_p\|_{\mathrm{X}}$, and equivalently $MSE_{range}\left(\hat{R}\right)$.

3.4 Orthonormality of the Haar Basis with Respect to the PSB Inner Product

In this section we show the orthonormality of the Haar basis w.r.t. the PSB inner product. We show it in two stages. First we show that the Haar basis is orthogonal w.r.t. the PSB inner product. Then we show how to normalize the basis.

Theorem 2. *The Haar basis is orthogonal with respect to the PSB inner product.*

Proof. A basic algebraic fact is that if a basis H is orthogonal and diagonalises a symmetric operator X, then it is still orthogonal for the dot product defined by X. All that we need to show is that Haar basis diagonalises X. We use the fact that the wavelet has one vanishing moment, so the mean of a wavelet is 0. Note that $X = (N+1)\,Id - \bar{\mathbf{1}} \cdot \bar{\mathbf{1}}^t$, where Id is the identity matrix, $\bar{\mathbf{1}}$ is the vector filled with 1, and $\bar{\mathbf{1}}^t$ is its transpose. $\bar{\mathbf{1}} \cdot \bar{\mathbf{1}}^t$ is the sum operator, which maps a wavelet to 0. Thus each wavelet is an eigen-vector of the operator X, and so is the constant function. Thus, Haar basis diagonlises X.

In fact, the same proof shows that *any* standard (not *weighted*, see [12]) wavelet basis diagonalises X.

Corollary 1. *Theorem 2 is valid for any standard wavelet basis.*

As we have seen, the Haar basis is orthogonal with respect to our PSB inner product. We normalize each basis vector in order to have an orthonormal basis. For the first basis vector $u_1 = (1, \ldots, 1)$ it is easy to verify that its norm is $\|u_1\|_{\mathrm{X}} = \sqrt{\langle u_1, u_1 \rangle_X} = \sqrt{N}$. For any other basis vector v at level i its norm is $\|u\|_{\mathrm{X}} = \sqrt{\frac{N}{2^i}\,(N+1)}$. In order to normalize the basis, we divide each basis vector by its norm. Transforming the basis w.r.t. the orthonotmal basis still takes linear time.

3.5 Building the Optimal Synopsis

First, the algorithm transforms the vector of prefix-sums with respect to the normalized Haar basis. Equivalently, the algorithm could transform the vector w.r.t. the *orthogonal* Haar basis and then normalize the wavelet coefficients (see [13]). The vector of prefix-sums, if not built yet, can be computed *during* the wavelet transform. Computing the Haar wavelet transform takes linear time using $O(N/B)$ I/Os. Next, the algorithm chooses the largest M coefficients which can be done in linear time using the *M-approximate quantile* algorithm [8]. Note that although there are $O(N^2)$ range-sum queries, our algorithm didn't use at any stage the vector of all possible queries. It was described for the purpose of analysis.

Based on Corollary 1, we can use the same construction for any wavelet basis. The following theorem follows from our construction, together with Thm. 1 and Thm. 2 (extended by Corollary 1):

Theorem 3. *An optimal wavelet synopses for a vector of size N, which minimizes the* MSE *measured over all possible range-sum queries, can be constructed in linear-time, using $O(N/B)$ I/Os, for a disk block of size B.*

3.6 Comparison Between the Optimal Thresholding and the Greedy Heuristic

The greedy heuristic for wavelet-synopses thresholding is commonly described as a two-stage process. First, the transform is computed w.r.t. the *orthogonal* Haar basis, where all non-zero coordinates are ± 1. Then, the coefficients are

normalized to be the coefficients of the linear combination w.r.t. the *normalized* basis (Sec. 2). We show that the greedy-heuristic thresholding is nearly identical to the optimal thresholding described in Thm. 3. Specifically, the resulting synopses may defer in at most a single coefficient.

The greedy heuristic (see [13]) transforms the data with respect to the orthogonal Haar basis, and normalizes each coefficient as follows: a coefficient of a vector at level i by multiplied by $\frac{1}{\sqrt{2^i}}$. Suppose that we scale all the coefficients of the greedy heuristic by multiplying them with the same factor $\sqrt{N\left(N+1\right)}$. Clearly, the greedy thresholding will still select the same coefficients to be part of the computed synopsis. Recall that the optimal synopsis computes the same Haar transform, and normalizes each coefficient as follows: a coefficient of the first basis vector is multiplied by $\sqrt{N}$, and any other coefficient is multiplied by $\sqrt{\frac{N}{2^i}\left(N+1\right)}$. As can be easily verified, except for the first coefficient, all coefficients in the optimal synopsis construction are identical to the scaled coefficients in the greedy heuristic. Therefore, the only possible difference between the optimal synopsis and the standard synopsis (obtained by the greedy heuristic) is a situation where the coefficient of v_0 is included in the standard synopsis but not in the optimal synopsis.

While the optimal synopsis and the standard synopsis are nearly identical, the difference in their error can be significant in extreme situations:

Theorem 4. *When using the greedy-heuristic that is based on point queries, instead of the above optimal thresholding, the mean-squared-error might be* $\Theta\left(\sqrt{N}\right)$ *times larger than the optimal error.*

Proof. Consider a wavelet transform that results in the following coefficients, normalized according to the greedy heuristic: $[\alpha_0, \ldots, \alpha_{N-1}] = [m, m, m, m, m-1, \epsilon, \epsilon, \ldots, \epsilon]$ and suppose that we have a synopsis consisting of 4 coefficients. The greedy heuristic would keep the first 4 coefficients, resulting with a mean-squared-error of $\sqrt{\left(m-1\right)^2 + \left(N-5\right)\cdot\epsilon^2}$, which is $\Theta\left(m\right)$ for $\epsilon = O(1/\sqrt{N})$. The optimal algorithm would normalize the first coefficient to $\frac{m}{\sqrt{N+1}}$, and consequently not keep it in the synopsis, but instead keep in the synopsis the next 4 coefficients: $m, m, m, m-1$. The error in this case is $\sqrt{\left(m/\sqrt{N+1}\right)^2 + \left(N-5\right)\cdot\epsilon^2}$, which is $\Theta\left(m/\sqrt{N}\right)$ for $\epsilon = O(1/\sqrt{N})$; that is, smaller by a factor of $\Theta(\sqrt{N})$ than that of the standard greedy heuristic.

Comment: Vectors of prefix-sums tend to be monotone increasing in database-systems, as in many cases the raw-data has non-negative values (for example in selectivity estimation). In this case we should slightly change the proof so that the wavelet coefficients would be of a non-decreasing vector. We would fix the "small" coefficients to be $\epsilon, ..., \epsilon, \frac{\epsilon}{2}, ..., \frac{\epsilon}{2}, \frac{\epsilon}{4}, ...$, according to their levels in the tree (in level i the "ϵ" coefficient would be divided by 2^i ($i < \log N$). One can easily verify that the resulting vector would be monotone non-decreasing, and yet the wavelet coefficients are small enough, such that the proof stands.

4 The Non-orthogonality of the Haar Basis over Raw Data

In this section we define the inner product that corresponds to the MSE when answering range-sum queries over the raw data. We then show that the Haar basis is not orthogonal with respect to this inner product. Consequently, the Haar basis is non-orthogonal w.r.t. the desired norm and Parseval's formula cannot be applied for optimal thresholding. We prove the non-orthogonality in two different ways. First we give an analytical proof, and then give a different, empirical proof. The latter also demonstrates an anomaly when using a non-orthogonal basis, where a larger synopsis may result with an increased error.

Lemma 2 (raw-data inner product). *Let $D = (d_1, ..., d_N)$ be a data vector. Let $\hat{D} = \left(\hat{d_1}, ..., \hat{d_N}\right)$ be a vector that approximates the raw data, D, built using the Haar-based wavelet synopsis. An answer to a range query $d_{l:r}$ is approximated as $\hat{d}_{l:r} = \sum_{i=l}^{r} \hat{d}_i$.*
As above, define $\hat{R} := \left(\hat{d}_{1:1}, ..., \hat{d}_{1:N}, \hat{d}_{2:2}, ..., \hat{d}_{2:N}, ..., \hat{d}_{N:N}\right)$. Denote:

$$X_{l:r} := \begin{pmatrix} & & \\ & 1 \ldots 1 & \\ & \vdots \ddots \vdots & \\ & 1 \ldots 1 & \\ & & \end{pmatrix} \begin{matrix} \\ l \\ \\ r \\ \\ \end{matrix}$$

and $X_{raw} := \sum_{l=1,...,N\ r=l,...,N} X_{l:r}$. Let $E = (e_1, ..., e_N) = \left(d_1 - \hat{d}_1, ..., d_N - \hat{d_N}\right)$ be the vector of errors. Then:

1. $E^T X_{raw} E = \frac{N(N+1)}{2} MSE_{range}\left(\hat{R}\right)$.
2. $\langle v, u \rangle_{X_{raw}} := v^T X_{raw} u$ *is an inner product.*

Due to space limitations, the proof is omitted and can be found in the full paper [11]. The proofs for non-orthogonality are briefly described here, and can also be found in [11]. The basic idea of the analytical proof is to give a counter example by choosing pairs of basis functions, and show their inner product $\langle , \rangle_{X_{raw}}$ is not 0. The basic idea of the empirical proof is to show an experiment where adding a coefficient to a Haar wavelet synopsis increases $\|\cdot\|_{X_{raw}}$. This is impossible for orthogonal bases, and thus the Haar basis is not orthogonal w.r.t. $\langle , \rangle_{X_{raw}}$.

5 Conclusions

In this paper we construct an operator that defines a norm that is equivalent to the mean-squared error over all possible *range-sum queries*, where the norm is measured on the *prefix-sums vector*. We show that the Haar basis (and in fact *any wavelet basis*) is orthogonal w.r.t. the inner product defined by this novel

operator. This allows us to use Parseval-based thresholding, and thus obtain the first linear time construction of a provably optimal wavelet synopsis for range-sum queries. We show that the new thresholding is very similar to the greedy-heuristic that is based on point queries. For the case of range-sum queries over the raw data, we define a similar operator, and show that Haar basis is not orthogonal w.r.t. the inner product defined by this operator. The problem that is at the heart of the subject is the representation of a symmetric operator (the X matrix in this paper) in a wavelet basis. The deep mathematical question is: what kind of matrices are diagonal in a wavelet basis.

Recently, an interesting relationship was discovered [18] between our prefix-sums based inner product and Lemma 3.1 from [16]. In their lemma, the authors prove a proportion between the error for range queries in raw-data based histograms, and the error for point queries in the corresponding prefix-sums based histograms. Using a generalization of their lemma to an arbitrary error vector, one can give an alternative proof, using probabilistic techniques, that our inner product expresses the mean-squared error over all range-sum queries.

This paper leads to three interesting open problems. The first one is exploring whether the prefix-sums X operator, and the operator-based inner-product method used here, are applicable in other fields of approximation theory. The second one is finding an optimal wavelet synopsis for range-sum queries over the raw-data representation. We constructed the operator that defines the mean-squared error over the raw data, but showed that Haar basis is not orthogonal w.r.t. the inner-product defined by this operator. The third one is finding an optimal workload-based wavelet synopsis for a workload of *range queries*. Recall that effective, yet non-optimal workload-based synopses for range queries were presented in [9, 17], and that efficient workload-based wavelet synopses for point queries were given in [12].

Acknowledgement. We thank an anonymous referee for helpful comments regarding relationships to the literature on approximation theory and applicability to a wider set of wavelet bases. We thank Leon Portman for helpful discussions, and for his help with the τ-synopses system [10]. We thank Yariv Matia for his help in conducting our experiments.

References

[1] K. Chakrabarti, M. Garofalakis, R. Rastogi, and K. Shim. Approximate query processing using wavelets. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, 2000*, pages 111–122.

[2] A. Deligiannakis and N. Roussopoulos. Extended wavelets for multiple measures. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 229–240.

[3] M. Garofalakis and P. B. Gibbons. Wavelet synopses with error guarantees. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, 2002.

[4] M. Garofalakis and A. Kumar. Deterministic wavelet thresholding for maximum-error metrics. In *Proceedings of the 2004 ACM PODS international conference on on Management of data*, pages 166–176.

[5] P. B. Gibbons and Y. Matias. Synopsis data structures for massive data sets. In *External Memory Algorithms, DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 50 (1999).

[6] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. J. Strauss. Optimal and approximate computation of summary statistics for range aggregates. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 227–236. ACM Press, 2001.

[7] W. Hardle, G. Kerkyacharian, D. Picard, and A. Tsybakov. *Wavelets, Approximation and Statistical Applications*, volume 129. New-York: Springer-Verlag, 1998.

[8] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 426–435, 1998.

[9] Y. Matias and L. Portman. Workload-based wavelet synopses. Technical report, Department of Computer Science, Tel Aviv University, 2003.

[10] Y. Matias and L. Portman. τ-synopses: a system for run-time management of remote synopses. In *International conference on Extending Database Technology (EDBT), Software Demo, 865-867 & ICDE'04, Software Demo*, March 2004.

[11] Y. Matias and D. Urieli. On the optimality of the greedy heuristic in wavelet synopses for range queries. Technical report, Department of Computer Science, Tel-Aviv University, 2004; revised, 2005.

[12] Y. Matias and D. Urieli. Optimal workload-based weighted wavelet synopses. In *Proceedings of the 2005 ICDT conference* (Full version in *TCS, special issue of ICDT*), January 2005.

[13] Y. Matias, J. S. Vitter, and M. Wang. Wavelet-based histograms for selectivity estimation. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 448–459, June 1998.

[14] Yves Meyer. *Wavelets and operators*, volume 37 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, Cambridge, 1992. Translated from the 1990 French original by D. H. Salinger.

[15] S. Muthukrishnan. Nonuniform sparse approximation using haar wavelet basis. Technical report, DIMACS, May 2004.

[16] S. Muthukrishnan and M. Strauss. Rangesum histograms. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 233–242, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[17] L. Portman. Workload-based wavelet synopses. Master's thesis, School of Computer Science, Tel Aviv University, 2003.

[18] M. Strauss. Personal communication, October 2005.

[19] J. S. Vitter and M. Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pages 193–204, June 1999.

[20] J. S. Vitter, M. Wang, and B. Iyer. Data cube approximation and histograms via wavelets. In *Proceedings of Seventh International Conference on Information and Knowledge Management*, pages 96–104, November 1998.

[21] M. Wang. *Approximation and Learning Techniques in Database Systems*. PhD thesis, Duke University, 1999.

Approximation in Preemptive Stochastic Online Scheduling

Nicole Megow[1,⋆] and Tjark Vredeveld[2,⋆⋆]

[1] Technische Universität Berlin, Institut für Mathematik, Strasse des 17. Juni 136, 10623 Berlin, Germany
nmegow@math.tu-berlin.de
[2] Maastricht University, Department of Quantitative Economics, P.O. Box 616, 6200 MD Maastricht, The Netherlands
t.vredeveld@ke.unimaas.nl

Abstract. We present a first constant performance guarantee for preemptive stochastic scheduling to minimize the sum of weighted completion times. For scheduling jobs with release dates on identical parallel machines we derive a policy with a guaranteed performance ratio of 2 which matches the currently best known result for the corresponding deterministic online problem.

Our policy applies to the recently introduced stochastic online scheduling model in which jobs arrive online over time. In contrast to the previously considered nonpreemptive setting, our preemptive policy extensively utilizes information on processing time distributions other than the first (and second) moments. In order to derive our result we introduce a new nontrivial lower bound on the expected value of an unknown optimal policy that we derive from an optimal policy for the basic problem on a single machine without release dates. This problem is known to be solved optimally by a Gittins index priority rule. This priority index also inspires the design of our policy.

1 Introduction

Stochastic scheduling problems have attracted researchers for about four decades, see e.g. [20]. A full range of articles concerns criteria that guarantee the optimality of simple policies for special scheduling problems. Only recently research interest has also focused on approximative policies [18, 26, 15, 21] for nonpreemptive scheduling. We are not aware of any approximation results for preemptive problems. Previous approaches, based on linear programming relaxations, do not seem to carry over to the preemptive setting. In this paper, we give a first approximation result for preemptive stochastic scheduling to minimize the weighted sum of completion times. We prove an approximation guarantee of 2 even in the recently introduced more general model of stochastic online scheduling [15, 4].

⋆ Supported by the DFG Research Center Matheon *Mathematics for key technologies* in Berlin.

⋆⋆ Research partially supported by METEOR, the Maastricht research school of Economics of Technology and Organizations.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 516–527, 2006.

This guarantee matches exactly the currently best known approximation result for the deterministic online version of this problem [14].

Problem definition. Let $J = \{1, 2, \ldots, n\}$ be a set of jobs which must be scheduled on m identical parallel machines. Each of the machines can process at most one job at a time, and any job can be processed by no more than one machine at a time. Each job j has associated a positive weight w_j and an individual release date $r_j \geq 0$ before which it is not available for processing. We allow preemption which means that the processing of a job may be interrupted and resumed later on the same or another machine.

The stochastic component in the model we consider is the uncertainty about processing times. Any job j must be processed for P_j units of time, where P_j is a random variable. By $\mathbb{E}[P_j]$ we denote the expected value of the processing time of job j and by p_j a particular realization of P_j. We assume that all random variables of processing times are stochastically independent and follow discrete probability distributions. With the latter restriction and a standard scaling argument, we may assume w.l.o.g. that P_j attains integral values in the set $\Omega_j \subseteq \{1, 2, \ldots, M_j\}$ and that all release dates are integral. The sample space of all processing times is denoted by $\Omega = \Omega_1 \times \cdots \times \Omega_n$.

The objective is to schedule all jobs so as to minimize the total weighted completion time of the jobs, $\sum_{j \in J} w_j C_j$, in expectation, where C_j denotes the completion time of job j. Adopting the well-known three-field classification scheme by Graham et al. [8], we denote the problem by $\mathrm{P} \,|\, r_j, pmtn \,|\, \mathbb{E}[\sum w_j C_j]$.

The solution of a stochastic scheduling problem is not a simple schedule, but a so-called *scheduling policy*. We follow the notion of scheduling policies as proposed by Möhring, Radermacher, and Weiss [17]. Roughly spoken, a scheduling policy makes scheduling decisions at certain *decision time points* t, and these decisions are based on information on the observed past up to time t, as well as the a priori knowledge of the input data of the problem. The policy, however, must not anticipate information about the future, such as the actual realizations p_j of the processing times of the jobs that have not yet been completed by time t.

Additionally, we restrict ourselves to so-called *online policies*, which learn about the existence and characteristics of a job only at its individual release date. This means for an online policy that it must not anticipate the arrival of a job at any time earlier than its release date. At this point in time, the job with its processing time distribution and weight is revealed. Thus, our policies are required to be online and non-anticipatory. However, an optimal policy can be offline as long as it is non-anticipatory. We refer to Megow, Uetz, and Vredeveld [15] for a more detailed discussion on stochastic online policies. In this paper, we concentrate on (online) approximation policies. As suggested in [15] we use a generalized definition of approximation guarantees from the stochastic scheduling setting [17].

Definition 1. *A (online) stochastic policy Π is a ρ-approximation, for some $\rho \geq 1$, if for all problem instances $\mathcal{I}$,*

$$\mathbb{E}[\Pi(\mathcal{I})] \;\leq\; \rho\, \mathbb{E}[\,\mathrm{OPT}(\mathcal{I})\,]\,,$$

where $\mathbb{E}[\Pi(\mathcal{I})]$ *and* $\mathbb{E}[\mathrm{OPT}(\mathcal{I})]$ *denote the expected values that the policy* Π *and an optimal non-anticipatory offline policy, respectively, achieve on a given instance* $\mathcal{I}$*. The value* ρ *is called performance guarantee of policy* Π*.*

Previous work. Stochastic scheduling has been considered for more than 30 years. Some of the first results on preemptive scheduling that can be found in literature are by Chazan, Konheim, and Weiss [2] and Konheim [11]. They formulated sufficient and necessary conditions for a policy to solve optimally the single machine problem where all jobs become available at the same time. Later Sevcik [24] developed an intuitive method for creating optimal schedules (in expectation). He introduces a priority policy that relies on an index which can be computed for each job based on the properties of a job, but not on other jobs.

Gittins [6] showed that this priority index is a special case of his Gittins index [6, 7]. Later in 1995, Weiss [30] formulated Sevcik's priority index again in terms of the Gittins index and names it a *Gittins index priority policy.* He also provided a different proof of the optimality of this priority policy, based on the work conservation invariance principle. Weiss covers a more general problem than the one considered here and in [2, 11, 24]: The holding costs (weights) of a job are not deterministic constants, but may vary during the processing of a job. At each state these holding costs are random variables.

For more general scheduling problems with release dates and/or multiple machines, no optimal policies are known. Instead, literature reflects a variety of research on restricted problems as those with special probability distributions for processing times or special job weights [1, 29, 19, 5, 9, 30].

For the parallel machine problem without release dates it is worthwhile to mention that Weiss [30] showed that the Gittins index priority policy above is asymptotically optimal and has a turnpike property, which means that there is a bound on the number of times that the policy differs from an optimal policy.

Optimal policies have only been found for a few special cases of stochastic scheduling problems. Already the deterministic counterpart of the scheduling problem we consider, is well-known to be NP-hard, even in the case that there is only a single processor or if all release dates are equal [12, 13]. Therefore, recently attempts have been made on obtaining approximation algorithms which have been successful in the nonpreemptive setting. Möhring, Schulz, and Uetz [18] derived first constant-factor approximations for the nonpreemptive problem with and without release dates. They were improved later by Megow et al. [15] and Schulz [21] for a more general setting. Skutella and Uetz [26] complemented the first approximation results by constant-approximative policies for scheduling with precedence constraints. In general, all given performance guarantees for nonpreemptive policies depend on a parameter defined by expected values of processing times and the coefficients of variation.

In contrast to stochastic scheduling, in a deterministic online model is assumed that no information about any future job arrival is available. However, once a job arrives, its weight and actual processing time become known immediately. The performance of online algorithms is typically assessed by their competitive

ratio [10, 27]. An algorithm is called ρ-competitive if it achieves for any instance a solution with a value at most ρ times the value of an optimal offline solution.

In this deterministic online model, Sitters [25] gave a 1.56-competitive algorithm for preemptive scheduling on a single machine. This is the currently best known result and it improved upon an earlier result by Schulz and Skutella [22]; they generalized the classical *Smith rule* [28] to the problem of scheduling jobs with individual release dates and achieved a competitive ratio of 2. This algorithm has been generalized further to the multiple machine problem without loss of performance by Megow and Schulz [14]. As far as we know, there is no randomized online algorithm known with a provable competitive ratio less than 2 for this problem. In contrast, Schulz and Skutella [23] provide a 4/3-competitive algorithm for the single machine problem.

Recently, the stochastic scheduling model as we consider it in this paper has been investigated; all obtained results which include asymptotic optimality [4] and approximation guarantees for deterministic [15] and randomized policies [15, 21] address nonpreemptive scheduling.

Our contribution. We derive a first constant performance guarantee for preemptive stochastic scheduling. For jobs with general processing time distributions and individual release dates, we give a 2-approximation policy for multiple machines. This performance guarantee matches the currently best known result in deterministic online scheduling although we consider a more general model. In comparison to the previously known results in this model in a nonpreemptive setting, our result stands out by being constant and independent of the probability distribution of processing times.

In general our policy is not optimal. However, on restricted problem instances it coincides with policies whose optimality is known. If processing times are exponentially distributed and release dates are absent, our policy coincides with the *Weighted shortest expected processing time* (WSEPT) rule. This classical policy is known to be optimal if all weights are equal [1] or, more general, if they are *agreeable*, which means that for any two jobs i, j holds that $\mathbb{E}[P_i] < \mathbb{E}[P_j]$ implies $w_i \leq w_j$ [9]. If only one machine is available, we solve the weighted problem $1 \,|\, pmtn \,|\, \mathbb{E}[\sum w_j C_j]$ optimally by utilizing the Gittins index priority policy [11, 24, 30]. Moreover, Pinedo showed in [19] that in presence of release dates the WSEPT rule is optimal if all processing times are exponentially distributed.

Our result is based on a new nontrivial lower bound for the preemptive stochastic scheduling problem. This bound is derived by borrowing ideas for a *fast single machine relaxation* from Chekuri et al. [3]. The crucial ingredient to our result is then the application of a Gittins index priority policy which is optimal to a relaxed version of our fast single machine relaxation.

2 A Gittins Index Priority Policy

As mentioned in the introduction, a Gittins index priority policy solves the single machine problem with trivial release dates to optimality, see [11, 24, 30]. This result is crucial for the approximation result we give in this paper; it inspires

the design of our policy and it serves as a tool for bounding the expected value of an unknown optimal policy for the more general problem that we consider. Therefore, we introduce in this section the Gittins index priority rule and some useful notation.

Given that a job j has been processed for y time units, we define the *expected investment* of processing this job for q time units or up to completion, which ever comes first, as

$$I_j(q,y) = \mathbb{E}\left[\min\{P_j - y, q\} \mid P_j > y\right].$$

The ratio of the weighted probability that this job is completed within the next q time units over the expected investment, is the basis of the Gittins index priority rule. Therefore, we define this as the *rank* of a sub-job of length q of job j, after it has completed y units of processing:

$$R_j(q,y) = \frac{w_j \Pr\left[P_j - y \le q \mid P_j > y\right]}{I_j(q,y)}.$$

For a given (unfinished) job j and attained processing time y, we are interested in the maximal rank it can achieve. We call this the Gittins index, or rank, of job j, after it has been processed for y time units.

$$R_j(y) = \max_{q \in \mathbb{R}^+} R_j(q,y).$$

The length of the sub-job achieving the maximal rank is denoted as

$$q_j(y) = \max\{\, q \in \mathbb{R}^+ \,:\, R_j(q,y) = R_j(y) \,\}.$$

With these definitions, we define the Gittins index priority policy.

Algorithm 1. Gittins index priority policy (GIPP)

At any time t, process an unfinished job j with currently highest rank $R_j(y_j(t))$, where $y_j(t)$ denotes the amount of processing that has been done on job j by time t. Break ties by choosing the job with the smallest job index.

Theorem 1 ([11, 24, 30]). *The Gittins index priority policy (GIPP) solves the preemptive stochastic scheduling problem* $1 \mid pmtn \mid \mathbb{E}\left[\sum w_j C_j\right]$ *optimally.*

The following properties of the Gittins indices and the lengths of sub-jobs achieving the Gittins index are well known, see [7, 30]. In parts, they have been derived earlier in the scheduling context by [11] and [24].

Proposition 1 ([7, 30]) *Consider a job j that has been processed for y time units. Then, for any $0 < \gamma < q_j(y)$ holds*

$$R_j(y) \;\le\; R_j(y+\gamma)\,, \tag{1}$$

$$q_j(y+\gamma) \;\le\; q_j(y) - \gamma\,, \tag{2}$$

$$R_j(y + q_j(y)) \;\le\; R_j(y)\,. \tag{3}$$

Let us denote the sub-job of length $q_j(y)$ that causes the maximal rank $R_j(y)$, a *quantum* of job j. We now split a job j into a set of n_j quanta, denoted by tuples (j,i), for $i = 1,\dots,n_j$. The processing time y_{ji} that a job j has attained up to a quantum (j,i) and the length of each quantum, q_{ji}, are recursively defined as $y_{j1} = 0$, $q_{ji} = q_j(y_{ji})$, and $y_{j,i+1} = y_{j,i} + q_{ji}$. By Proposition 1(1), we know that, while processing a quantum, the rank of the job does not decrease, whereas Proposition 1(3) and the definition of $q_j(y)$ tell us that the rank is strictly lower at the beginning of the next quantum. Hence, once a quantum has been started Gipp will process it for its complete length or up to the completion of the job, whatever comes first. Thus, a job is preempted only at the end of a quantum. Obviously, the policy Gipp processes job quanta nonpreemptively in non-increasing order of their ranks.

Based on the definitions above, we define the set $H(j,i)$ of all quanta that preceed quantum (j,i) in the Gipp order. Let $\mathcal{Q}$ be the set of all quanta, i. e., $\mathcal{Q} = \{(k,l) \mid k = 1,\dots,n,\ l = 1,\dots,n_k\}$, then

$$H(j,i) = \{(k,l) \in \mathcal{Q} \mid R_k(y_{kl}) > R_j(y_{ji})\} \cup \{(k,l) \in \mathcal{Q} \mid R_k(y_{kl}) = R_j(y_{ji}) \wedge k \le j\}\,.$$

As the Gittins index of a job is decreasing with every finished quantum 1(3), we know that $H(j,h) \subseteq H(j,i)$, for $h \le i$. In order to uniquely relate higher priority quanta to one quantum of a job, we introduce the notation $H'(j,i) = H(j,i) \setminus H(j,i-1)$, where we define $H(j,0) = \emptyset$. Note that the quantum (j,i) is also contained in the set of its higher priority quanta $H'(j,i)$. In the same manner, we define the set of lower priority quanta as $L(j,i) = \mathcal{Q} \setminus H(j,i)$.

With these definitions and the observations above we can give a closed formula for the expected objective value of Gipp.

Lemma 2. *The optimal policy for the scheduling problem* $1 \mid pmtn \mid \mathbb{E}[\sum w_j C_j]$, Gipp, *achieves the expected objective value of*

$$\mathbb{E}[\,\textsc{Gipp}\,] = \sum_j w_j \sum_{i=1}^{n_j} \sum_{(k,l) \in H'(j,i)} \Pr[P_j > y_{ji} \wedge P_k > y_{kl}] \cdot I_k(q_{kl}, y_{kl}).$$

Proof. Consider a realization of processing times $p \in \Omega$ and a job j. Let i_p be the index of the quantum in which job j finishes, i. e., $y_{ji_p} < p_j \le y_{ji_p} + q_{ji_p}$. The policy Gipp processes quanta of jobs that have not completed nonpreemptively in non-increasing order of their ranks. Hence,

$$C_j(p) \;=\; \sum_{(k,l) \in H(j,i_p)\,:\,p_k > y_{kl}} \min\{q_{kl}, p_k - y_{kl}\}\,. \tag{4}$$

For an event $\mathcal{E}$, let $\chi(\mathcal{E})$ be an indicator random variable which equals 1 if and only if the event $\mathcal{E}$ occurs. The expected value of $\chi(\mathcal{E})$ equals then the probability with that the event $\mathcal{E}$ occurs, i. e., $\mathbb{E}[\chi(\mathcal{E})] = \Pr[\mathcal{E}]$. Additionally, we denote by ξ_{kl} the special indicator random variable for the event $P_k > y_{kl}$.

We take expectations on both sides of equation (4) over all realizations. This yields

$$
\begin{aligned}
\mathbb{E}\left[C_j\right] &= \mathbb{E}\left[\sum_{h:y_{jh}<P_j\le y_{j,h+1}} \sum_{(k,l)\in H(j,h):P_k>y_{kl}} \min\{q_{kl}, P_k - y_{kl}\}\right] \\
&= \mathbb{E}\left[\sum_{h=1}^{n_j} \chi(y_{jh} < P_j \le y_{j,h+1}) \sum_{(k,l)\in H(j,h)} \xi_{kl}\cdot \min\{q_{kl}, P_k - y_{kl}\}\right] \\
&= \mathbb{E}\left[\sum_{h=1}^{n_j} \chi(y_{jh} < P_j \le y_{j,h+1}) \sum_{i=1}^{h} \sum_{(k,l)\in H'(j,i)} \xi_{kl}\cdot \min\{q_{kl}, P_k - y_{kl}\}\right] \\
&= \mathbb{E}\left[\sum_{i=1}^{n_j}\sum_{h=i}^{n_j} \chi(y_{jh} < P_j \le y_{j,h+1}) \sum_{(k,l)\in H'(j,i)} \xi_{kl}\cdot \min\{q_{kl}, P_k - y_{kl}\}\right] \\
&= \mathbb{E}\left[\sum_{i=1}^{n_j} \chi(y_{ji} < P_j) \sum_{(k,l)\in H'(j,i)} \xi_{kl}\cdot \min\{q_{kl}, P_k - y_{kl}\}\right] \\
&= \mathbb{E}\left[\sum_{i=1}^{n_j} \xi_{ji} \sum_{(k,l)\in H'(j,i)} \xi_{kl}\cdot \min\{q_{kl}, P_k - y_{kl}\}\right]. \qquad (5)
\end{aligned}
$$

The equalities follow from an index rearrangement and the facts that $H(j,h) = \cup_{i=1}^{h} H'(j,i)$ for any h and that n_j is an upper bound on the number of quanta of job j.

For jobs $k \neq j$, the processing times P_j and P_k are independent random variables and thus, the same holds for their indicator random variables ξ_{ji} and ξ_{kl} for any i, l. Using linearity of expectation, we rewrite (5) as

$$
\begin{aligned}
&= \sum_{i=1}^{n_j} \sum_{(k,l)\in H'(j,i)} \mathbb{E}\left[\xi_{ji}\cdot\xi_{kl}\cdot \min\{q_{kl}, P_k - y_{kl}\}\right] \\
&= \sum_{i=1}^{n_j} \sum_{(k,l)\in H'(j,i)} \sum_{x} x\cdot \Pr\left[\xi_{ji}=\xi_{kl}=1 \wedge \min\{q_{kl}, P_k - y_{kl}\} = x\right] \\
&= \sum_{i=1}^{n_j} \sum_{(k,l)\in H'(j,i)} \sum_{x} x\cdot \Pr\left[\xi_{ji}=\xi_{kl}=1\right]\cdot \Pr\left[\min\{q_{kl}, P_k - y_{kl}\} = x \,|\, \xi_{kl}=1\right] \\
&= \sum_{i=1}^{n_j} \sum_{(k,l)\in H'(j,i)} \Pr\left[P_j > y_{ji} \wedge P_k > y_{kl}\right]\cdot \mathbb{E}\left[\min\{q_{kl}, P_k - y_{kl}\} \,|\, P_k > y_{kl}\right] \\
&= \sum_{i=1}^{n_j} \sum_{(k,l)\in H'(j,i)} \Pr\left[P_j > y_{ji} \wedge P_k > y_{kl}\right]\cdot I_k(q_{kl}, y_{kl}),
\end{aligned}
$$

where the third equality follows from conditional probability and the fact that either $j \neq k$, thus ξ_{ji} and ξ_{kl} are independent, or $(j,i) = (k,l)$ and thus the variables ξ_{ji} and ξ_{kl} are the same. Weighted summation over all jobs concludes the proof. □

3 A New Lower Bound on Parallel Machines

For the scheduling problem $\mathrm{P}\,|\,r_j, pmtn\,|\,\mathbb{E}\,[\sum w_j C_j\,]$ and most of its relaxations, optimal offline policies and the corresponding expected objective values are unknown. Therefore, we use lower bounds on the optimal value in order to compare the expected outcome of a policy with the expected outcome $\mathbb{E}\,[\,\mathrm{OPT}\,]$ of an unknown optimal policy OPT. The trivial bound $\mathbb{E}\,[\,\mathrm{OPT}\,] \geq \sum_j w_j(\,r_j + \mathbb{E}\,[\,P_j\,]\,)$ is unlikely to suffice proving constant approximation guarantees. However, we are not aware of any other bounds known for the general preemptive problem. LP-based approaches as they are used in the non-preemptive setting [18, 26, 4, 15, 21] do not transfer directly.

We derive in this section a new non-trivial lower bound for preemptive stochastic scheduling on parallel machines. We utilize the knowledge of Gipp's optimality for the single machine problem without release dates, see Theorem 1. To do so, we show first that the *fast single machine* relaxation introduced in deterministic online environment [3] applies in the stochastic setting as well.

Lemma 3. *Denote by $\mathcal{I}$ an instance of the problem $\mathrm{P}\,|\,r_j, pmtn\,|\,\mathbb{E}\,[\sum w_j C_j\,]$, and let $\mathcal{I}'$ be the same instance to be scheduled on a single machine of speed m times the speed of the machines used for scheduling instance $\mathcal{I}$. The optimal single machine policy OPT_1 yields an expected value $\mathbb{E}\,[\,\mathrm{OPT}_1(\mathcal{I}')\,]$ on instance $\mathcal{I}'$. Then, for any parallel machine policy Π holds*

$$\mathbb{E}\,[\,\Pi(\mathcal{I})\,] \;\geq\; \mathbb{E}\,[\,\mathrm{OPT}_1(\mathcal{I}')\,]\;.$$

Proof. Given a parallel machine policy Π, we provide a policy Π' for the fast single machine that yields an expected objective value $\mathbb{E}\,[\,\Pi'(\mathcal{I}')\,] \leq \mathbb{E}\,[\,\Pi(\mathcal{I})\,]$ for any instance $\mathcal{I}$. Then the lemma follows since an optimal policy OPT_1 on the single machine yields an expected objective value $\mathbb{E}\,[\,\mathrm{OPT}_1(\mathcal{I}')\,] \leq \mathbb{E}\,[\,\Pi'(\mathcal{I}')\,]$.

We construct policy Π' by letting its first decision point coincide with the first decision point of policy Π (the earliest release date). At any of its decision points, Π' can compute the jobs to be scheduled by policy Π and due to the fact that the processing times of all jobs are discrete random variables, it computes the earliest possible completion time of these jobs, in the parallel machine schedule. The next decision point of Π', is the minimum of these possible completion times and the next decision point of Π. Between two consecutive decision points of Π', the policy schedules the same set of jobs that Π schedules, for the same amount of time. This is possible as the single machine on which Π' operates works m times as fast.

In this way, we ensure that all job completions in the parallel machine schedule obtained by Π, coincide with a decision point of policy Π'. Moreover, as Π' schedules the same set of jobs as Π between two decision points, any job that completes its processing at a certain time t in the schedule of Π, will also be completed by time t in the schedule of Π'. □

With this relaxation, we derive a lower bound on the expected optimal value.

Theorem 2. *The expected value of an optimal policy* OPT *for the parallel machine problem $\mathcal{I}$ is bounded by*

$$\mathbb{E}\left[\,\mathrm{OPT}(\mathcal{I})\,\right] \;\geq\; \frac{1}{m}\;\sum_{j} w_j \sum_{i=1}^{n_j} \sum_{(k,\ell)\in H'(j,i)} \Pr\left[P_j > y_{ji} \wedge P_k > y_{k\ell}\right] \cdot I_k(q_{k\ell}, y_{k\ell})\,.$$

Proof. We consider the fast single machine instance $\mathcal{I}'$ as introduced in the previous lemma and relax it further to instance $\mathcal{I}'_0$ by setting all release dates equal. By Theorem 1, the resulting problem can be solved optimally by GIPP. With Lemma 3 we have then

$$\mathbb{E}\left[\,\mathrm{OPT}(\mathcal{I})\,\right] \;\geq\; \mathbb{E}\left[\,\mathrm{OPT}_1(\mathcal{I}')\,\right] \;\geq\; \mathbb{E}\left[\,\mathrm{GIPP}(\mathcal{I}'_0)\,\right]. \tag{6}$$

By Lemma 2 we know

$$\mathbb{E}\left[\,\mathrm{GIPP}(\mathcal{I}'_0)\,\right] = \sum_{j} w_j \sum_{i=1}^{n_j} \sum_{(k,l)\in H'(j,i)} \Pr\left[P'_j > y'_{ji} \wedge P'_k > y'_{kl}\right] \cdot I'_k(q'_{kl}, y'_{kl}), \tag{7}$$

where the dashes indicate the modified variables in the fast single machine instance $\mathcal{I}'_0$. By definition holds $P'_j = P_j/m$ for any job j as well as $\Pr\left[P_j > x\right] = \Pr\left[P'_j > x/m\right]$, and the probability $\Pr\left[P_j - y = x \,|\, P_j > y\right]$ for the remaining processing time after y units of processing remains the same on the fast machine. Moreover, the investment $I'_j(q', y')$ for any sub-job of length $q' = q/m$ of job $j \in I'$ after it has received $y' = y/m$ units of processing coincides with

$$\begin{aligned} I'_j(q', y') \;&=\; \mathbb{E}\left[\,\min\{P'_j - y', q'\} \,|\, P'_j > y'\,\right] \\ &=\; \frac{1}{m}\mathbb{E}\left[\,\min\{P_j - y, q\} \,|\, P_j > y\right] \;=\; \frac{1}{m}\, I_j(q, y)\,. \end{aligned}$$

We conclude that the partition of jobs into quanta in instance $\mathcal{I}$ immediately gives the partition for the fast single machine instance $\mathcal{I}'$. Each quantum (j, i) of job j maximizes the rank $R_j(q, y_{ji})$ and thus $q' = q/m$ maximizes the rank $R'_j(q/m, y/m) = R_j(q, y)/m$ on the single machine; thus, the quanta are simply shortened to an m-fraction of the original length, $q'_{ji} = q_{ji}/m$ and thus, $y'_{ji} = \sum_{l=1}^{i-1} q'_{jl} = y_{ji}/m$.

Combining these observations with (6) and (7) yields

$$\mathbb{E}\left[\,\mathrm{OPT}(\mathcal{I})\,\right] \;\geq\; \frac{1}{m}\;\sum_{j} w_j \sum_{i=1}^{n_j} \sum_{(k,l)\in H'(j,i)} \Pr\left[P_j > y_{ji} \wedge P_k > y_{kl}\right] \cdot I_k(q_{kl}, y_{kl})\,.$$

□

Theorem 2 above and Lemma 2 imply immediately

Corollary 1. *The lower bound on the optimal preemptive policy for parallel machine scheduling on an instance $\mathcal{I}$ equals an m-fraction of the expected value achieved by* GIPP *on the relaxed instance $\mathcal{I}_0$ without release dates but the same processing times to be scheduled on one machine, i. e.,*

$$\mathbb{E}\left[\,\mathrm{OPT}(\mathcal{I})\,\right] \;\geq\; \frac{\mathbb{E}\left[\,\mathrm{GIPP}(\mathcal{I}_0)\,\right]}{m}\,. \tag{8}$$

4 A 2-Approximation on Parallel Machines

Simple examples show that GIPP is not an optimal policy for scheduling problems with release dates and/or multiple machines. The following policy uses a modified version of GIPP where the rank of jobs is updated only after the completion of a quantum.

Algorithm 2. Follow Gittins Index Priority Policy (F-GIPP)

At any time t, process an available job j with highest rank $R_j(y_{j,k+1})$, where (j,k) is the last quantum of j that has completed, or $k=0$ if no quantum of job j has been completed.

Note, that the decision time points in this policy are release dates and any time, when a quantum or a job completes. In contrast to the original Gittins index priority policy, F-GIPP considers only the rank $R_j(y_{ji} = \sum_{k=1}^{i-1} q_{jk})$ that a job had before processing quanta (j,i) even if (j,i) has been processing for some time less than q_{ji}. Informally speaking, the policy F-GIPP updates the ranks only after quantum completions and then follows GIPP.

This policy applied to a deterministic scheduling instance coincides with the P-WSPT rule by Megow and Schulz [14], which is a generalization of Smith's optimal nonpreemptive single machine algorithm [28] to the deterministic counterpart of our scheduling problem without release dates. It has a competitive ratio of 2, and we prove the same performance guarantee for the more general stochastic online setting.

Theorem 3. *The online policy* F-GIPP *is a deterministic* 2*-approximation for the preemptive scheduling problem* $\mathrm{P} \,|\, r_j, pmtn \,|\, \mathbb{E}\,[\sum w_j C_j\,]$.

Proof. This proof incorporates ideas from [14] applied to the more complex stochastic setting. Fix a realization $p \in \Omega$ of processing times and consider a job j and its completion time $C_j^{\text{F-GIPP}}(p)$. Job j is processing in the time interval $[\,r_j, C_j^{\text{F-GIPP}}(p)\,]$. We split this interval into two disjunctive sets of sub-intervals, $T(j,p)$ and $\overline{T}(j,p)$, respectively. Let $T(j,p)$ denote the set of sub-intervals in which job j is processing and $\overline{T}(j,p)$ contains the remaining sub-intervals. Denoting the total length of all intervals in a set T by $|T|$, we have

$$C_j^{\text{F-GIPP}}(p) = r_j + |T(j,p)| + |\overline{T}(j,p)|\,.$$

In intervals of the set $\overline{T}(j,p)$, no machine is idle and F-GIPP schedules only quanta with a higher priority than (j,i_p), the final quantum of job j. Thus $|\overline{T}(j,p)|$ is maximized if all these quanta are scheduled between r_j and $C_j^{\text{F-GIPP}}(p)$ with an upper bound on the overall length of the total sum of quantum lengths on m machines. The total length of intervals in $T(j,p)$ is p_j and it follows

$$C_j^{\text{F-GIPP}}(p) \le r_j + p_j + \frac{1}{m} \cdot \sum_{\substack{(k,l)\in H(j,i_p):\\ p_k > y_{kl}}} \min\{q_{kl}, p_k - y_{kl}\}\,.$$

Weighted summation over all jobs and taking expectations on both sides give with the same arguments as in Lemma 2:

$$\sum_j w_j \mathbb{E}\left[C_j^{\text{F-GIPP}}\right] \le \sum_j w_j \left(r_j + \mathbb{E}\left[P_j\right]\right) + \frac{1}{m} \cdot \sum_j w_j \sum_{i=1}^{n_j} \sum_{(k,l)\in H'(j,i)} \Pr\left[P_j > y_{ji} \wedge p_k > y_{kl}\right] \cdot I_k(q_{kl}, y_{kl}) .$$

Finally, we apply the trivial lower bound $\mathbb{E}[\,\text{OPT}\,] \ge \sum_j w_j(r_j + \mathbb{E}[\,P_j\,])$ and Theorem 2, and the approximation result follows. □

In absence of release dates, our policy coincides with GIPP and is thus optimal on a single machine (Theorem 1). Nevertheless, for general input instances the approximation factor of 2 is best possible for F-GIPP which follows directly from a deterministic worst-case instance in [14].

Concluding remarks. In a full version of our paper [16], we introduce a second single machine policy, which deviates less from the original Gittins index priority rule than F-GIPP does. Thus, we use more information on the actual state of the set of known, unfinished jobs. This single machine policy can be immediately extended to the parallel machine setting by randomized machine assignment. For both policies, on the single and on multiple machines, we prove the performance guarantee of 2. Clearly, this result does not improve the approximation guarantee of of F-GIPP in the previous section. But, while the analysis of F-GIPP is tight, we conjecture that the true approximation ratio of the new policy is less than 2.

References

1. J. L. Bruno, P. J. Downey, and G. N. Frederickson. Sequencing tasks with exponential service times to minimize the expected flowtime or makespan. *Journal of the ACM*, 28:100–113, 1981.
2. D. Chazan, A. G. Konheim, and B. Weiss. A note on time sharing. *Journal of Combinatorial Theory*, 5:344–369, 1968.
3. C. Chekuri, R. Motwani, B. Natarajan, and C. Stein. Approximation techniques for average completion time scheduling. *SIAM Journal on Computing*, 31:146–166, 2001.
4. M.C. Chou, H. Liu, M. Queyranne, and D. Simchi-Levi. On the asymptotic optimality of a simple on-line algorithm for the stochastic single machine weighted completion time problem and its extensions, 2006. Operations Research, to appear.
5. E. G. Coffman, M. Hofri, and G. Weiss. Scheduling stochastic jobs with a two point distribution on two parallel machines. *Probability in the Engineering and Informational Sciences*, 3:89–116, 1989.
6. J. C. Gittins. Bandit processes and dynamic allocation indices. *Journal of the Royal Statistical Society, Series B*, 41:148–177, 1979.
7. J. C. Gittins. *Multi-armed Bandit Allocation Indices*. Wiley, New York, 1989.
8. R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

9. T. Kämpke. Optimal scheduling of jobs with exponential service times on identical parallel processors. *Operations Research*, 37(1):126–133, 1989.
10. A. Karlin, M. Manasse, L. Rudolph, and D. Sleator. Competitive snoopy paging. *Algorithmica*, 3:70–119, 1988.
11. A. G. Konheim. A note on time sharing with preferred customers. *Probability Theory and Related Fields*, 9:112–130, 1968.
12. J. Labetoulle, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinooy Kan. Preemptive scheduling of uniform machines subject to release dates. In W. R. Pulleyblank, editor, *Progress in Combinatorial Optimization*, pages 245–261. Academic Press, New York, 1984.
13. J. K. Lenstra, A. H. G. Rinooy Kan, and P. Brucker. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1:243–362, 1977.
14. N. Megow and A. S. Schulz. On-line scheduling to minimize average completion time revisited. *Operations Research Letters*, 32:485–490, 2004.
15. N. Megow, M. Uetz, and T. Vredeveld. Models and algorithms for stochastic online scheduling. *Mathematics of Operations Research, to appear*, 2006.
16. N. Megow and T. Vredeveld. Approximation results for preemptive stochastic online scheduling. Technical Report 8/2006, Technische Universität Berlin, April 2006.
17. R. H. Möhring, F. J. Radermacher, and G. Weiss. Stochastic scheduling problems I: General strategies. *ZOR - Zeitschrift für Operations Research*, 28:193–260, 1984.
18. R. H. Möhring, A. S. Schulz, and M. Uetz. Approximation in stochastic scheduling: the power of LP-based priority policies. *Journal of the ACM*, 46:924–942, 1999.
19. M. Pinedo. Stochastic scheduling with release dates and due dates. *Operations Research*, 31:559–572, 1983.
20. M. Pinedo. Off-line deterministic scheduling, stochastic scheduling, and online deterministic scheduling: A comparative overview. In J. Leung, editor, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, chapter 38. CRC Press, 2004.
21. A. S. Schulz. New old algorithms for stochastic scheduling. In S. Albers, R. H. Möhring, G. Ch. Pflug, and R. Schultz, editors, *Algorithms for Optimization with Incomplete Information*, number 05031 in Dagstuhl Seminar Proceedings, 2005.
22. A. S. Schulz and M. Skutella. The power of α-points in preemptive single machine scheduling. *Journal of Scheduling*, 5:121–133, 2002.
23. A. S. Schulz and M. Skutella. Scheduling unrelated machines by randomized rounding. *SIAM Journal on Discrete Mathematics*, 15:450–469, 2002.
24. K.C. Sevcik. Scheduling for minimum total loss using service time distributions. *Journal of the ACM*, 21:65–75, 1974.
25. R. A. Sitters. *Complexity and Approximation in Routing and Scheduling*. PhD thesis, Technische Universiteit Eindhoven, 2004.
26. M. Skutella and M. Uetz. Stochastic machine scheduling with precedence constraints. *SIAM Journal on Computing*, 34:788–802, 2005.
27. D. Sleator and R. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, 1985.
28. W. Smith. Various optimizers for single-stage production. *Naval Research Logistics Quarterly*, 3:59–66, 1956.
29. R. R. Weber. Scheduling jobs with stochastic processing requirements on parallel machines to minimize makespan or flow time. *Journal of Applied Probability*, 19:167–182, 1982.
30. G. Weiss. On almost optimal priority rules for preemptive scheduling of stochastic jobs on parallel machines. *Advances in Applied Probability*, 27:827–845, 1995.

Greedy in Approximation Algorithms*

Julián Mestre

Department of Computer Science
University of Maryland, College Park, MD 20742

Abstract. The objective of this paper is to characterize classes of problems for which a greedy algorithm finds solutions provably close to optimum. To that end, we introduce the notion of k-extendible systems, a natural generalization of matroids, and show that a greedy algorithm is a $\frac{1}{k}$-factor approximation for these systems. Many seemly unrelated problems fit in our framework, e.g.: b-matching, maximum profit scheduling and maximum asymmetric TSP.

In the second half of the paper we focus on the maximum weight b-matching problem. The problem forms a 2-extendible system, so greedy gives us a $\frac{1}{2}$-factor solution which runs in $O(m \log n)$ time. We improve this by providing two linear time approximation algorithms for the problem: a $\frac{1}{2}$-factor algorithm that runs in $O(bm)$ time, and a $\left(\frac{2}{3} - \epsilon\right)$-factor algorithm which runs in expected $O\left(bm \log \frac{1}{\epsilon}\right)$ time.

1 Introduction

Perhaps the most natural first attempt at solving any combinatorial optimization problem is to design a greedy algorithm. The underlying idea is simple: we make locally optimal choices hoping that this will lead us to a globally optimal solution. Needless to say that such an algorithm may not always work, therefore a natural question to ask is: for which class of problems does this approach work? A classical theorem due to Edmonds and Rado answers this question; to state this result we first need to define our problem more rigorously.

A *subset system* is a pair $(E, \mathcal{L})$, where E is a finite set of elements and $\mathcal{L}$ is a collection of subsets of E such that if $A \in \mathcal{L}$ and $A' \subseteq A$ then $A' \in \mathcal{L}$. Sets in $\mathcal{L}$ are called *independent*, and should be regarded as feasible solutions of our problem. Given a positive weight function $w : E \to \mathbb{R}^+$ there is a natural optimization problem associated with $(E, \mathcal{L})$ and w, namely that of finding an independent set of maximum weight. We want to study the following algorithm, which from now on we simply refer to as Greedy: start from the empty solution and process the elements in decreasing weight order, add an element to the current solution only if its addition preserves independence.

A matroid is a subset system $(E, \mathcal{L})$ for which the following property holds:

$$\forall\, A, B \in \mathcal{L} \text{ and } |A| < |B| \text{ then } \exists\, z \in B \setminus A \text{ such that } A + z^1 \in \mathcal{L}$$

* Research supported by NSF Awards CCR-01-05413 and CCF-04-30650, and the University of Maryland Dean's Dissertation Fellowship.

[1] The notation $A + z$ means $A \cup \{z\}$, likewise $A - z$ means $A \setminus \{z\}$.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 528–539, 2006.

Matroids were first introduced by Whitney [24] as an abstraction of the notion of independence from linear algebra and graph theory. Rado [21] showed that if a given problem has the matroid property then Greedy always finds an optimal solution. In turn, Edmonds [11] proved the other direction of the implication, i.e., if Greedy finds an optimal solution for *any* weight function defined on the elements then the problem must have the matroid property.

A rich theory of matroids exists, see [22, 18] for a thorough treatment of the subject. Many generalizations along two main directions have been proposed. One approach is to define a more general class of problems. Greedy no longer works, therefore alternative algorithms must be designed; examples of this are greedoids [15], two-matroid intersection [10], and matroid matching [17]. Another approach is to study structures where Greedy finds optimal solutions for some, but not all weight functions; symmetric matroids [7], sympletic matroids [6] and the work of Vince [23] are along these lines.

Although different in nature, both approaches have the same objective in mind: exact solutions. In this paper we study Greedy from the point of view of approximation algorithms. Our main contribution is the introduction of k-extendible systems, a natural generalization of matroids. We show that Greedy is a $\frac{1}{k}$-factor approximation for k-extendible systems.

Given a subset system $(E, \mathcal{L})$, Korte and Hausman [14] showed that for the maximization problem defined by $(E, \mathcal{L})$, Greedy achieves its worst approximation ratio on 0-1 weight functions. Consider the 0-1 function w_A defined as $w_A(x) = 1$ for $x \in A$ and 0 otherwise. The cost of the solution Greedy finds, comes from the elements in A the algorithm happens to pick, these elements form an independent set which is maximal with respect to A. Let γ_A be the ratio between the smallest and the largest maximal independent subsets of A. Notice that γ_A is the worst greedy can do on w_A. Let $\gamma = \min_{A \subseteq E} \gamma_A$. Korte and Hausman showed that Greedy is a γ-factor approximation for $(E, \mathcal{L})$.

While this result tells us how well Greedy performs on a particular system, in some cases it may be difficult to establish γ for a given combinatorial problem—which can be regarded as a class of systems, as every instance of the problem defines a system. Our k-extendible framework better highlights the structure of the problem and allows us to easily explain the performance of Greedy on seemingly unrelated problems such as b-matching, maximum profit scheduling and maximum asymmetric TSP. For some of these, an algorithm tailored to the specific problem yields a better approximation ratio than that offered by Greedy. This should not come as a surprise, after all Greedy is a generic algorithm that we can try on nearly every problem. The goal of this paper is to characterize those problems for which a simple greedy strategy produces nearly optimal solutions and to better understand its shortcomings. Along these lines is the recent work by Borodin et al. [5], who introduced the paradigm of priority algorithms, a formal class of algorithms that captures most greedy-like algorithms. Lower bounds on the approximation ratio any priority algorithm can achieve were derived for scheduling [5], set cover, and facility location problems [1].

In particular, our framework explains why Greedy produces $\frac{1}{2}$-approximate solutions for b-matching. Given a graph $G = (V, E)$ with n vertices and m edges and degree constraints $b : V \to \mathbb{N}$ for the vertices, a *b-matching* is a set of edges M such that for all $v \in V$ the number of edges in M incident to v, denoted by $\deg_M(v)$, is at most $b(v)$. Polynomial time algorithms exist to solve the problem optimally: A maximum size b-matching can be found in $O(nm \log n)$ time and maximum weight in $O\left(\sum b(v) \min(m \log n, n^2)\right)$ time; both results are due to Gabow [12]. Greedy on the other hand produces approximate solutions but has the advantage of being simple and much faster, running in just $O(m \log n)$ time. This time savings can be further improved. For instance, for maximum weight matching (the case where $b(v) = 1$ for all v) Preis [20] proposed a $\frac{1}{2}$-approximation algorithm which runs in linear time. Drake et al. [9] designed an alternative simpler algorithm that greedily finds disjoint heavy paths and keeps the best of the two matchings defined on the path; the same authors in later work [8] designed an algorithm with an approximation factor of $\frac{2}{3} - \epsilon$ which runs in $O\left(\frac{m}{\epsilon}\right)$ time. Finally, Pettie and Sanders [19] gave randomized and deterministic algorithms with the same approximation guarantee of $\frac{2}{3} - \epsilon$ which run in $O(m \log \frac{1}{\epsilon})$ time. We note that a better approximation ratio can be obtained using local search [2] or the limited-backtrack greedy scheme of Arora *et at* [3], albeit at a very high running time. The challenge here is to get a fast algorithm with a good approximation guarantee.

In the second half of the paper we explore this tradeoff for b-matching and provide a $\frac{1}{2}$-approximation which runs in $O(bm)$ time and a $\left(\frac{2}{3} - \epsilon\right)$-factor randomized algorithm that runs in expected $O\left(bm \log \frac{1}{\epsilon}\right)$ time, where $b = \max_u b(u)$. Our algorithms build upon the work of [9] and [19]. The main difficulty in extending previous results to b-matching is the way the optimal solution and the one produced by the algorithm are compared in the analysis. This was done by taking the symmetric difference of the two, which for matchings yields a collection of simple paths and cycles. Unfortunately this does not work for b-matching, a more careful pairing argument must be provided.

2 *k*-Extendible Systems

The following definitions are with respect to a given system $(E, \mathcal{L})$ and a particular weight function. Let $A \in \mathcal{L}$, we say B is an *extension* of A if $A \subseteq B$ and $B \in \mathcal{L}$. We denote by $\mathrm{OPT}(A)$ an extension of A with maximum weight. Note that $\mathrm{OPT}(\emptyset)$ is an independent set with maximum weight.

Definition 1. *The subset system $(E, \mathcal{L})$ is k-extendible if for all $C \in \mathcal{L}$ and $x \notin C$ such that $C + x \in \mathcal{L}$ and for every extension D of C there exists a subset $Y \subseteq D \setminus C$ with $|Y| \leq k$ such that $D \setminus Y + x \in \mathcal{L}$.*

Notice that if $x \in D$ or $C = D$ then the property holds trivially by letting $Y = \emptyset$, therefore we do not need to consider these two cases in our proofs.

Our goal is to characterize problems for which a greedy algorithm will produce good solutions. In Section 2.1 we show that Greedy is a $\frac{1}{k}$-factor approximation

for k-extendible systems. We also show a close relation between k-extendible systems and matroids, starting with the following theorem:

Theorem 1. *The system $(E, \mathcal{L})$ is a matroid if and only if is 1-extendible.*

Proof. First we prove the $\Rightarrow$ direction: given sets $C \subset D \in \mathcal{L}$ and an element $x \notin D$ we need to find Y such that $D \setminus Y + x$ is independent. Set $A = C + x$ and $B = D$. If $|A| = |B|$ then the two sets differ by one element, by setting $Y = B \setminus A$ we get the k-extendible property. Otherwise we can repeatedly apply the matroid property to add an element from $B \setminus A$ to A until $|A| = |B|$. Again $Y = D \setminus A$ has cardinality 1. Since $D \setminus Y + x = A \in \mathcal{L}$ we get that $(E, \mathcal{L})$ is 1-extendible.

Let us show the other direction. Given two independent sets A and B such that $|A| < |B|$, we need to find z. Notice that if $A \subseteq B$ we are done, any $z \in B \setminus A$ will do, this is because any subset of $B \in \mathcal{L}$ is independent, in particular $A + z$. Suppose then that $A \not\subseteq B$. The idea is to pick $x \in A \setminus B$ and then find, if needed, an element y in $B \setminus A$ such that $B - y + x \in \mathcal{L}$. Remove y from B, add x, and repeat until $A \subseteq B$, at this point return any element $z \in B \setminus A$.

Pick any x in $A \setminus B$, if $B + x \in \mathcal{L}$ we are done since we do not need to pick a y. Otherwise, set $C = A \cap B$ and $D = B$, since the system is 1-extendible there exists Y such that $D \setminus Y + x \in \mathcal{L}$. Moreover Y consists of exactly one element $y \in D \setminus C = B \setminus A$, which is exactly what we were looking for. □

2.1 Greedy

Given $(E, \mathcal{L})$ and $w : E \to \mathbb{R}^+$ a natural first attempt at finding a maximum weight independent set is to use the greedy algorithm on the right. Starting from an empty solution S, we try to add elements to S one at a time, in decreasing weight order. We add x to S only if $S + x$ is independent.

```
GREEDY(G, w)
1  sort edges in decreasing weight
2  S ← ∅
3  for x ∈ E in order
4  do if S + x ∈ L
5        then S ← S + x
6  return S
```

Corollary 1. *Greedy solves the optimization problem defined by $(E, \mathcal{L})$ for any weight function if and only if $(E, \mathcal{L})$ is 1-extendible.*

This follows from Theorem 1 and the work of Rado [21] and Edmonds [11]. Now we generalize one direction of this result for arbitrary k.

Theorem 2. *Let $(E, \mathcal{L})$ be k-extendible, Greedy is a $\frac{1}{k}$-factor approximation for the optimization problem defined by $(E, \mathcal{L})$ and any weight function w.*

Let $x_1, x_2, \ldots x_l$ be the elements picked by greedy, also let $S_0 = \emptyset, \ldots S_l$ be the successive solutions, that is $S_i = S_{i-1} + x_i$. To prove Theorem 2 we need the following lemma whose proof we defer for a moment.

Lemma 1. *If $(E, \mathcal{L})$ is k-extendible then the ith element x_i picked by Greedy is such that $w(OPT(S_{i-1})) \leq w(OPT(S_i)) + (k-1)w(x_i)$.*

Remember that we can express the optimal solution as $\mathrm{OPT}(\emptyset)$. Starting from S_0 we can apply Lemma 1 l times to get:

$$\begin{aligned} w(\mathrm{OPT}(S_0)) &\leq w(\mathrm{OPT}(S_l)) + (k-1)\sum_{i=1}^{l} w(x_i) \\ &= w(S_l) + (k-1)w(S_l) \\ &= k\, w(S_l). \end{aligned}$$

We can replace $w(\mathrm{OPT}(S_l))$ with $w(S_l)$ because the set S_l is maximal. Hence Greedy returns a solution S_l with cost at least $\frac{1}{k}$ that of the optimal solution. Now it all boils down to proving Lemma 1.

Notice that $\mathrm{OPT}(S_{i-1})$ is an extension of S_{i-1}. Since $S_{i-1} + x_i \in \mathcal{L}$, we can find $Y \subseteq \mathrm{OPT}(S_{i-1}) \setminus S_{i-1}$ such that $\mathrm{OPT}(S_{i-1}) \setminus Y + x_i \in \mathcal{L}$. Thus,

$$\begin{aligned} w(\mathrm{OPT}(S_{i-1})) &= w(\mathrm{OPT}(S_{i-1}) \setminus Y + x_i) + w(Y) - w(x_i), \\ &\leq w(\mathrm{OPT}(S_i)) + w(Y) - w(x_i). \end{aligned}$$

The second line follows because $\mathrm{OPT}(S_{i-1})\setminus Y + x_i$ is an extension of $S_{i-1} + x_i$ and $\mathrm{OPT}(S_i)$ is one with maximum weight. Now let us look at an element $y \in Y$, we claim that $w(y) \leq w(x_i)$. Suppose for the sake of contradiction that $w(y) > w(x_i)$. Since $y \notin S_{i-1}$ this means that y was considered by Greedy before x_i and was dropped. Therefore there exist $j \leq i$ such that $S_j + y \notin \mathcal{L}$, but $S_j + y \subseteq \mathrm{OPT}(S_{i-1}) \in \mathcal{L}$, a contradiction. All weights are positive, therefore $w(Y) \leq kw(x_i)$, and the lemma follows.

2.2 Examples of k-Extendible Systems

Now we show that many natural problems fall in our k-extendible framework.

Maximum weight b-matching: Given a graph $G = (V, E)$ and degree constraints $b : V \to \mathbb{N}$ for the vertices, a *b-matching* is a set of edges M such that for all $v \in V$ the number of edges in M incident to v, denoted by $\deg_M(v)$, is at most $b(v)$.

Theorem 3. *The subset system associated with b-matching is 2-extendible.*

Proof. Let $C + (u, v)$ and D be valid solutions, where $C \subseteq D$ and $(u, v) \notin D$. We know that $\deg_C(u) < b(u)$ and $\deg_C(v) < b(v)$, otherwise $C + (u, v)$ would not be a valid solution. Now if $\deg_D(u) = b(u)$ we can find an edge in $D \setminus C$ incident to u, add this edge to Y and do the same for the other endpoint. Clearly $D \setminus Y + (u, v) \in \mathcal{L}$ and $|Y| \leq 2$, therefore the system is 2-extendible. □

Maximum profit scheduling: We are to schedule n jobs on a single machine. Each job i has release time r_i, deadline d_i, and profit w_i, all positive integers. Every job takes the same amount of time $L \in Z^+$ to process. (See [4] for a 2-approximation algorithm when the job lengths are arbitrary.) Our objective is to find a non-preemptive schedule that maximizes the weight of the jobs done on time. A job i is done on time if it starts and finishes in the interval $[r_i, d_i]$.

Theorem 4. *The subset system associated with maximum profit scheduling is 1-extendible when $L = 1$.*

Proof. Let $C + i$ be a feasible set of jobs, and D and extension of C. A schedule for a certain set of jobs can be regarded as matching between those jobs and time slots. Let M_1 and M_2 be the matchings for $C + i$ and D respectively. The set $M_1 \cup M_2$ contains a path starting on i ending on a job $j \in D \setminus C$ or a time slot t. Alternating the edges of M_2 along the path we get a schedule for $D + i - j$ in the first case, and for $D + i$ in the latter. □

For $L > 1$ we model the problem with a slightly different subset system. Let the elements of E be pairs (i, t) where t denotes the time job i is scheduled, and $r_i \leq t \leq d_i - L$. A set of elements is independent if it specifies a feasible schedule. Greedy considers the jobs in decreasing weight and adds the job being processed *somewhere* in the current schedule, if no place is available the job is dropped.

Theorem 5. *The subset system described above for maximum profit scheduling is 2-extendible for any $L > 1$.*

Proof. Let $C + (i, t)$ be a feasible schedule and D and extension of C. Adding i at time t to D may create some conflicts, which can be fixed by removing the jobs i overlaps with. Since all jobs have the same length, job i overlaps with at most two other jobs. □

Maximum asymmetric traveling salesman problem: We are given a complete directed graph with non-negative weights and we must find a maximum weight tour that visits every city exactly once. The problem is NP-hard; the best known approximation factor for it is $\frac{5}{8}$ [16].

The elements of our subset system are the directed edges of the complete graph; a set is independent if its edges form a collection of vertex disjoint paths or a cycle that visits every vertex exactly once.

Theorem 6 ([13]). *The subset system for maximum ATSP is 3-extendible.*

Proof. As usual let $C + (x, y)$ be independent, and D be an extension of C. First remove from D the edges (if any) out of x and into y, these are clearly at most two and not in C. If we add (x, y) to D then every vertex has in-degree and out-degree at most one, but there may be a non-Hamiltonian cycle which uses (x, y). There must be an edge in the cycle, not in C, that we can remove to break it. Therefore we need to remove at most three edges in total. □

Matroid intersection: This last theorem shows a nice relationship between matroids and k-extendible systems.

Theorem 7. *The intersection of k matroids is k-extendible*

Proof. Let $(E, \mathcal{L}_i)$ for $1 \leq i \leq k$ be our k matroids and let $\mathcal{L} = \cap_i \mathcal{L}_i$. We need to show that for every $C \subseteq D \in \mathcal{L}$ and $x \notin C$ such that $C + x \in \mathcal{L}$ there exist $Y \subseteq D \setminus C$ with at most k element such that $D \setminus Y + x \in \mathcal{L}$.

Since the above sets are in $\mathcal{L}$ they are also in $\mathcal{L}_i$. By Theorem 1 these individual matroids are 1-extendible, therefore we can find Y_i with at most one element such $D \setminus Y_i + x \in \mathcal{L}_i$. Set $Y = \cup_i Y_i$, clearly $|Y| \leq k$ and for all i we have $D \setminus Y + x \in \mathcal{L}_i$, which implies independence with respect to $\mathcal{L}$. □

3 A Linear Time $\frac{1}{2}$-Approximation for b-Matching

Because maximum weight b-matching can be solved exactly in $O\left(\sum b(v) \min(m \log n, n^2)\right)$ time [12], Greedy should be regarded as a tradeoff: we sacrifice optimality in order to get a much simpler algorithm which runs in $O(m \log n)$ time. This tradeoff can be further improved to obtain a linear time $\frac{1}{2}$-approximation, our solution builds upon the work of Drake and Hougardy [9]. Let $b = \max_{v \in V} b(v)$, in this section we show:

Theorem 8. *There is a $O(bm)$ time $\frac{1}{2}$-approximation algorithm for b-matching.*

The main procedure of our algorithm, LINEAR-MAIN, iteratively calls FIND-WALK, which greedily finds a heavy walk. Starting at some vertex u we take the heaviest edge (u, v) out of u, delete it from the graph, reduce $b(u)$ by one, and repeat for v. If at some point the $b(\cdot)$ value of a vertex becomes zero we delete all the remaining edges incident to it.

As we construct the walk we decrease the $b(\cdot)$ value of the vertices in the walk. Except for the endpoints every node will have its $b(\cdot)$ value decreased by 1 for every two edges in the walk incident to it. This means that M, the set of all walks, is not a valid solution as we can only guarantee that $\deg_M(u) \leq 2b(u)$ for every vertex u.

Now consider choosing every other edge in a walk starting with the first edge. For any vertex the number of chosen edges incident to it is at most how much its $b(\cdot)$ value was decreased while finding this walk. The same holds for the complement of this set, that is, picking every other edge starting with the second edge. We can therefore split M into two sets M_1 and M_2 by taking alternating edges of individual walks. These are valid solutions to our problem since for every vertex u we have $\deg_{M_i}(u) \leq b(u)$. Because $M = M_1 \cup M_2$, picking the one with

```
LINEAR-MAIN(G, w)
1  M ← ∅
2  while ∃ u ∈ V such that
           b(u) > 0 and deg(u) > 0
3  do M ← M + FIND-WALK(u)
4  split M into M_1 and M_2
5  return argmax{w(M_i)}
```

```
FIND-WALK(u)
1  b(u) ← b(u) − 1
2  if deg(u) = 0
3     then return ∅
4  let (u, v) be the heaviest edge out of u
5  remove (u, v) from G
6  if b(u) = 0
7     then remove all edges incident to u
8  return (u, v) + FIND-WALK(v)
```

Fig. 1. A linear time $\frac{1}{2}$ approximation for b-matching

maximum weight we are guaranteed a solution with weight at least $\frac{w(M)}{2}$. We now concentrate our effort in showing that $w(M)$ is an upper bound on the cost of the optimal solution.

Let M_{OPT} be the optimal solution. We can imagine including an additional step in the FIND-WALK(u) function in which an edge $e \in M_{OPT}$ is assigned to the heavy edge (u, v): If $(u, v) \in M_{OPT}$ then we assign it to itself, otherwise we pick any edge $e \in M_{OPT}$ incident to u. In either case after e is assigned we remove it from M_{OPT}, so that it is not later assigned to a different edge.

It may be that some edges in M do not receive any edge from M_{OPT}, but can an edge in M_{OPT} be left unassigned? The following lemma answers this question and relates the cost of the two edges.

Lemma 2. *The modified* FIND-WALK *procedure assigns every edge $e \in M_{OPT}$ to a unique edge $(u, v) \in M$, furthermore $w(e) \leq w(u, v)$.*

Proof. Suppose, for the sake of contradiction, that $(x, y) \in M_{OPT}$ was not assigned. It is easy to see that if the $b(\cdot)$ value of some vertex u becomes 0 then all edges in M_{OPT} incident to u must be assigned. Thus when the algorithm terminated $b(x), b(y) > 0$ and $\deg(x) = \deg(y) = 0$. Therefore the edge (x, y) must have been deleted from the graph because it was traversed (chosen in M). In this case we should have assigned (x, y) to itself. We reached a contradiction, therefore all edges in M_{OPT} are assigned a unique edge in M.

If (x, y) was assigned to itself then the lemma follows, suppose then that it got assigned to (x, v) in the call FIND-WALK(x). Notice that at the moment the call was made $b(x), b(y) > 0$. If at this moment (x, y) was present in the graph the lemma follows as (x, v) is the heaviest edge out of x. We claim this is the only alternative. If (x, y) had been deleted before it would be because it was traversed and thus it should have been assigned to itself. □

An immediate corollary of Lemma 2 is that $w(M_{OPT}) \leq w(M)$, which as mentioned implies the algorithm returns a solution with cost at least $\frac{w(M_{OPT})}{2}$. Now we turn our attention to the time complexity.

The running time is dominated by the time spent finding heavy edges. This is done by scanning the adjacency list of the appropriate vertex. An edge (x, y) may be considered several times while looking for a heavy edge out of x and y. The key observation is that this can happen at most $b(x) + b(y)$ times. Each time we reduce the value of either endpoint by one, when one of them reaches 0 all edges incident to that endpoint are deleted and after that (x, y) is never considered again. Adding up over all edges we get a total time of $O(bm)$.

4 A Randomized $\left(\frac{2}{3} - \epsilon\right)$-Factor Algorithm

In this section we generalize ideas from Pettie and Sander [19] to improve the approximation ratio of our linear time algorithm. We will develop a randomized algorithm that returns a solution with expected weight at least $\left(\frac{2}{3} - \epsilon\right) w(M_{OPT})$ and runs in expected $O\left(bm \log \frac{1}{\epsilon}\right)$ time.

```
LINEAR-RANDOM(G, w)
 1  M ← ∅
 2  do
 3      pick a vertex u uniformly at random
 4      with prob deg_M(u) / b do
 5          pick (u, v) ∈ M uniformly at random
 6          find max-benefit compatible piece P about (u, v)
 7          M ← M ⊕ P
 8      with prob (b(u) − deg_M(u)) / b do
 9          find max-benefit compatible arm A out of u
10          M ← M ⊕ A
11  repeat k times
```

Fig. 2. A linear time $\left(\frac{2}{3}-\epsilon\right)$-factor algorithm for b-matching

Before describing the algorithm we need to define a few terms, all of which are with respect to a given solution M. An edge e is *matched* if $e \in M$ otherwise we say e is *free*. A set of edges S can be used to update the matching by taking the symetric difference of M and S denoted by $M \oplus S = (M \cup S) \setminus (M \cap S)$. The set S is said to be *compatible* with M if $M \oplus S$ is a valid b-matching.

Our algorithm works by iteratively finding a compatible set of edges and updating our current solution M with it. To keep the running time low we only look for *arms* and *pieces*. An *arm* A out of a vertex u consists of a free edge (u, x) followed, maybe, by a matched edge (x, y). The *benefit* of A is defined as $w(u, x) - w(x, y)$, note that $\text{benefit}(A) = w(M \oplus A) - w(M)$. Let $(u, v) \in M$, a *piece* P about (u, v) consists of the edge (u, v), and, possibly, of arms A_u and A_v out of u and v. The benefit of the piece is defined as $\text{benefit}(A_u) + \text{benefit}(A_v) - w(u, v)$. Notice that if A_u and A_v use the same matched edge then $\text{benefit}(P) < w(M \oplus P) - w(M)$, otherwise these two quantities are the same.

We now describe in detail an iteration of our algorithm. First we pick a vertex u uniformly at random. Then we probabilistically decide to either: choose an edge $(u, v) \in M$ and augment M using a max-benefit compatible piece about (u, v), augment M with a max-benefit compatible arm out of u, or simply do nothing. See Fig. 2 for the exact probabilities of these events. This is repeated k times, the parameter k will be determined later to obtain:

Theorem 9. *The procedure* LINEAR-RANDOM *finds a b-matching in $O\left(bm \log \frac{1}{\epsilon}\right)$ time with expected weight at least $\left(\frac{2}{3}-\epsilon\right) w(M_{OPT})$.*

Let us first prove the approximation ratio of LINEAR-RANDOM. Our plan is to construct a set Q of pieces and arms with benefit at least $2w(M_{OPT}) - 3w(M)$ and then argue that the expected gain of each iteration is a good fraction of this. Note that if $2w(M_{OPT}) - 3w(M) \leq 0$ then M is already a $\frac{2}{3}$-approximate solution. In what follows we assume without loss of generality that M_{OPT} and M are disjoint—any overlap only makes our bounds stronger.

In order to construct Q we need to pair edges of M_{OPT} and M. Every edge $(u, v) \in M_{OPT}$ is paired with $(u, x) \in M$ via u and $(v, y) \in M$ via v in such a way that every edge in M is paired with at most two edges, one via each endpoint. If $\deg_{M_{OPT}}(u) > \deg_M(u)$ then the excess of M_{OPT} edges are assigned to u. Thus every edge $(u, v) \in M_{OPT}$ is paired/assigned exactly twice, once via each endpoint.

For every edge $(u, x) \in M$ we build a piece P by finding arms A_u and A_x out of u and x. To construct A_u follow, if any, the edge $(u, y) \in M_{OPT}$ paired with (u, x) via u, then take, if any, the edge $(y, z) \in M$ paired with (u, y) via y. A similar procedure is used to construct A_x. Finally we assign P to vertex u and add it to Q. Also for every $u \in V$ which has been assigned edges $(u, v) \in M_{OPT}$ we grow an arm A out of u using (u, v). These arms are assigned to u and added to Q.

Every edge in M_{OPT} appears in exactly two of the pieces and arms in Q, on the other hand every edge in M appears at most three times. Therefore the benefit of Q is at least $2w(M_{OPT}) - 3w(M)$.

How many pieces/arms can be assigned to a single vertex u? At most $\deg_M(u)$ pieces, one per $(u, x) \in M$, and at most $b(u) - \deg_M(u)$ arms, one per $(u, v) \in M_{OPT}$ which did not get paired up with M edges via u. A simple case analysis shows that all these pieces and arms are compatible with M. Therefore the expected benefit of the piece or arm picked in any given iteration is:

$$\begin{aligned}
\mathrm{E}[\text{benefit}] &= \frac{1}{n} \sum_{u \in V} \frac{b(u) - \deg_M(u)}{b} \text{max-arm}(u) + \sum_{(u,v) \in M} \frac{1}{b} \text{max-piece}(u, v) \\
&\geq \frac{1}{bn} \sum_{u \in V} \text{benefit of pieces/arms assigned to } u \\
&\geq \frac{1}{bn} \text{benefit}(Q) \\
&\geq \frac{3}{bn} \left(\frac{2}{3} w(M_{OPT}) - w(M) \right)
\end{aligned}$$

From this inequality we can derive the following lemma which is very similar to Lemma 3.3 from [19], we include its proof for completeness.

Lemma 3. *After running* LINEAR-RANDOM *for k iterations M has an expected weight of at least $\frac{2}{3} w(M_{OPT})(1 - e^{-\frac{3k}{bn}})$*

Proof. Let $X_i = \frac{2}{3} w(M_{OPT}) - w(M_i)$, where M_i is the matching we get at the end of the ith iteration. From the above inequality and the fact that the gain of each iteration is at least as much as the benefit of the piece/arm found we can infer that $\mathrm{E}[X_{i+1} | X_i] \leq X_i - \frac{3}{bn} X_i$. Thus $\mathrm{E}[X_{i+1}] \leq \mathrm{E}[X_i] \left(1 - \frac{3}{bn}\right)$, and

$$\mathrm{E}[X_k] \leq \mathrm{E}[X_0] \left(1 - \frac{3}{bn}\right)^k \leq \frac{2}{3} w(M_{OPT})\, e^{-\frac{3k}{bn}}.$$

By setting $k = \frac{bn}{3} \log \frac{1}{\epsilon}$ we get a matching with expected cost at least $\left(\frac{2}{3} - \epsilon\right) w(M_{OPT})$. Let us now turn our attention to the running time.

To compute a max benefit arm out of a vertex u we follow free edges (u, v) and if $\deg_M(v) = b(v)$ we scan the list of matched edges incident to v to find the lightest such edge; among the arms found we return the best. Notice that this can take as much as $O(b \deg(v))$ time. Suppose now, that we already had computed for every vertex which is the lightest matched edge incident to it, then the task can be carried out in just $O(\deg(v))$ time.

To produce a max benefit piece about (u, v) we can try finding max benefit arms out of u and v in $O(\deg(u) + \deg(v))$ time. This unfortunately does not always work as the resulting piece may not be compatible, consider finding arms $\{(u, x)\}$ and $\{(v, x)\}$ with $\deg_M(x) = b(x) - 1$, or $\{(u, x), (x, z)\}$ and $\{(v, x), (x, z)\}$ with $\deg_M(x) = b(x)$; both arms are compatible by themselves, but x cannot take both at once. If this problem arises, it can be solved by taking the best arm for u and the second best arm for v, or the other way around, and keeping the best pair. To find the second best arm we need to have access to the second lightest matched edge incident to any vertex.

Once we found our piece/arm we have to update the matching. This may change the lightest matched edges incident to vertices on the piece/arm. Since there are most 6 such vertices the update can be carried out in $O(b)$ time. The expected work done in a single iteration is given by:

$$\begin{aligned}
\mathrm{E}[\text{work}] &\leq \frac{1}{n} \sum_{u \in V} \frac{b(u) - \deg_M(u)}{b} (\deg(u) + b) + \sum_{(u,v) \in M} \frac{1}{b} (\deg(u) + \deg(v) + b) \\
&\leq \frac{1}{n} \sum_{v \in V} \deg(u) + b(u) + \sum_{(u,v) \in M} \frac{1}{b} \deg(v) \leq \frac{3}{n} \sum_{u \in V} \deg(u) = \frac{6m}{n}
\end{aligned}$$

The third inequality assumes $b(u) \leq \deg(u)$. If this is not the case we can just set $b(u)$ to be $\deg(u)$ which does not change the optimal solution.

There are $k = \frac{bn}{3} \log \frac{1}{\epsilon}$ iterations each taking $O(\frac{m}{n})$ time, by linearity of expectation the total expected running time is $O(bm \log \frac{1}{\epsilon})$.

5 Conclusion

We introduced the notion of k-extendible systems which allowed us to explain the performance of the greedy algorithm on seemingly disconnected problems. We also provided better approximation algorithms for b-matching, a specific problem that falls in our framework. It would be interesting to improve the approximation factor of other problems in this class beyond $\frac{1}{k}$.

Acknowledgments. Thanks to Hal Gabow and Allan Borodin for their encouraging words and for providing references to recent work on approximating maximum weight matching and priority algorithms. Special thanks to Samir Khuller for pointing out the problem and providing comments on earlier drafts.

References

1. S. Angelopoulos and A. Borodin. The power of priority algorithms for facility location and set cover. *Algorithmica*, 40(4):271–291, 2004.
2. E. M. Arkin and R. Hassin. On local search for weighted k-set packing. *Mathematics of Operations Research*, 23(3):640–648, 1998.
3. V. Arora, S. Vempala, H. Saran, and V. V. Vazirani. A limited-backtrack greedy schema for approximation algorithms. In *FSTTCS*, pages 318–329, 1994.
4. A. Bar-Noy, R. Bar-Yehuda, A. Freund, J. S. Naor, and B. Schieber. A unified approach to approximating resource allocation and scheduling. *Journal of the ACM*, 48(5):1069–1090, 2001.
5. A. Borodin, M. N. Nielsen, and C. Rockoff. (Incremental) Priority algorithms. *Algorithmica*, 37(4):295–326, 2003.
6. A. V. Borovik, I. Gelfand, and N. White. Symplectic matroid. *Journal of Algebraic Combinatorics*, 8:235–252, 1998.
7. A. Bouchet. Greedy algorithm and symmetric matroids. *Mathematical Programming*, 38:147–159, 1987.
8. D. E. Drake and S. Hougardy. Improved linear time approximation algorithms for weighted matchings. In *APPROX*, pages 14–23, 2003.
9. D. E. Drake and S. Hougardy. A simple approximation algorithm for the weighted matching problem. *Information Processing Letters*, 85:211–213, 2003.
10. J. Edmonds. Minimum partition of a matroid into independent subsets. *J. of Research National Bureau of Standards*, 69B:67–77, 1965.
11. J. Edmonds. Matroids and the greedy algorithm. *Mathematical Programming*, 1:127–36, 1971.
12. H. N. Gabow. An efficient reduction technique for degree-constrained subgraph and bidirected network flow problems. In *STOC*, pages 448–456, 1983.
13. T. A. Jenkyns. The greedy travelling salesman's problem". *Networks*, 9:363–373, 1979.
14. B. Korte and D. Hausmann. An analysis of the greedy algorithm for independence systems. *Ann. Disc. Math.*, 2:65–74, 1978.
15. B. Korte and L. Lovász. Greedoids—a structural framework for the greedy algorithm. In *Progress in Combinatorial Optimization*, pages 221–243, 1984.
16. M. Lewenstein and M. Sviridenko. Approximating asymmetric maximum TSP. In *SODA*, pages 646–654, 2003.
17. L. Lovász. The matroid matching problem. In *Algebraic Methods in Graph Theory*, Colloquia Mathematica Societatis Janos Bolyai, 1978.
18. J. G. Oxley. *Matroid Theory*. Oxford University Press, 1992.
19. S. Pettie and P. Sanders. A simpler linear time $2/3-\epsilon$ approximation to maximum weight matching. *Information Processing Letters*, 91(6):271–276, 2004.
20. R. Preis. Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs. In *STACS*, pages 259–269, 1999.
21. R. Rado. A theorem on independence relations. *Quart. J. Math.*, 13:83–89, 1942.
22. A. Schrijver. *Combinatorial Optimization*. Springer, 2003.
23. A. Vince. A framework for the greedy algorithm. *Discrete Applied Mathematics*, 121(1-3):247–260, 2002.
24. H. Whitney. On the abstract properties of linear dependence. *American Journal of Mathematic*, 57:509–533, 1935.

I/O-Efficient Undirected Shortest Paths with Unbounded Edge Lengths*

(Extended Abstract)

Ulrich Meyer[1,**] and Norbert Zeh[2,***]

[1] Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, Saarbrücken, 66123, Germany
umeyer@mpi-sb.mpg.de
[2] Faculty of Computer Science, Dalhousie University, 6050 University Ave, Halifax, NS B3H 1W5, Canada
nzeh@cs.dal.ca

Abstract. We show how to compute single-source shortest paths in undirected graphs with non-negative edge lengths in $\mathcal{O}(\sqrt{nm/B}\log n + MST(n,m))$ I/Os, where n is the number of vertices, m is the number of edges, B is the disk block size, and $MST(n,m)$ is the I/O-cost of computing a minimum spanning tree. For sparse graphs, the new algorithm performs $\mathcal{O}((n/\sqrt{B})\log n)$ I/Os. This result removes our previous algorithm's dependence on the edge lengths in the graph.

1 Introduction

Let $G = (V, E)$ be a graph, let s be a vertex of G, called the *source vertex*, and let $\ell : E \to \mathbb{R}^+$ be an assignment of non-negative real lengths to the edges of G. The *single-source shortest-path* (SSSP) problem is to find, for every vertex $v \in V$, the distance, $\text{dist}_G(s, v)$, from s to v, that is, the length of a shortest path from s to v in G. We focus mostly on the equivalent *closest-source shortest-path* (CSSP) problem: In addition to the input for SSSP, let $w : V \to \mathbb{R}^+$ be an assignment of non-negative *weights* to the vertices of G. Then compute for every vertex $x \in G$, its distance $D(x) = \min\{w(y) + \text{dist}_G(y, x) \mid y \in G\}$ from the closest source. If y is a vertex such that $w(y) + \text{dist}_G(y, x) = D(x)$, a *shortest path to* x, denoted $\pi(x)$, is a path of length $\text{dist}_G(y, x)$ from y to x. The classical SSSP-algorithm for general graphs is Dijkstra's algorithm [6], which has seen many improvements, particularly for undirected graphs with integer or float edge lengths [12, 13], and undirected graphs with real edge lengths [11]. When applied to massive graphs that do not fit in memory and are stored on disk, however, Dijkstra's algorithm and its improved variants perform poorly. This is because they access the data in a random fashion.

More recently, much work has focused on SSSP in massive graphs. These algorithms are analyzed in the *I/O-model* [1], which assumes that the computer

* For more details, see [10].
** Research supported by DFG grant ME 2088/1-3.
*** Research supported by NSERC and CFI.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 540–551, 2006.

has a main memory that can hold M vertices or edges and that the graph is stored on disk. In order to process the graph, pieces of it have to be loaded into memory, which happens in blocks of B consecutive data items. Such a transfer is referred to as an *I/O-operation* (I/O). The complexity of an algorithm is the number of I/Os it performs; e.g., $\text{sort}(n) = \Theta((n/B)\log_{M/B}(n/B))$ I/Os to sort n numbers [1].

Little is known about solving SSSP in general *directed* graphs I/O-efficiently. For *undirected* graphs, the algorithm of Kumar and Schwabe (*KS-SSSP*) [7] performs $\mathcal{O}(n+(m/B)\log(n/B))$ I/Os. For dense graphs, the second term dominates; but for sparse graphs, the I/O-bound becomes $\mathcal{O}(n)$. The SSSP-algorithm of Meyer and Zeh (*MZ-SSSP*) [9] extends the ideas of [8] for breadth-first search (BFS) to graphs with edge lengths between 1 and W, leading to an I/O-bound of $\mathcal{O}(\sqrt{nm\log W/B}+MST(n,m))$, where $MST(n,m)$ is the I/O-cost of computing a minimum spanning tree.[1] This paper removes the algorithm's dependence on W using a number of new ideas, proving the following result:

Theorem 1. *SSSP in an undirected graph with n vertices, m edges, and non-negative*[2] *edge lengths can be solved in $\mathcal{O}(\sqrt{nm/B}\log n + MST(n,m))$ I/Os.*

Note that for sparse graphs, the cost of our algorithm is $\mathcal{O}((n/\sqrt{B})\log n)$ I/Os. The rest of the paper is organized as follows: Section 2 discusses the ideas of KS-SSSP and MZ-SSSP that are reused in our algorithm. Section 3 augments MZ-SSSP to make it independent of the edge lengths, assuming an appropriate graph partition. The algorithm's complexity now depends on a parameter of the partition, called its depth. Section 4 describes a recursive shortest-path algorithm that uses another partition into "well-separated" subgraphs, allowing the computation of shortest paths in the whole graph using nearly independent computations on these subgraphs. Section 5 argues that any graph can be partitioned into such well-separated subgraphs, while at the same time ensuring that each has a partition of small depth in the sense of Sect. 3. This allows the two approaches from Sects. 3 and 4 to be combined to obtain an efficient CSSP-algorithm (which also solves SSSP) as stated in Thm. 1 above.

2 Previous Work: KS-SSSP and MZ-SSSP

KS-SSSP. KS-SSSP [7] is an I/O-efficient version of Dijkstra's algorithm. It uses a priority queue Q to maintain the tentative distances of all vertices and retrieves the vertices one by one from Q, by increasing tentative distances. After retrieving a vertex x from Q, x is *visited*, that is, its incident edges are *relaxed*, where the relaxation of an edge xy replaces the tentative distance $d(y)$ of y with $\min(d(y), d(x)+\ell(xy))$; this is reflected by updating the priority of y in Q.

[1] The current bounds for $MST(n,m)$ are $\mathcal{O}(\text{sort}(m)\log\log(nB/m))$ deterministically [2] and $\mathcal{O}(\text{sort}(m))$ randomized [5].

[2] In this paper, it is assumed that edge lengths are strictly positive. Length-0 edges can be handled by treating each connected component of the subgraph they induce as a single vertex.

The main contribution of KS-SSSP is an I/O-efficient priority queue that supports Update(x, p), Delete(x), and DeleteMin operations, each in amortized $\mathcal{O}((1/B)\log(n/B))$ I/Os. The latter two respectively delete x or the item with minimum priority from the priority queue. The former replaces x's current priority p_x with $\min(p_x, p)$ if $x \in Q$. If $x \notin Q$ and x has never been in Q, it is inserted with priority p. If x has been in Q before, but has been deleted, the operation does nothing! This particular behaviour of update operations is supported only for SSSP-computations in undirected graphs that visit vertices by increasing distances, because information about the structure of the resulting update sequence is used to ensure this behaviour (see [7] for details). As our algorithms visit vertices out of order, more effort is required to ensure this behaviour in our algorithms. Details appear in the full paper.

Given this fairly powerful priority queue, visiting a vertex x reduces to scanning its adjacency list $E(x)$ and performing an Update$(y, d(x)+\ell(xy))$ operation on Q for *every* edge $xy \in E(x)$. Thus, KS-SSSP performs $\mathcal{O}(m)$ priority queue operations, which cost $\mathcal{O}((m/B)\log(n/B))$ I/Os, and it spends $\mathcal{O}(1+\deg(x)/B)$ I/Os to retrieve the adjacency list of each vertex x, $\mathcal{O}(n+m/B)$ I/Os in total. For sparse graphs, the bottleneck of KS-SSSP is thus the random accesses to the adjacency lists. This problem is addressed by MZ-SSSP and by our new algorithm.

MZ-SSSP. MZ-SSSP partitions the vertex set of G into $q = \mathcal{O}(n/\mu)$ carefully chosen sets $V_1, \ldots, V_q$, called *vertex clusters*; $1 \le \mu \le \sqrt{B}$ is a parameter specified later. For each vertex cluster V_i, the adjacency lists of the vertices in V_i are concatenated to form an *edge cluster* E_i. The edges in each edge cluster E_i are stored consecutively on disk.

The shortest-path computation now proceeds as in KS-SSSP, except that a *hot pool* $\mathcal{H}$ acts as an intermediary between the priority queue and the adjacency lists. When a vertex x is retrieved from Q, it is only *released*, which means that the hot pool $\mathcal{H}$ is instructed to visit x. $\mathcal{H}$ may delay visiting x, but not long enough to compute incorrect distances, as formalized by the following property:

(SP) If vertex y is visited before vertex x, then $D(y) \le D(x) + \mathrm{dist}_G(x, y)/2$.

This implies in particular that the vertices along any shortest path are visited by increasing distances, which immediately implies the correctness of the algorithm.

The hot pool $\mathcal{H}$ is a buffer space storing adjacency lists. When a vertex x needs to be visited and $E(x)$ is in $\mathcal{H}$, the edges in $E(x)$ are relaxed. If $E(x)$ is not in $\mathcal{H}$, then *the entire edge cluster containing* $E(x)$ is loaded into $\mathcal{H}$ before x is visited. This ensures that only $\mathcal{O}(n/\mu + m/B)$ I/Os are performed to load edges into the hot pool, because every edge cluster is loaded only once. The difficult part is looking for adjacency lists in $\mathcal{H}$ efficiently, which can be done in amortized $\mathcal{O}(\mu \log W/B)$ I/Os per edge, provided that the cluster partition has certain properties, discussed in the next section. A partition with these properties can be computed in $\mathcal{O}(MST(n, m) + (n/B)\log W)$ I/Os. By using a priority queue that exploits the bounded range of the edge lengths to support Update, Delete, and DeleteMin operations in amortized $\mathcal{O}((1/B)\log W)$ I/Os, the complexity of

the algorithm thus becomes $\mathcal{O}(n/\mu + (m\mu \log W)/B + MST(n,m))$ I/Os, which is $\mathcal{O}(\sqrt{nm \log W/B} + MST(n,m))$ if $\mu = \sqrt{nB/(m \log W)}$ is chosen.

3 A Length-Independent MZ-SSSP

This section presents a new implementation of MZ-SSSP, called MZ-SSSP*. The cost of MZ-SSSP* is $\mathcal{O}((n/\mu) \log n + m(\mu d + \log n)/B)$ I/Os, where d is a parameter of the used cluster partition, called the *depth* of the partition. Section 5 will be concerned with ensuring that $d = \mathcal{O}(\log n)$, which, after choosing $\mu = \sqrt{nB/m}$, leads to the desired complexity of $\mathcal{O}(\sqrt{nm/B} \log n)$ I/Os, plus the cost for computing the partition, which will be $\mathcal{O}(MST(n,m))$ I/Os.

3.1 μ-Partitions

The efficient implementation of the hot pool in MZ-SSSP* requires that the used cluster partition has a number of properties. For an edge $e \in G$, the *category* of e is the integer c such that $2^{c-1} \leq \ell(e) < 2^c$. A *c-component* of G is a maximal connected subgraph of G all of whose edges have category c or less. A *category component* is a c-component, for some c. A *c-cluster* is a vertex cluster V_i that is contained in a c-component, but not in a $(c-1)$-component. The corresponding edge cluster E_i is also referred to as a c-cluster. The *diameter* of a vertex set V' is equal to $\max\{\text{dist}_G(x,y) \mid x, y \in V'\}$. Now the partition required by MZ-SSSP* is a *μ-partition* of G, which is a partition of V into $q = \mathcal{O}(n/\mu)$ vertex clusters $V_1, \ldots, V_q$ with the following properties:

(C1) Every cluster V_i contains at most μ vertices,
(C2) Every c-cluster V_i has diameter at most $\mu 2^c$,
(C3) No $(c-1)$-component contributes vertices to two c-clusters, and
(C4) Every c-component contributing a vertex to a c'-cluster with $c' > c$ has diameter at most $2^c \mu$.

The *depth* of a cluster is the difference between the category of the cluster and the category of the shortest edge with exactly one endpoint in the cluster. The depth of the partition is the maximal depth of its clusters. A *(μ, d)-partition* is a μ-partition of depth d. Note that $d \leq \log W$.

Even though every edge with exactly one endpoint in a c-cluster of a (μ, d)-partition has category at least $c - d$, edges between vertices in the same c-cluster may be arbitrarily short. A *mini-cluster* is a $(c-d)$-component contained in a c-cluster. (Note that, in a (μ, d)-partition, any $(c-d)$-component is either completely contained in or disjoint from a given c-cluster.) Mini-clusters have to be treated specially, in order to ensure correctness of the algorithm.

The hot pool also requires information about which vertices of which cluster are contained in which category component. This information is provided by a *cluster tree* T_i associated with each cluster V_i. To define these cluster trees, the *component tree* T_c of G needs to be defined first: A c-component is *maximal* if it is properly contained in a $(c+1)$-component or it is equal to G; all vertices of G are maximal 0-components. The vertex set of T_c consists of all maximal

category components. Component C is the parent of component C' if $C' \subset C$ and $C \subseteq C''$ for all $C'' \supset C'$. Now the cluster tree T_i of a c-cluster V_i is the subtree of T_c containing all nodes of category c or less that are ancestors of vertices in V_i (which are leaves of T_c).

3.2 Shortest Paths

The shortest-path computation proceeds in iterations; each iteration releases a vertex x from the priority queue Q and inserts a Visit$(x, d(x))$ signal into the hot pool to induce the relaxation of all edges incident to x. Before releasing x from Q, a Scan operation is invoked on the hot pool to ensure that all edges that need to be relaxed before releasing x are relaxed. This operation is described in the next section, which discusses the implementation of the hot pool.

Our algorithm uses the same priority queue as KS-SSSP. It also requires a distance repository REP, which stores the tentative distances of all vertices in G, maintained using Update$(x, d(x))$ operations, and allows the retrieval of the tentative distances of all nodes in a cluster tree T_i using a ClusterQuery(T_i) operation. Every edge relaxation in our algorithm performs an update on the priority queue *and* on the repository.

The repository can be implemented as an augmented buffered repository tree (BRT) [4], which supports Update$(x, d(x))$ operations in amortized $\mathcal{O}(\log n/B)$ I/Os and ClusterQuery(T_i) operations in amortized $\mathcal{O}((1 + r_i) \log n + |T_i|/B)$ I/Os, where r_i is the number of cluster tree roots that are contained in or adjacent to T_i. It is easy to prove that $\sum_{i=1}^{q} r_i = \mathcal{O}(n/\mu)$. Details appear in the full paper.

3.3 Hot Pool

The hot pool consists of a hierarchy of $r = \lceil \log W \rceil$ *edge buffers* $\mathrm{EB}_1, \ldots, \mathrm{EB}_r$, a hierarchy of r *tree buffers* $\mathrm{TB}_1, \ldots, \mathrm{TB}_r$, and a hierarchy of r *signal buffers* $\mathrm{SB}_1, \ldots, \mathrm{SB}_r$. Each of these hierarchies is implemented as a single stack with markers indicating the boundaries between consecutive buffers. Buffers EB_i, TB_i, and SB_i form *level* i of the hot pool.

The edge buffers hold edges that have been loaded into the hot pool. Edge buffer EB_{i+1} is inspected by Scan operations about half as often as EB_i. If an edge xy is stored in EB_i, then TB_i stores all ancestors of x in T_c that have category at most i. The signal buffers store signals that are used to trigger edge relaxations and movements of edges between different edge buffers. The purpose of moving edges between different edge buffers is to initially store edges in buffers that are inspected infrequently and later, when the time of their relaxation approaches, move them to buffers that are inspected more frequently, in order to avoid delaying their relaxation for too long. The inspection of edge buffers is controlled by *due times* $t_1 \le t_2 \le \cdots \le t_{r+1} = +\infty$ associated with these buffers. These due times satisfy the following condition:

(DT) For $1 \le i < r$, $t_{i+1} = t_i$ or $2^{i-3} \le t_{i+1} - t_i \le 2^{i-2}$.

The due times are initialized as $t_i = \min\{w(x) \mid x \in G\} + 2^{i-2}$, for $1 \le i \le r$. Due time t_i indicates that EB_i has to be inspected for edges to be relaxed or

moved to lower buffers before the first vertex with tentative distance $d(x) \geq t_i$ is released from Q. The hot pool maintains the following invariant:

(HP) After loading a c-edge cluster E_j into the hot pool, an edge $xy \in E_j$ is stored in the lowest edge buffer EB_i such that $i \geq c - d$ and the i-component C containing vertex x satisfies $d(C) < t_{i+1}$.

The tree buffers are used to check this condition. In particular, component C is stored as a node of T_j in the tree buffer TB_i, and this copy of C in TB_i always stores the correct value of $d(C) = \min\{d(x) \mid x \in C\}$.[3] To achieve this, an $\mathrm{Update}(y, d(x) + \ell(xy))$ signal is inserted into an appropriate signal buffer whenever an edge xy is relaxed; this signal updates the tentative distance of every ancestor of y in T_c that is stored in a tree buffer to which this signal is applied.

As discussed in the previous subsection, the shortest-path algorithm inserts a $\mathrm{Visit}(x, d(x))$ signal into the hot pool to trigger the relaxation of edges incident to x. This signal is inserted into SB_{c_x}, where $c_x = c - d$ if x is contained in a c-cluster. From there, it travels only up to level $c + O(\log n)$ and is then discarded. In order to achieve this insertion into SB_{c_x} without performing a random access, the signal *is sent to level* c_x by inserting it, with priority c_x, into a priority queue SQ^+. Priority queue SQ^+ is used to send signals to higher levels. Another priority queue SQ^- is used to send signals to lower levels.

Scanning the hot pool: The Scan operation scans a prefix $\mathrm{EB}_1, \ldots, \mathrm{EB}_j$ of edge buffers for edges that need to be relaxed or moved to other levels. Let f be the minimum priority of the vertices in Q. Since f is the priority of the next vertex to be released from Q, edge buffers $\mathrm{EB}_1, \ldots, \mathrm{EB}_j$ such that $t_1 \leq \cdots \leq t_j \leq f < t_{j+1}$ need to be scanned. The scanning of an edge buffer EB_i may decrease f. Then the *updated* value of f is used to decide whether to include EB_{i+1} in the scan.

The Scan operation can be divided into two phases: The *up-phase* inspects edge buffers $\mathrm{EB}_1, \ldots, \mathrm{EB}_j$, relaxes edges, and moves edges whose relaxation is not imminent to higher levels. The *down-phase* inspects $\mathrm{EB}_1, \ldots, \mathrm{EB}_j$ in reverse order, assigns new due times to $\mathrm{EB}_1, \ldots, \mathrm{EB}_j$, and moves edges to lower levels if the maintenance of property (HP) requires it. These two phases perform the following operations on each inspected level i:

Up-phase

- Retrieve all signals sent to level i from SQ^+ and insert them into SB_i.
- For every $\mathrm{Visit}(x, d(x))$ signal in SB_i such that $x \notin \mathrm{TB}_i$, load the edge cluster E_h containing $E(x)$ into EB_i; load the corresponding cluster tree T_h into TB_i and retrieve the tentative distances of all nodes in T_h from REP. E_h is loaded only once, even if more than one vertex in V_h is to be visited.
- For every cluster tree node C in TB_i and every $\mathrm{Update}(x, d)$ signal in SB_i such that $x \in C$, replace $d(C)$ with $\min(d(C), d)$.
- For every $\mathrm{Visit}(x, d(x))$ signal in SB_i, process the mini-cluster containing x. The details are explained below. For every category-c edge xy with $c \geq i$

[3] This is not quite correct; C stores only an upper bound $d^*(C) \geq d(C)$; but this upper bound suffices to move edges to lower buffers in time for their relaxation.

relaxed during this process, send an Update$(y, d(x) + \ell(xy))$ signal to level $\max(i+1, c - \log n - d - 1)$ (using SQ^+) and, if $c \leq i + \log n + d + 1$, to level i (using SQ^-). Such an Update signal is said to have category c.
- Move all cluster tree nodes C in TB_i to TB_{i+1} for which either the category of C is greater than i or the tentative distance of the i-component containing C is at least t_{i+1}. For every cluster tree leaf (vertex) x moved to TB_{i+1}, move $E(x)$ from EB_i to EB_{i+1}.
- Move update signals with category greater than $i - d$ to SB_{i+1}. Discard all other signals in SB_i.
- Test whether $f \geq t_{i+1}$ and, if so, continue to level $i + 1$.

Down-phase
- Update the due time t_i: If $f + 2^{i-1} \geq t_{i+1}$, then $t_i = t_{i+1}$. Otherwise, let $t_i = (t_{i+1} + f)/2$. It is easy to check that this maintains Property (DT).
- Retrieve all signals sent to level i from SQ^- and insert them into SB_i. At this point, they will all be Update signals sent during scans of higher levels. As in the up-phase, apply these signals to the nodes stored in TB_i.
- Move all cluster tree nodes C in TB_i to TB_{i-1} for which the category of C is less than i and the $(i-1)$-component containing C has tentative distance less than t_i. Discard all cluster tree nodes of category i. For every cluster tree leaf x moved to TB_{i-1}, move $E(x)$ from EB_i to EB_{i-1}.
- Move all signals of category less than $i + \log n + d + 1$ to SB_{i-1}. Discard all other signals in SB_i.

To implement these different steps efficiently, the nodes in T_c are numbered in preorder. The algorithm then keeps the cluster tree nodes in TB_i sorted by their preorder numbers, the signals in SB_i sorted by the preorder numbers of the vertices they affect, and the edges in EB_i sorted by the preorder numbers of their first endpoints. It is easy to show then that both phases can be implemented by scanning the involved buffers a constant number of times, except that the signals retrieved from SQ^+ and SQ^- have to be sorted before merging them into SB_i.

We show in the full paper that due times of empty levels can be represented implicitly using the due times of the two closest non-empty levels. This avoids spending I/Os on accessing due times of empty levels. Accesses to due times of non-empty levels can be charged to accesses to elements in these levels.

Processing mini-clusters: The processing of a mini-cluster C involves visiting all vertices in C that have Visit$(x, d(x))$ signals in SB_i. Since the vertices in the mini-cluster are connected by potentially very short edges, it may also be necessary to immediately visit other vertices in the same mini-cluster. In particular, starting with their current tentative distances, we apply a bounded version of Dijkstra's algorithm to the mini-cluster. This can be done in internal memory because the mini-cluster has at most $\mu \leq \sqrt{B}$ vertices and, thus, at most B edges. When Dijkstra's algorithm is about to visit a vertex x, the vertex is visited if $d(x) \leq t_i$. Otherwise, the algorithm terminates. Once Dijkstra's algorithm terminates, the tentative distances of all vertices in the mini-cluster that have not been visited are updated, that is, for each such vertex, Update$(x, d(x))$ operations are performed on Q and REP, and $d(x)$ is updated in TB_i. For every visited

vertex x and every category-c edge xy in $E(x)$ with $c \geq i$, Update$(y, d(x)+\ell(xy))$ signals are sent to the levels specified in the discussion of the up-phase. Finally, a Delete(x) operation is performed on Q for every visited vertex x. This is necessary to ensure that x is not visited again because, during the processing of the mini-cluster, vertices not yet released from Q may be visited.

3.4 Analysis

The lengthy and technical correctness proof of our algorithm is omitted from this extended abstract due to lack of space. The main idea is to prove the following lemma, which immediately implies the algorithm's correctness.

Lemma 1. *MZ-SSSP* has property (SP).*

The key to proving this is the following lemma.

Lemma 2. *A vertex x visited during a scan of level i satisfies $t_i - 2^{i-2} \leq d(x) \leq d^*(x) \leq t_i$, where $d^*(x)$ is the tentative distance stored with x in* TB_i.

From Lem. 2, Property (SP) follows almost immediately, ignoring a few technical details: Consider a vertex y that is visited before the current scan of level i. Then one can show that this vertex satisfies $d(y) < t_i$ because otherwise, level i would have been scanned before visiting y. Thus, if x and y do not belong to the same mini-cluster, then, because the path from x to y must include a category-i edge and by Lem. 2, $d(y) \leq d(x)+\mathrm{dist}_G(x, y)/2$, which implies Property (SP). If x and y belong to the same mini-cluster, it can be shown that they are visited by increasing tentative distances, that is, $d(y) \leq d(x)$, which again implies Property (SP).

The key to the analysis of the I/O-complexity is to prove that the hot pool maintains Property (HP), which we do in the full paper. Given this, we obtain

Lemma 3. *Excluding the cost of computing the (μ, d)-partition, MZ-SSSP* performs $\mathcal{O}((n/\mu)\log n + m(\mu d + \log n)/B)$ I/Os.*

Proof sketch. Observe that the algorithm performs $\mathcal{O}(m)$ priority queue operations and Update operations on REP. All these operations have an amortized cost of $\mathcal{O}((1/B)\log n)$ I/Os, which gives a cost of $\mathcal{O}((m/B)\log n)$ I/Os for these operations. Only $\mathcal{O}(m)$ signals are sent to the different levels of the hot pool, which costs $\mathcal{O}(\mathrm{sort}(m))$ I/Os for the involved operations on SQ^+ and SQ^- and for sorting these signals before insertion into the signal buffers.

The remainder of the complexity analysis hinges on two claims: (1) Every cluster is loaded into the hot pool only once. This results in a cost of $\mathcal{O}(n/\mu + m/B)$ I/Os for reading edge clusters and cluster trees, plus $\mathcal{O}((n/\mu)\log n + n/B)$ I/Os for answering cluster queries on REP. (2) Every signal traverses at most $d + \log n + 2$ levels in the hot pool; every edge and cluster tree node traverses at most d levels in the hierarchy, remaining at each level for at most $\mathcal{O}(\mu)$ scans of this level. This implies a cost of $\mathcal{O}((m/B)(d + \log n + 2))$ I/Os for scanning the signals and $\mathcal{O}(md\mu/B)$ I/Os for scanning edges and cluster tree nodes. Summing up the different costs proves the lemma.

The number of levels traversed by each edge or signal is easily seen to be as claimed. The number of scans of a level during which an edge remains at a given level follows from properties (C2) and (C4) and the fact that t_i increases by at least 2^{i-3} every time level i is scanned, which is easy to prove. Finally, Property (HP) implies immediately that every cluster is loaded only once because an edge xy, once loaded, reaches level c_x in time for its relaxation. □

4 A Recursive Shortest-Path Algorithm

This section describes a CSSP-algorithm that uses in a sense the exact opposite of a μ-partition of low depth. Section 4.1 defines the partition required by the algorithm. Section 4.2 shows that shortest paths in the whole graph can be computed by solving nearly independent CSSP-problems on the graphs in the partition. This section proves only the correctness of the algorithm. Its complexity is analyzed in Sect. 5, where it is combined with MZ-SSSP* to obtain the final algorithm.

4.1 Barrier Decomposition

The algorithm uses a *barrier decomposition* of G, which consists of a number of multigraphs $G_0, \ldots, G_q$ and vertex sets $\emptyset = B_0, \ldots, B_q$, called *barriers*, with the following properties:

(B1) Every graph G_i represents a connected vertex-induced subgraph H_i of G; $H_0 = G$.
(B2) For $i < j$, $H_i \cap H_j = \emptyset$ or $H_j \subset H_i$. If $H_j \subset H_i$ and $H_i \subseteq H_k$ for all $H_k \supset H_j$, G_i is the *parent* of G_j (and G_j a *child* of G_i).
(B3) For all i, graph G_i is obtained from H_i by contracting each graph H_j such that G_j is a child of G_i into a single vertex $r(G_j)$, called the *representative* of G_j. For a vertex $x \in H_j$, $r(G_j)$ is considered the representative of x in G_i and denoted by r_x. For $x \in G_i$, let $r_x = x$.
(B4) For a given graph G_j with parent G_i, B_j is the set of vertices in $(V(H_i) \cup B_i) \setminus V(H_j)$ that are reachable from H_j using edges of length at most $2n\ell_{\max}(H_j)$, where $\ell_{\max}(H_j)$ is the length of the longest edge in H_j.
(B5) No set B_i contains a graph representative.

Intuitively, for every graph G_j, the set B_j forms a barrier between H_j and the rest of G in the sense that a shortest-path between two vertices in H_j cannot contain a vertex not in $V(H_j) \cup B_j$.

4.2 The Algorithm

Now assume that a Dijkstra-like CSSP algorithm $\mathcal{A}$ is given, that is, an algorithm that visits every vertex exactly once and, when it does, relaxes all edges incident to x. Assume also that algorithm $\mathcal{A}$ has property (SP). Given a barrier decomposition of G, the CSSP problem in G can then be solved using the following recursive algorithm. The algorithm requires the use of the distance repository REP from Sect. 3, augmented to support a GraphQuery(G_i) operation, which

returns the tentative distances of all vertices in G_i; for a graph representative $x = r(G_j)$, let $d(x) = \min\{d(y) \mid y \in H_j\}$. In the full paper, we show how to perform such an operation in amortized $\mathcal{O}((1 + c_i) \log n + |V(G_i)|/B)$ I/Os, where c_i is the number of children of G_i.

ShortestPaths(G_i)**:** Run a modified version of algorithm $\mathcal{A}$ on the graph $G_i \cup B_i$ obtained from $G[V(H_i) \cup B_i]$ by contracting each graph H_j such that G_j is a child of G_i into a single vertex $r(G_j)$. The modifications are as follows:

- Terminate $\mathcal{A}$ as soon as all vertices in G_i have been visited. In particular, it is not necessary to visit all vertices in B_i.
- When $\mathcal{A}$ visits a vertex x that is not a graph representative, relax all its incident edges. In particular, for each such edge xy, where r_y may or may not be a graph representative, replace $d(y)$ with $\min(d(y), d(x) + \ell(xy))$ in REP and $d(r_y)$ with $\min(d(r_y), d(x) + \ell(xy))$ in $\mathcal{A}$'s data structures.
- When $\mathcal{A}$ visits a graph representative $r(G_j)$:
  - Recursively invoke ShortestPaths(G_j) with the weights of all vertices in $V(G_j) \cup B_j$ initialized to their current tentative distances. (These distances are retrieved from REP.)
  - If the recursive call visits vertices in B_j, reflect this in the data structures of the current invocation to ensure that these vertices are not visited again. (E.g., if $\mathcal{A}$ is Dijkstra's algorithm or MZ-SSSP*, remove these vertices from the priority queue.)
  - If the recursive call updates the tentative distances of vertices in B_j, reflect this in the data structures of the current invocation. (E.g., if $\mathcal{A}$ is MZ-SSSP*, update their priorities in the priority queue and send corresponding Update signals to the hot pool.)
  - Relax all edges with exactly one endpoint in H_j, that is, for each edge xy such that $x \in H_j$ and $y \in H_i \setminus H_j$, replace $d(r_y)$ with $\min(d(r_y), d(x) + \ell(xy))$.

The initial invocation is on graph G_0, which ensures that all vertices in G are visited. The following lemma shows that this solves the CSSP problem.

Lemma 4. *For every vertex $x \in H_i \cup B_i$ visited by ShortestPaths(G_i), $d(x) = D(x)$ at the time when x is visited.*

Proof. The proof is by induction on the number of descendants of G_i. If there is none, the algorithm behaves like $\mathcal{A}$ and the claim follows because, by (SP), all vertices on $\pi(x)$ are visited in order.

So assume that G_i has at least one child G_j, that there exists a vertex $x \in H_i \cup B_i$ such that $d(x) > D(x)$ when x is visited, and that every vertex z preceding x on $\pi(x)$ satisfies $d(z) = D(z)$ when it is visited. First assume that x is not visited in a recursive call Shortest-Path(G_j), where G_j is a child of G_i. Let y be x's predecessor on $\pi(x)$, and let r_y be its representative in G_i. r_y must be visited after x because otherwise $d(x) = D(x)$ when x is visited. Hence, by (SP), $D(y) \geq D(r_y) \geq D(x) - \text{dist}_{G_i \cup B_i}(r_y, x)/2 \geq D(x) - \text{dist}_G(y, x)/2$, a contradiction because $y \in \pi(x)$.

Now assume that x is visited during a recursive call ShortestPaths(G_j). Then the claim follows by induction if we can prove that $D_{H_j \cup B_j}(x) = D(x)$. If $\pi(x) \subseteq H_j \cup B_j$, this is trivial. So assume that $\pi(x)$ contains at least one vertex outside $H_j \cup B_j$. Let z be the last such vertex on $\pi(x)$, and let y be its successor on $\pi(x)$, which is in $H_j \cup B_j$. We need to prove that $w(y) = D(y)$.

Assume the contrary. Then $r = r(G_j)$ must be visited before r_z, that is, by (SP), $D(r) \leq D(r_z) + \text{dist}_{G_i \cup B_i}(r_z, r)/2$. However, $D(u) < D(r) + n \cdot \ell_{\max}(H_j) \leq D(r) + \text{dist}_{G_i \cup B_i}(r_z, r)/2$, for all $u \in H_j$, because $z \notin H_j \cup B_j$. Hence, $D(u) < D(r_z) + \text{dist}_{G_i \cup B_i}(r_z, r) \leq D(z) + \text{dist}_G(z, u)$. Thus, z cannot belong to $\pi(u)$, for any $u \in H_j$, and $x \notin H_j$. Then, however, x is visited only if $D_{H_j \cup B_j}(x) \leq D_{H_j \cup B_j}(u)$, for some $u \in H_j$. Since $D(x) \leq D_{H_j \cup B_j}(x)$ and $D_{H_i \cup B_i}(u) = D(u)$, this implies again that $z \notin \pi(x)$, a contradiction. □

5 The Final Algorithm

Our final algorithm is based on the recursive framework of Sect. 4 and uses MZ-SSSP* or, on small graphs, Dijkstra's algorithm to compute shortest paths on the different graphs in the barrier decomposition. To achieve the claimed I/O-complexity, the following properties of the barrier decomposition are required:

(P1) The barrier decomposition consists of $\mathcal{O}(n/\mu)$ multigraphs $G_0, \ldots, G_q$.
(P2) Each graph G_i has at most $\sqrt{B}$ vertices or is equipped with a $(\mu, \log n + 2)$-partition. In the former case, it is called *atomic*; in the latter, *compound*.
(P3) If the parent G_i of G_j is atomic, then G_j is G_i's only child. If the parent G_i of G_j is compound, then B_j is a subset of a vertex cluster of G_i, and this vertex cluster contains only one graph representative, namely $r(G_j)$. This implies in particular that $|B_j| \leq \mu \leq \sqrt{B}$, for all j.

In the full paper, we prove the following lemma.

Lemma 5. *It takes $\mathcal{O}(MST(n, m))$ I/Os to compute a barrier decomposition of an undirected graph G that has properties (P1)–(P3).*

In a nutshell, such a decomposition can be obtained as follows: In [9], a procedure is described that computes a μ-partition of a graph in $\mathcal{O}(MST(n, m) + (n/B) \log W)$ I/Os, by computing a minimum spanning tree T and then computing c-clusters iteratively using $\log W$ scans of an Euler tour of T. Using an algorithm from [3], the component tree T_c can be computed from T in $\mathcal{O}(\text{sort}(n))$ I/Os; the $\log W$ scans of the Euler tour can then be simulated in $\mathcal{O}(\text{sort}(n))$ I/Os using one traversal of T_c. Once this μ-partition is given, two more traversals of T_c are needed. The first one refines the partition so that all clusters of depth greater than $\log n + 2$ have a particularly simple structure. The second one splits each of these deep clusters into three parts: a top, middle, and bottom part, which correspond to top, middle, and bottom parts of its cluster tree. The top and bottom parts define clusters in $(\mu, \log n + 2)$-partitions of two graphs G_i and G_k in the barrier decomposition. The middle part G_j defines an atomic graph in the barrier decomposition that is a child of G_i and the parent of G_k.

By Lem. 5, it takes $\mathcal{O}(MST(n,m))$ I/Os to compute the desired decomposition of the graph. Using MZ-SSSP* to solve CSSP in a compound graph G_i in the computed barrier decomposition takes $\mathcal{O}(((n_i + |B_i|)/\mu)\log n + (m_i\mu \log n)/B)$ I/Os, where n_i is the number of vertices in G_i and m_i is the number of edges in G_i. If G_i is atomic, Dijkstra's algorithm can be used to solve CSSP in G_i, which incurs $\mathcal{O}(1 + m_i/B)$ I/Os because G_i fits in memory.

It is easy to see that $\sum_{i=1}^{q}(n_i+|B_i|) = \mathcal{O}(n)$ and $\sum_{i=1}^{q} m_i = \mathcal{O}(m)$. Hence, the cost of all CSSP-computations on graphs G_i is $\mathcal{O}((n/\mu)\log n + (m\mu \log n)/B)$.

The cost of all repository operations can be bounded as follows: The algorithm performs exactly one subgraph query per graph G_i and at most two cluster queries per cluster: one when the cluster is loaded into the hot pool and another one when the graph representative $r(G_j)$ in the cluster is visited. Moreover, it is easy to show that the sum of the r_j and c_i is $\mathcal{O}(n/\mu)$, so that the cost of all queries on the repository is $\mathcal{O}((n/\mu)\log n + n/B) = \mathcal{O}((n/\mu)\log n)$ I/Os. Since the algorithm performs only $\mathcal{O}(m)$ edge relaxations, the cost of all Update operations on the repository is $\mathcal{O}((m/B)\log n)$ I/Os.

Summing the costs of all parts of the algorithm yields an I/O-complexity of $\mathcal{O}((n/\mu)\log n + (m\mu \log n)/B + MST(n,m))$, which is $\mathcal{O}(\sqrt{nm/B}\log n + MST(n,m))$ for $\mu = \sqrt{nB/m}$. This proves Thm. 1.

References

1. A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Comm. ACM*, pp. 1116–1127, 1988.
2. L. Arge, G. S. Brodal, and L. Toma. On external-memory MST, SSSP and multi-way planar graph separation. *J. Alg.*, 53(2):186–206, 2004.
3. L. Arge, L. Toma, and N. Zeh. I/O-efficient algorithms for planar digraphs. In *Proc. 15th SPAA*, pp. 85–93, 2003.
4. A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. On external memory graph traversal. In *Proc. 11th SODA*, pp. 859–860, 2000.
5. Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. 6th SODA*, pp. 139–149, 1995.
6. E. W. Dijkstra. A note on two problems in connection with graphs. *Num. Math.*, 1:269–271, 1959.
7. V. Kumar and E. J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proc. 8th SPDP*, pp. 169–176, 1996.
8. K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In *Proc. 10th ESA*, LNCS 2461, pp. 723–735. Springer-Verlag, 2002.
9. U. Meyer and N. Zeh. I/O-efficient undirected shortest paths. In *Proc. 11th ESA*, LNCS 2832, pp. 434–445. Springer-Verlag, 2003.
10. U. Meyer and N. Zeh. I/O-efficient undirected shortest paths with unbounded weights. Tech. Report CS-2006-04, Faculty of Comp. Sci., Dalhousie Univ., 2006.
11. S. Pettie and V. Ramachandran. Computing shortest paths with comparisons and additions. In *Proc. 13th SODA*, pp. 267–276, 2002.
12. M. Thorup. Undirected single source shortest paths with positive integer weights in linear time. *J. ACM*, 46:362–394, 1999.
13. M. Thorup. Floats, integers, and single source shortest paths. *J. Alg.*, 35:189–201, 2000.

Stochastic Shortest Paths Via Quasi-convex Maximization

Evdokia Nikolova*
Jonathan A. Kelner, Matthew Brand, and Michael Mitzenmacher

MIT CSAIL, MERL, Harvard University / Cambridge MA

Abstract. We consider the problem of finding shortest paths in a graph with independent randomly distributed edge lengths. Our goal is to maximize the probability that the path length does not exceed a given threshold value (deadline). We give a surprising exact $n^{\Theta(\log n)}$ algorithm for the case of normally distributed edge lengths, which is based on quasi-convex maximization. We then prove average and smoothed polynomial bounds for this algorithm, which also translate to average and smoothed bounds for the parametric shortest path problem, and extend to a more general non-convex optimization setting. We also consider a number other edge length distributions, giving a range of exact and approximation schemes.

1 Introduction

Finding shortest paths between a given source and destination is a classic and fundamental problem in theoretical computer science which has influenced a wide array of other fields. It is less clear what a *stochastic* shortest path would mean, when the edge lengths are random with given distributions. Is it the shortest path on average, or the path minimizing a combination of mean and variance, or minimizing some other criterion? Is it found adaptively or non-adaptively? A variety of problem variants have appeared in the literature, most minimizing the expected length of a path, or a combination of expected length and expected cost such as bicriterion problems [15], [19]. Adaptive formulations have prevailed, perhaps because a non-adaptive minimization of the expected path length trivially reduces to the deterministic shortest path problem.

Few researchers have considered optimizing a non-linear function of the (random) path length. Some notable work includes that of Loui [12] who seeks the path maximizing an expected utility of the path length for a class of monotone decreasing utility functions. Fan *et al.* [6] present an adaptive heuristic for paths that maximize the probability of arriving on time. Formulations of this type with nonlinear objective, though perhaps most useful in practice, have been sparse, because different sources of hardness arise from many levels: combinatorial, distributional, analytic, functional, to list a few. For example, in the absence of randomness, the combinatorial nature of the problem may be hard to approximate for certain objective functions (*e.g.,* longest path [9]).

* nikolova@mit.edu. MIT CSAIL, Ongoing doctoral thesis work. Part of this work was done while the author was at Mitsubishi Electric Research Labs. Supported in part by NSF grants ANI-0225660 and ITR-0219018.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 552–563, 2006.

In the absence of graph structure, the objective function in itself may be difficult to optimize: we can solve efficiently linear programming but not even quadratic or more generally non-convex programming. The distributions may be hard to work with: calculating values of the cumulative distribution function of the sum of n Bernoulli random variables is #P-hard as it corresponds to counting knapsack solutions [11]. Computing the expectation $\mathbf{E}[u(X)] = \int u(x)f(x)dx$ of the non-linear utility function $u(.)$ of the random path length X with probability density function $f(.)$ may not even have a closed form, thus making the standard notion of computational hardness inapplicable. Superimposing these sources of difficulty may ultimately lead to a problem that has no hope of even being categorized as to what level of hardness it has—partly because we do not understand to what extent each source contributes to the overall complexity.

We thus focus on a stochastic shortest paths model which can effectively factor the sources of difficulty above and at the same time has an innovative solution drawing from a variety of areas. Inspired by recent formulations of the stochastic knapsack and other classic problems turned stochastic [5], [7] our goal is to maximize the probability that the path length would not exceed some threshold value. This is a natural formulation which is also very practical: For example, this is our objective when we are going to the airport and want to pick the path that would maximize our probability of arriving on time for our flight. We consider a pre-planning (nonadaptive) scenario and note that it can easily be converted to an adaptive one by rerunning our algorithms on the fly with updated information.

Apart from the inherent practicality of the problem, it reveals a deeper theoretic structure intertwining areas such as nonconvex programming, the geometry of path polytopes and combinatorial optimization. As a preview to some of the open questions, we give an exact $n^{\Theta(\log n)}$ algorithm for our main model with edge lengths drawn from normal distributions. It is unknown whether a polynomial exact algorithm exists or whether this problem is complete for the corresponding complexity class LogNP [21]. Our algorithm also reveals a somewhat unexpected connection between Kelner & Spielman's recent techniques for linear programming [10] and the much more general field of nonconvex optimization. We extend their techniques to get polynomial-time average and smoothed complexity for our superpolynomial algorithms. We stress that these smoothed results are stronger than previous smoothed results in that they do not perturb the feasible set (the path polytope), but just the objective function (the plane on which the polytope is projected). Or, in the terms of the stochastic shortest paths terminology, only the edge means and variances and *not* the solution paths themselves, are slightly perturbed. As an added benefit, we reveal a connection between the stochastic and parametric shortest path problems, which implies new average and smoothed results for the parametric shortest path problem as well. Our results can also generalize to a wide class of non-convex optimization problems, known as low-rank quasiconcave minimization [18].

1.1 Our Results

We define a model for the stochastic shortest path problem in which the edge lengths are independent random variables drawn from known distributions. The optimal path

maximizes the probability that the path length does not exceed a given threshold (deadline). This objective arises naturally in practice where a user wants to maximize the probability of arriving on time to a destination. In an effort to decouple the complexity inherent in this objective from the distributional and analytic complexity of the problem, our first model draws the edges from normal distributions. We show that for a large range of deadlines our problem entails the maximization of a quasi-convex function over the path polytope. Due to the particular form of our quasi-convex objective, the optimal path is attained at an extreme point of the dominant of the projection (shadow) of the path polytope onto a two-dimensional plane. We thus give an exact algorithm for finding the optimal path by walking along extreme points of the shadow dominant. We then establish an equivalence between the shadow dominant and the optimal cost envelope of the parametric shortest path problem. Consequently, this proves that our algorithm has a worst case running time $n^{\Theta(\log n)}$. We give a pseudopolynomial algorithm for the remaining range of deadlines.

In the following section we extend the techniques from Kelner & Spielman [10] to prove linear average and smoothed complexity of the shadow of the path polytope and consequently polynomial running time of our algorithm. These results also imply new polynomial average and smoothed bounds on the complexity of the parametric shortest path problem and hold for a wider class of non-convex optimization problems than the specific stochastic shortest path objective.

Finally we extend our model to distributions other than the normal. For edge lengths coming from a Poisson or a gamma distribution with a fixed second parameter, or more generally distributions which are additive and satisfy stochastic dominance, we show that the problem easily reduces to the deterministic shortest path problem. For the case of exponential and Bernoulli random variables, we give polynomial (PTAS) and quasi-polynomial (QPTAS) approximation schemes respectively based on a discretization of the state space of the random edge lengths.

1.2 Related Work

The majority of the related literature on stochastic shortest paths focuses on adaptive algorithms, which compute the next best hop based on information about realized edge lengths so far [2], [22], [3], [20], [6], [13]. Most of the adaptive formulations focus on minimizing expected path length; few consider minimizing a non-linear function of the length and settle for heuristic algorithms [6].

The most closely related nonadaptive formulation to our model is that of Loui [12]. Loui considers a general utility function of path length which is monotone and nondecreasing, and proves that the expected utility becomes separable into the edge lengths only when the utility function is linear or exponential. In that case the path that maximizes expected utility can be found via traditional shortest path algorithms. For general utility functions he gives an algorithm based on an enumeration of paths, with a very large running time $O(n^n)$. In a consequent paper, Mirchandani and Soroush give exponential algorithms and heuristics for quadratic utility functions [14]. For non-monotone utility functions Nikolova, Brand and Karger [17] give hardness results and pseudopolynomial algorithms. For a separate model on bicriteria shortest paths with monotone objective, Ackerman *et al.* [1] give different average and smoothed analyses.

2 Problem Definitions and Quasi-convex Maximization

2.1 Stochastic Shortest Path Definition

Consider a graph $G = (V, E)$, with $|V| = n$ nodes and $|E| = m$ edges. We are given a source S and destination T. Each edge i has an independent random variable length (travel time) X_i. We have a deadline in time t, and we would like to find an ST-path which maximizes the probability that we reach the destination within time t. Thus, we would like to solve

$$\max_{\pi} \Pr\Big(\sum_{i\in\pi} X_i \le t\Big) \quad \text{for paths } \pi \text{ between the source and destination.} \tag{1}$$

In the following sections, we see that different distributional assumptions for the edge lengths lead to problem complexity and algorithms of very different nature.

2.2 Parametric Shortest Path Definition

Consider a graph G with distinguished source S and destination T. Each edge i has a parameter dependent length $u_i + \lambda w_i$, where u_i, w_i are nonnegative constants. and $\lambda \in [0, \infty)$. The parametric shortest paths problem looks for the parameter values (breakpoints) $\lambda \in (0, \infty)$ at which the shortest path changes. Carstensen [4] proved that the number of breakpoints is at least $n^{\Omega(\log n)}$ in the worst case, and one can easily show a matching upper bound for general graphs (A more involved proof on the upper bound is also available in Carstensen [4]).

In the next section we will establish a connection between the stochastic shortest paths with normal distributions and the parametric shortest paths problem, which will enable us to apply our average and smoothed results for the former to the parametric shortest path setting as well.

2.3 Quasi-convex Maximization

In this section we briefly define convex functions and their generalization to quasi-convex functions and state the main property of their global maxima.

Let C be a convex set.

Definition 1. *A function $f : C \to (-\infty, \infty]$ is* convex *if for all $x, y \in C$ and $\alpha \in [0, 1]$,*

$$f(\alpha x + (1-\alpha)y) \le \alpha f(x) + (1-\alpha) f(y).$$

A function $f : C \to (-\infty, \infty]$ is quasi-convex *if all its lower level sets $L_\gamma = \{x \mid x \in C,\ f(x) \le \gamma\}$ are convex.*

Informally, quasi-convex functions have a convex cross-section at any height (level).

Definition 2. *We say that x is an* extreme *point of the set C if it cannot be represented as a convex combination of two other points in the set C,*

$$x = \alpha y + (1-\alpha) z \text{ for } y, z \in C,\ \alpha \in (0,1) \qquad \Rightarrow y = z = x.$$

The following important property of quasi-convex maximization seems to be attributed to folklore. A statement of the theorem without proof appears in the *Introduction to Global Optimization* [8]; our proof is deferred to the full version of this paper.

Theorem 1. *Let $C \subset \mathcal{R}^m$ be a compact convex set. A quasi-convex function $f : C \to \mathcal{R}$ that attains a maximum over C, attains the maximum at some extreme point of C.*

We will need a few more definitions. The *shadow* of a convex set in $\mathcal{R}^m$ onto a two-dimensional subspace is the orthogonal projection of the set onto the subspace. The *dominant* of a set C in $\mathcal{R}^m$ is defined as the set of all points that are greater than a point in C, $\{x \in \mathcal{R}^m \mid x \geq y \text{ for some } y \in C\}$.

3 Stochastic Shortest Paths with Normal Distributions

In this section we apply quasi-convex maximization to a graph with normally distributed edge lengths, in which we have to select the most certain route to reach a destination by a given time.

Assume each edge i has independent normally distributed length $X_i \sim N(\mu_i, \sigma_i^2)$. Our problem is to

$$\max_{\pi} \Pr(\sum_{i \in \pi} X_i \leq t) \quad \text{for paths } \pi \text{ between the source and destination.} \tag{2}$$

For any path π, this probability can be computed by

$$\Pr\Big(\sum_{i\in\pi} X_i \leq t\Big) = \Pr\Big(\frac{\sum X_i - \sum \mu_i}{\sqrt{\sum \sigma_i^2}} \leq \frac{t - \sum \mu_i}{\sqrt{\sum \sigma_i^2}}\Big) = \Phi\Big(\frac{t - \sum \mu_i}{\sqrt{\sum \sigma_i^2}}\Big),$$

where $\Phi(.)$ is the cumulative distribution function of the standard normal random variable $N(0, 1)$. Since Φ is monotone increasing, the problem is equivalent to finding the ST-path which maximizes its argument,

$$\max_{\pi} \frac{t - \sum_{i\in\pi} \mu_i}{\sqrt{\sum_{i\in\pi} \sigma_i^2}}. \tag{3}$$

The objective in Eq. (3) cannot be separated into edge costs and does not satisfy suboptimality so a dynamic programming approach based on substructure would fail. To better understand the properties of the objective function, we formulate it as a continuous optimization problem over the path polytope in $\mathcal{R}^m$, where m is the number of edges.

Index all edges by $1, 2, ..., m$. Represent each edge subset by its incidence vector $x \in \mathcal{R}^m$, with $x_i = 1$ if edge i is in the subset and $x_i = 0$ otherwise. All 2^m subsets of edges correspond to the vertices of the unit hypercube in $\mathcal{R}^m$. The *ST-path polytope* (or, the *path polytope* for short) is the convex hull of incidence vectors of (simple) ST-

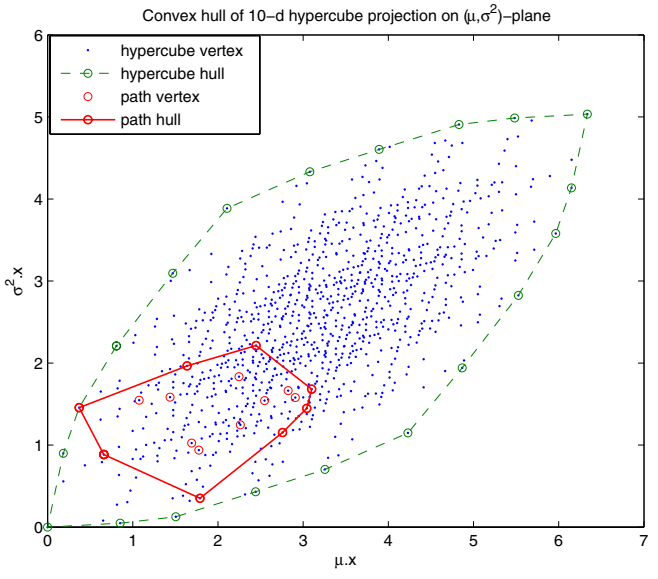


Fig. 1. Projection of the unit hypercube (representing all edge subsets) and the path polytope onto the (μ, σ^2)-plane

paths. It is a subset of the unit hypercube in $\mathcal{R}^m$, and its vertices are a subset of the vertices of the hypercube. Thus, the optimal ST-path is a solution to

$$\begin{aligned} \text{maximize} \quad & \frac{t - \mu \cdot x}{\sqrt{\sigma^2 \cdot x}} \\ \text{subject to} \quad & x \in \text{ path polytope} \\ & x \in \{0,1\}^m, \end{aligned} \tag{4}$$

where by $\{0,1\}^m$ we denote the set of $0{-}1$ vectors of length m. Projecting the path polytope onto the span of vectors $\mu = (\mu_1, ..., \mu_m)$ and $\sigma^2 = (\sigma_1^2, ..., \sigma_m^2)$ defines a convex polygon, which we call the path polytope shadow. The objective in Eq. (4) is not separable, far from linear or quadratic and not even convex. This places it in a category of mathematical programming and combinatorial optimization problems, for which there are no general efficient algorithms. Although the integer constraints are what usually causes the main difficulty, in this case it is not clear how to solve even the fractional version.

It turns out our objective has special structure which forces its maximum to lie on the boundary of the feasible set. In particular, it is quasi-convex on a subset of the path polytope and monotone in $\mu \cdot x$ and $\sigma^2 \cdot x$ on the remaining subset of the polytope. This is not automatically good news since we do not have a polynomial description of the boundary of the path polytope or even its shadow. For example computing the rightmost and uppermost vertices of the path polytope shadow corresponds to finding the longest path, in terms of the edge means and edge variances respectively. Thus the computation of the path convex hull is in general strongly NP-hard [9]. On the other

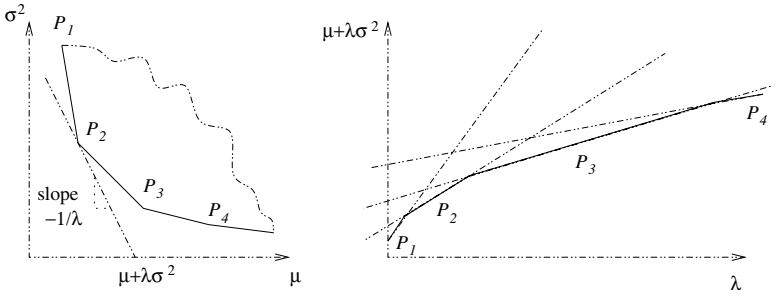


Fig. 2. Correspondence of extreme points $P_1, P_2, ...$ of the dominant of the shadow path polytope (left) and linear segments from the parametric path cost-function $g(\lambda) = \inf_x\{\mu \cdot x + \lambda(\sigma^2 \cdot x)\}$ (right)

hand, we can efficiently find the extreme points on the dominant of the shadow hull, since they optimize the linear objective $\gamma\mu + (1-\gamma)\sigma^2$ for $\gamma \in [0,1]$.

Our main theorem 2 shows that for sufficiently early departure time (eliminating the incidence of a longest path problem), our objective is quasi-convex and we can solve the stochastic shortest paths problem exactly in time $n^{\Theta(\log n)}$. We first state a lemma about the correspondence of the stochastic and parametric shortest paths problems. Its proof is deferred to the full paper version.

Lemma 1. *There is a one-to-one correspondence between the extreme points on the shadow of the path polytope dominant on the plane spanned by vectors* $u=(u_1, ..., u_m)$, $w = (w_1, ..., w_m)$ *and the breakpoints of the parametric shortest path problem with edge weights* $u_i + \lambda w_i$.

By Lemma 1, the results for the complexity of the parametric shortest paths problem [4], [16] imply equivalent bounds for the number of extreme points on the shadow dominant.

Corollary 1. *The dominant of the path polytope shadow has* $n^{\Theta(\log n)}$ *extreme points in the worst case.*

We now turn to the main result in this section.

Theorem 2. *When the deadline* t *is no less than the mean of the smallest-mean path, the solution to Eq. (4) is an extreme point of the dominant of the path polytope shadow and can be found in time* $n^{\Theta(\log n)}$.

Proof. We first consider the relaxed version of Eq. (4). Denoting $z_1 = \mu \cdot x$ and $z_2 = \sigma^2 \cdot x$, the system becomes equivalent to

$$\begin{aligned} &\text{maximize} \quad \frac{t - z_1}{\sqrt{z_2}} \\ &\text{subject to} \quad (z_1, z_2) \in \text{ path polytope shadow } S \end{aligned} \tag{5}$$

We first show that the induced objective $f(z_1, z_2) = \frac{t-z_1}{\sqrt{z_2}}$ is quasi-convex on a subset of the feasible set $\bar{S} = S \cap \{z_1 \mid z_1 < t\}$ (which is non-empty assuming there is a path

with mean less than t). Since $z_1 = \mu \cdot x < t$, the value of $f(z_1, z_2)$ on this feasible subset is positive, and must contain the maximum. Consider the level set $L_\gamma = \{z \in \mathcal{R}^2 \mid f(z) \leq \gamma\}$. This set consists of points (z_1, z_2) such that

$$\frac{t - z_1}{\sqrt{z_2}} \leq \gamma \qquad \Longleftrightarrow \qquad z_2 \geq \left(\frac{t - z_1}{\gamma}\right)^2,$$

hence for positive γ and $z_1 < t$, the level set L_γ is convex. Therefore $f(z_1, z_2)$ is quasi-convex on $\bar{S}$, which is the part of path polytope shadow to the left of $z_1 = t$. By Theorem 1, the maximum is attained at an extreme point of $\bar{S}$. Further, since $f(z_1, z_2)$ is monotone decreasing in both z_1 and z_2, the solution must be an extreme point of the dominant of the shadow, to the left of $z_1 = t$.

Now, any extreme point of the shadow is the projection of an exteme point of the original path polytope (which has integer coordinates). Hence the optimal solution of the relaxed program (5) is also a solution to the integer program (4).

Next, the extreme points of the dominant of the shadow can be found in time linear in their number, for example with a binary search type enumeration as follows. Each extreme point on the shadow dominant is the solution to a linear program

$$\begin{aligned} \min \quad & c \cdot z \\ \text{subject to} \quad & z \in \text{ shadow path polytope} \end{aligned} \tag{6}$$

for some $c = (c_1, c_2) \geq 0$. Equivalently, each extreme point corresponds to a path minimizing $c_1 z_1 + c_2 z_2$ where $z_1 = \mu \cdot x$ is the total mean of the path and z_2 is the total variance so for $c_1, c_2 \geq 0$ it can be found via any shortest path algorithm. To find all extreme points on the shadow dominant, we start with its two endpoints: the leftmost point, which corresponds to the path with smallest mean, and the bottom-most point, which is the path of smallest variance. Denote these $\pi_1 = (m_1, s_1), \pi_2 = (m_2, s_2) \in \mathcal{R}^2$, where m_i is the mean and s_i the variance of path π_i, then solve Eq. (6) with $(c_1, c_2) = (-\frac{s_2 - s_1}{m_2 - m_1}, 1)$ if $m_2 - m_1 \neq 0$, otherwise $(c_1, c_2) = (1, 0)$. The new solution is $\pi_3 = (m_3, s_3)$, a vertex between π_1 and π_2 on the shadow boundary. If different from both π_1 and π_2, we repeat the procedure for finding a vertex between π_1, π_3 and between π_3, π_2, etc. Clearly in this way we find all vertices on the shadow dominant in time linear in their number, multiplied by the time to solve the auxiliary program (6). Similar enumeration methods for extreme points are discussed in Carstensen [4].

Finally, since there are $N = n^{\Theta(\log n)}$ extreme points of the shadow dominant in the worst case by Corollary 1 and we can find each in polynomial time, the running time of the algorithm is $n^{\Theta(\log n)}$.

When the departure time is closer to the deadline, so that any shortest path has mean greater than t, our objective is no longer quasi-convex, in fact it is increasing in the variance. Since finding the simple path with highest variance is strongly NP-hard [9], we might not expect to find a good polynomial-time approximation. Settling for potentially non-simple paths, we can give a pseudopolynomial dynamic programming solution, which finds the path of smallest mean for every possible value of its variance and then selects the ST-path with optimal objective value.

Theorem 3. *For general deadline t, the solution to Eq. (4) can be found in time $O(\sigma^2 nm)$ where σ^2 is the maximum variance of an edge.*

4 Average and Smoothed Complexity

In this section we show that if the edge weight vectors $u, w \in \mathcal{R}^m$ are uniformly random unit vectors or fixed vectors which are slightly perturbed, then the expected number of extreme points on the path polytope shadow is linear and consequently our $n^{\Theta(\log n)}$ algorithm from the previous section will have a low expected polynomial running time. The techniques in this section are motivated by the recent techniques of Kelner and Spielman [10] for the polynomial simplex algorithm for linear programming.

Note that the vertices of the path polytope P are a subset of the vertices of the unit hypercube, in particular:

Fact 1. *Each edge of the polytope P has length at least* 1.

Fact 2. *The polytope P is contained in the unit hypercube, which in turn is contained in a ball with radius $\sqrt{m}/2$.*

4.1 Average Bounds

Theorem 4. *Let $u, w \in \mathcal{R}^m$ be uniformly random unit vectors and let V be their span. Then the expectation of the number of edges on the projection of P onto V is at most $2\sqrt{2}\pi m$.*

Proof. By Fact 2, the perimeter of the shadow of P onto V is bounded above by $\pi\sqrt{m}$. Next, for each edge I of the polytope P, denote by $S_I(V)$ the event that edge I appears in the shadow, and let $l(I)$ be the length of the edge in the shadow. The sum of expected edge lengths in the shadow is at most equal to the biggest possible perimeter:

$$\sum_I \mathbf{E}[l(I)] = \sum_I \mathbf{E}[l(I)|S_I(V)] \Pr[S_I(V)] \leq \pi\sqrt{m}.$$

By Lemma 2 below, $\mathbf{E}[l(I)|S_I(V)] \geq \frac{1}{2\sqrt{2m}}$. Therefore,

$$\mathbf{E}[\text{number of shadow edges}] = \sum_I \Pr[S_I(V)] \leq 2\sqrt{2}\pi m,$$

where m is the dimension of the polytope P, in our case it is the number of edges of the original graph.

Lemma 2. *For all edges I of the polytope P, $\mathbf{E}[l(I)|S_I(V)] \geq \frac{1}{2\sqrt{2m}}$.*

Proof. We first note a direct corollary from Lemma 3.8 in Kelner & Spielman [10], namely that if an edge I of the polytope appears in the shadow, it must make a small angle $\theta_I(V)$ with the projection plane V, $\Pr_V\left[\cos(\theta_I(V)) \geq \frac{1}{\sqrt{2m}} \mid S_I(V)\right] \geq \frac{1}{2}$.

Now, since any edge in the polytope P has length at least 1 (by Fact 1 above), the length of the edge in the shadow would be at least $\cos(\theta_I(V))$ and its expectation provided it appears in the shadow is

$$\mathbf{E}[l(I)|S_I(V)] \geq \frac{1}{\sqrt{2m}} \frac{1}{2}.$$

4.2 Smoothed Bounds

We now provide smoothed results for the maximization of our quasi-convex objective. In particular, we show that the expected number of extreme points (equivalently edges) on the projection of a general $0-1$ vertex polytope onto a perturbed plane is polynomial in m and $1/\rho$, the inverse of our perturbation.

We first define a ρ-perturbation of the vector u, for $\rho > 0$. Choose an angle $\theta \in [0, \pi]$ at random from an exponential distribution with mean ρ, restricted to the range $[0, \pi]$. Set the ρ-perturbation of u to be a unit vector chosen uniformly at random at an angle θ to u. The following theorem states that the expected number of edges on the polytope shadow is polynomial.

Theorem 5. *Let $u_1, u_2 \in \mathcal{R}^m$ be given vectors and let v_1 and v_2 be their respective ρ-perturbations. Denote $V = \mathrm{span}(v_1, v_2)$. The expected number of edges of the projection of P onto V is at most $4\pi\sqrt{2m}/\rho$, for $\rho < 1/\sqrt{m}$.*

The theorem follows similarly to the argument in Section 4.1 from the next lemma.

Lemma 3. *With the variables above,* $\Pr_{v_1,\, v_2}[\cos(\theta_I(V)) \leq \epsilon \mid S_I(V)] \leq 4(\epsilon/\rho)^2$.

This lemma generalizes the lemma of Kelner and Spielman [10] by allowing both v_1 and v_2 to be drawn from ρ-perturbed distributions, as opposed to requiring one of them to be uniformly random. Its proof is deferred to the full version of this paper.

Naturally, the smaller the perturbation, the weaker the bound in the theorem. However setting $\rho = \frac{1}{\sqrt{2m}}$ for example, gives the linear bound $8\pi m$ which is just a little larger than the bound on the number of shadow edges for the average case. Finally note that by Lemma 1, these bounds imply linear (in the number of graph edges) average and smoothed bounds for the number of optimal paths in the parametric shortest paths problem as well.

5 Extensions to Other Distributions

5.1 Poisson and Additive Stochastic Dominant Distributions—Exact Solution

The probability distribution $\mathcal{D}(\lambda)$ is called *additive* if the sum of two independent random variables with distributions $\mathcal{D}(\lambda_1)$ and $\mathcal{D}(\lambda_2)$ is another random variable with the same distribution and parameter equal to the sum, $\mathcal{D}(\lambda_1 + \lambda_2)$. With a slight abuse of notation we use $\mathcal{D}(\lambda)$ to also denote a random variable with this distribution. Assume in addition that the distribution $\mathcal{D}$ satisfies *stochastic dominance*, that is $\Pr(\mathcal{D}(\lambda_1) \leq t) \geq \Pr(\mathcal{D}(\lambda_2) \leq t)$ whenever $\lambda_1 \leq \lambda_2$. Examples of such distributions are Poisson and $gamma(a, b)$ with constant parameter b.

Suppose the random length of edge i is $X_i \sim \mathcal{D}(\lambda_i)$. Now, despite the non-separable objective function, the form of distribution makes the problem separable:

$$\Pr\Big(\sum_{i\in\pi} X_i \leq t\Big) = \Pr\Big(\sum_{i\in\pi} \mathcal{D}(\lambda_i) \leq t\Big) = \Pr\Big(\mathcal{D}\big(\sum_{i\in\pi} \lambda_i\big) \leq t\Big) \geq \Pr\Big(\mathcal{D}(\lambda') \leq t\Big),$$

where the last inequality follows from the stochastic dominance property of the distribution for all $\lambda' \geq \sum_{i\in\pi} \lambda_i$. With this, the optimal path is the one that has the smallest

sum of distribution parameters along its links and can be found exactly with a deterministic shortest path algorithm.

5.2 Exponential PTAS and Bernoulli QPTAS

Unlike the Poisson, the exponential distribution is not additive and we cannot write a simple closed form expression for the objective function. We propose a polynomial-time approximation scheme, based on dynamic programming over a discretization of the distribution parameter space. More precisely, we give a bicriterion approximation, which is given a lateness tolerance p and for $\epsilon > 0$ it finds an ST-path π satisfying

$$\Pr\Big(\sum_{i\in\pi} X_i < t(1+\epsilon)\Big) > 1-\epsilon p.$$

Due to space constraints, we defer the algorithm description to the full paper version and only state its running time.

Theorem 6. *An approximately optimal path π with $\Pr[\sum_{i\in\pi} X_i > (1+\epsilon)] \le p(1+\epsilon)$ can be computed in time $O(n^4 \log n) O\Big(\frac{\log(1/\epsilon)}{\epsilon^4} \log \frac{1}{\epsilon p}\Big) \gamma^{O(\frac{1}{\epsilon}\log 1/\epsilon)}$.*

A similar discretization of the state space yields a quasi-polynomial approximation scheme for the case of Bernoulli distributions; we omit the details from this version.

6 Conclusion

We have considered a novel framework for stochastic shortest paths with independent random edge lengths. When the edges are normally distributed, we give an exact $n^{\Theta(\log n)}$ algorithm. Several points worth noting are that this is an unusual algorithm (not based on dynamic programming) with an unusual running time for the classic shortest path problem in the presence of uncertainty. Although the problem is inherently discrete, in its core are properties from continuous optimization. One possibility to prove a polynomial worst-case bound on our $n^{\Theta(\log n)}$ algorithm is to restrict the class of graphs under consideration. We conjecture that the number of extreme points on the corresponding shadow dominant of planar graphs is polynomial (linear) in the size of the graph.

We present polynomial average and smoothed bounds with respect to the means and variances of the edge length distributions in the case of normal distributions. We note that these bounds hold for the maximization of any quasi-convex function of rank 2 (that is, a function of the form $f(a \cdot x, b \cdot x)$ for vectors $a, b \in \mathcal{R}^m$) over general polytopes with $0-1$ vertex coordinates. Our results could be further generalized [18] and apply to diverse other settings as well as serve of independent interest to non-convex optimization.

Other open questions for our stochastic shortest path model include considering correlated as well as dynamically varying edge length distributions.

Acknowledgement. We thank Brian Dean, David Karger, Asu Ozdaglar and Santosh Vempala for valuable suggestions.

References

1. H. Ackermann, A. Newman, H. Röglin, and B. Vöcking. Decision making based on approximate and smoothed pareto curves. In *Proc. of 16th ISAAC*, pages 675–684, 2005.
2. D. Bertsekas. *Dynamic Programming and Optimal Control*, volume II, 2nd Edition. Athena Scientific, Belmont, MA, 2001.
3. J. Boyan and M. Mitzenmacher. Improved results for route planning in stochastic transportation networks. In *Proc. of Symposium of Discrete Algorithms*, 2001.
4. P. Carstensen. *The complexity of some problems in parametric linear and combinatorial programming*. Ph.D. Thesis, Mathematics Dept., U. of Michigan, Ann Arbor, Mich., 1983.
5. B. Dean, M. Goemans, and J. Vondrak. Approximating the stochastic knapsack: the benefit of adaptivity. In *Proceedings of FOCS*, pages 208–217, 2004.
6. Y. Fan, R. Kalaba, and I. J. E. Moore. Arriving on time. *Journal of Optimization Theory and Applications*, forthcoming.
7. A. Goel and P. Indyk. Stochastic load balancing and related problems. In *Proceedings of the 40th Symposium on Foundations of Computer Science*, 1999.
8. R. Horst, P. M. Pardalos, and N. V. Thoai. *Introduction to Global Optimization*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 2000.
9. D. Karger, R. Motwani, and G. D. S. Ramkumar. On approximating the longest path in a graph. *Algorithmica*, 18:82–98, 1997.
10. J. A. Kelner and D. A. Spielman. A randomized polynomial-time simplex algorithm for linear programming. *Electronic Colloquium on Computational Complexity*, No. 156, 2005.
11. J. Kleinberg, Y. Rabani, and É. Tardos. Allocating bandwidth for bursty connections. *SIAM Journal on Computing*, 30(1):191–217, 2000.
12. R. P. Loui. Optimal paths in graphs with stochastic or multidimentional weights. *Communications of the ACM*, 26:670–676, 1983.
13. E. D. Miller-Hooks and H. S. Mahmassani. Least expected time paths in stochastic, time-varying transportation networks. *Transportation Science*, 34:198–215, 2000.
14. P. Mirchandani and H. Soroush. Optimal paths in probabilistic networks: a case with temporary preferences. *Computers and Operations Research*, 12(4):365–381, 1985.
15. J. Mote, I. Murthy, and D. Olson. A parametric approach to solving bicriterion shortest path problems. *European Journal of Operational Research*, 53:81–92, 1991.
16. K. Mulmuley and P. Shah. A lower bound for the shortest path problem. *Journal of Computer and System Sciences*, 63(2):253–267, 2001.
17. E. Nikolova, M. Brand, and D. R. Karger. Optimal route planning under uncertainty. In *Proceedings of International Conference on Automated Planning and Scheduling*, 2006.
18. E. Nikolova and J. A. Kelner. On the hardness and smoothed complexity of low-rank quasiconcave minimization. *Manuscript*, May, 2006.
19. S. Pallottino and M. G. Scutella. Shortest path algorithms in transportation models: Classical and innovative aspects. Technical Report TR-97-06, Universita di Pisa Dipartimento di Informatica, Pisa, Italy, 1997.
20. C. H. Papadimitriou and M. Yannakakis. Shortest paths without a map. *Theoretical Computer Science*, 84:127–150, 1991.
21. C. H. Papadimitriou and M. Yannakakis. On limited nondeterminism and the complexity of the V-C dimension. *Journal of Computer and System Sciences*, 53(2):161–170, 1996.
22. G. H. Polychronopoulos and J. N. Tsitsiklis. Stochastic shortest path problems with recourse. *Networks*, 27(2):133–143, 1996.

Path Hitting in Acyclic Graphs*

Ojas Parekh[1] and Danny Segev[2]

[1] Department of Mathematics and Computer Science, Emory University, USA
ojas@mathcs.emory.edu
[2] School of Mathematical Sciences, Tel-Aviv University, Israel
segevd@post.tau.ac.il

Abstract. An instance of the *path hitting* problem consists of two families of paths, $\mathcal{D}$ and $\mathcal{H}$, in a common undirected graph, where each path in $\mathcal{H}$ is associated with a non-negative cost. We refer to $\mathcal{D}$ and $\mathcal{H}$ as the sets of *demand* and *hitting* paths, respectively. When $p \in \mathcal{H}$ and $q \in \mathcal{D}$ share at least one mutual edge, we say that p *hits* q. The objective is to find a minimum cost subset of $\mathcal{H}$ whose members collectively hit those of $\mathcal{D}$.

In this paper we provide constant factor approximation algorithms for path hitting, confined to instances in which the underlying graph is a tree, a spider, or a star. Although such restricted settings may appear to be very simple, we demonstrate that they still capture some of the most basic covering problems in graphs.

1 Introduction

The input to the *path hitting* problem consists of two families of paths, $\mathcal{D}$ and $\mathcal{H}$, in a common undirected graph $G = (V, E)$, where each path $p \in \mathcal{H}$ is associated with a non-negative cost c_p. We refer to $\mathcal{D}$ and $\mathcal{H}$ as the sets of *demand* and *hitting* paths, respectively. When $p \in \mathcal{H}$ and $q \in \mathcal{D}$ share at least one mutual edge, we say that p *hits* (or *intersects*) q. The objective is to find a minimum cost subset of $\mathcal{H}$ whose members collectively hit those of $\mathcal{D}$. As we demonstrate in the sequel, numerous special cases of path hitting have been extensively studied; however, to the best of our knowledge, the present paper is the first to address this problem in its utmost generality.

Arbitrary graphs are well-understood. A rather straightforward lower bound on the approximability of path hitting can be derived by observing that it is at least as hard to approximate as *set cover*. Given an instance of the latter problem, with a ground set $U = \{e_1, \ldots, e_n\}$ and a collection $S_1, \ldots, S_m$ of subsets of U, we construct a path hitting instance as follows. The graph G is bipartite and complete with sides $\{x_1, \ldots, x_n\}$ and $\{y_1, \ldots, y_n\}$. The demand paths are $\{(x_1, y_1), \ldots, (x_n, y_n)\}$. For each subset S_i there is a corresponding hitting path p_i that traverses the edges $\{(x_j, y_j) : e_j \in S_i\}$ but none of the edges $\{(x_j, y_j) : e_j \notin S_i\}$. It is easy to see that for every $I \subseteq \{1, \ldots, m\}$ the subset of paths $\{p_i : i \in I\}$ hits all demand paths if and only if $\bigcup_{i \in I} S_i = U$. Therefore, path hitting cannot be approximated within a factor of $(1 - \epsilon) \ln |\mathcal{D}|$ for any $\epsilon > 0$,

* Due to space limitations, some technical details and proofs are omitted from this extended abstract. We refer the reader to the full version of this paper (currently available online at http://www.math.tau.ac.il/~segevd), in which all missing details are provided.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 564–575, 2006.

unless NP $\subset$ TIME($n^{O(\log \log n)}$) [6]. On the positive side, path hitting can be viewed as a special case of set cover: The set of elements to cover is $\mathcal{D}$ and the collection of subsets corresponds to $\mathcal{H}$, meaning that a path $p \in \mathcal{H}$ covers the demand paths it intersects, with cost c_p. This interpretation immediately implies an $O(\log |\mathcal{D}|)$ approximation for path hitting, by applying the greedy set cover algorithm [13, 16].

Implicit demands. Another related observation is that path hitting generalizes the *multicut* problem, in which given an undirected graph with non-negative edge costs and a collection of k pairs of vertices, $\{s_1, t_1\}, \ldots, \{s_k, t_k\}$, we seek a minimum cost set of edges whose removal disconnects each of these pairs. The latter problem can be restated as that of simultaneously hitting each s_i-t_i path using an edge set of minimum total cost. However, in this context the demand paths are represented implicitly, by specifying which pairs should be disconnected. While Garg, Vazirani and Yannakakis [9] devised an $O(\log k)$ approximation for the multicut problem, a hardness result of $\Omega(\log \log n)$ has recently been obtained by Chawla, Krauthgamer, Kumar, Rabani and Sivakumar [3], assuming a stronger version of the Unique Games Conjecture.

1.1 Motivation for Studying Restricted Topologies

In light of these observations, we focus our attention on the approximability of the path hitting problem confined to instances in which the underlying graph is a tree, a spider[1], or a star. Although such restricted settings may appear to be very simple, we proceed by demonstrating that they still generalize some of the most basic graph covering problems.

Edge dominating set. Let $G = (V, E)$ be an undirected graph, where each edge $e \in E$ is associated with a non-negative cost c_e. An edge e is said to *dominate* an edge f if $e \cap f \neq \emptyset$. The goal is to find a subset $E' \subseteq E$ of minimum total cost such that each edge of G is dominated by at least one member of E'. We note that this problem is known to generalize both *edge cover* and *vertex cover* (see, for example, [18]). Even when restricted to stars, path hitting captures the edge dominating set problem as a special case. Assuming that $V = \{v_1, \ldots, v_n\}$, we construct a star S on the vertex set $\{r, x_1, \ldots, x_n\}$, with r serving as a center. The demand and hitting paths are $\mathcal{D} = \mathcal{H} = \{\langle x_i, r, x_j \rangle : (v_i, v_j) \in E\}$, and the cost of $\langle x_i, r, x_j \rangle$ is $c_{(v_i, v_j)}$. There is a one-to-one correspondence between these two instances, since $E' \subseteq E$ dominates all edges of G if and only if $\{\langle x_i, r, x_j \rangle : (v_i, v_j) \in E'\}$ hits each demand path in $\mathcal{D}$. Carr, Fujito, Konjevod and Parekh [2] were the first to have achieved significant progress with respect to the weighted version of the edge dominating set problem, for which they proposed a $2\frac{1}{10}$-approximation. This factor was improved to 2 by Fujito and Nagamochi [8] and independently by Parekh [18].

Tree augmentation. Given an undirected tree $T = (V, E)$ and a set of auxiliary edges $\mathcal{E} \subseteq V \times V$ coupled with non-negative costs, the tree augmentation problem asks to identify a minimum cost subset of $\mathcal{E}$ whose addition to T makes the newly formed graph 2-edge connected. Menger's Theorem allows us to interpret this problem as a special case of path hitting: The demand paths are $\mathcal{D} = E$, whereas for each $(u, v) \in \mathcal{E}$

[1] A spider is a subdivision of a star or, alternatively, the result of identifying the roots of a collection of disjoint paths.

the unique path in T connecting u and v plays the role of a hitting path. Since tree augmentation has enjoyed sustained interest spanning decades, it is beyond the scope of this paper to present an inclusive overview, and the reader is referred to a short survey of directly related results [5, Sec. 1]. Nevertheless, we remark that there are several tree augmentation algorithms that achieve an approximation guarantee of 2 (for example, those in [7, 14] and variants of [11, 12]). In addition, an improved factor of $\frac{3}{2}$ was obtained by Even, Feldman, Kortsarz and Nutov [5] for the unweighted case.

Tree multicut. The relation to the multicut problem described earlier implies that when the input graph is a tree $T = (V, E)$ and $\mathcal{H} = E$, path hitting reduces to *multicut on a tree*. Garg et al. [10] proved that this problem is at least as hard to approximate as vertex cover. They also presented a primal-dual algorithm that constructs a feasible solution whose cost is at most twice the optimum. An LP-rounding algorithm with a similar approximation guarantee has recently been suggested by Levin and Segev [15]. We note that the hardness proof of Garg et al. can be easily modified to show that the problem of hitting subtrees of a given tree using its set of edges is at least as hard to approximate as set cover. Therefore, in an attempt to achieve a sub-logarithmic approximation factor, assuming that $\mathcal{D}$ consists of paths, rather than arbitrary subtrees, is indeed necessary.

1.2 Results and Techniques

The main contribution of this paper are LP-based approximation algorithms for path hitting on trees, spiders, and stars. As a secondary objective, we make a concentrated effort to unify the algorithmic methods utilized in approximating previously studied special cases. Our findings, and the techniques by which we derive them, can be briefly summarized as follows.

Descending paths. We begin by presenting a natural LP-relaxation of path hitting in trees, that results from formulating this problem as an integer covering program. Simple examples illustrate that there are instances for which this linear program does not have an integral optimal solution. Nevertheless, we constructively prove the existence of such a solution when the paths in $\mathcal{D}$ and $\mathcal{H}$ are *descending*, that is, each path has the property that one of its endpoints is an ancestor of the other, with respect to an arbitrary root we fix in advance. Our proof shows how to extend the algorithm of Garg et al. [10] so that it constructs an integral primal solution and a dual solution that satisfy complementary slackness conditions.

General paths. We make use of this integrality result to approximate the general problem, where $\mathcal{D}$ and $\mathcal{H}$ may contain non-descending paths, as follows. We first define a new set of hitting paths $\mathcal{H}'$ by performing *path splitting*, a preprocessing step in which each hitting path is replaced by two descending paths of equal cost. Then, using the optimal fractional solution we identify a new set of demand paths $\mathcal{D}'$, all of which are descending. Finally, we solve the problem of hitting $\mathcal{D}'$ using a minimum cost subset of $\mathcal{H}'$ to optimality, and translate the solution we obtain to a corresponding feasible subset of $\mathcal{H}$ whose cost is within factor 4 of optimum. When exactly one of $\mathcal{D}$ and $\mathcal{H}$ consists of descending paths, a simplification of our analysis yields an improved factor of 2, thus providing a new tree augmentation method and recovering the multicut algorithm of Levin and Segev [15]. These results are described in Section 2.

A new variant of edge cover. In an attempt to outdo the above-mentioned algorithm, we introduce and study the *edge cover with assignment* problem. Let $G = (V, E)$ be an undirected multigraph with edge costs c_e for each $e \in E$ and assignment costs $s_{v,e}$ for each $v \in V$ and $e \in \delta(v)$. A subset $E' \subseteq E$ is called an *edge cover* if every vertex $v \in V$ is adjacent to some edge in E'. The objective is to find an edge cover E' and a function $\phi : V \to E'$ that assigns each vertex to an adjacent edge in E' so as to minimize the sum of edge costs and assignment costs. When there are no assignment costs we obtain the standard edge cover problem, which is a matching problem in disguise that can be solved in polynomial time [4, 17]. In Section 3 we demonstrate that edge cover with assignment can be interpreted as an equivalent edge cover problem, which is created by modifying the given multigraph and its edge costs. This reduction enables us to derive a polyhedral description of the former problem by adapting that of the latter [1, 4]. However, rather than using this description we consider a simplified set of constraints, and prove that the resulting linear program has an integrality gap of exactly $\frac{4}{3}$.

Spiders and stars. One possible direction for improving the approximation factor of 4 is to avoid the path splitting step. In Section 4 we prove that this task can be accomplished when the input graph is a spider. In this case, we solve the LP-relaxation right away and utilize its optimal solution to identify a new set of descending demand paths $\mathcal{D}'$. We then formulate the problem of hitting $\mathcal{D}'$ using a subset of $\mathcal{H}$ as an instance of edge cover with assignment in an auxiliary multigraph. Finally, we solve the resulting problem, and prove that after translating its solution back to the original graph we obtain a feasible subset of $\mathcal{H}$ with cost of at most 3.219 times the optimum. Our analysis is based on combining the integrality result for descending paths and the integrality gap of the simplified relaxation of edge cover with assignment. In the full version of this paper, we also show that the suggested algorithm provides an $\frac{8}{3}$-approximation in stars.

2 Path Hitting in Trees

The main result of this section is an algorithm for path hitting in trees, with an approximation guarantee of 4. We begin by presenting a natural LP-relaxation of path hitting and its dual. Next, we extend the algorithm of Garg et al. [10] for multicuts in trees, and utilize complementary slackness conditions to prove that the new algorithm constructs an optimal solution when the paths in $\mathcal{D}$ and $\mathcal{H}$ are descending. Finally, we show how to manipulate this algorithm in order to approximate the general problem.

2.1 A Linear Program and Its Dual

Our algorithms and their analysis will be based on the following LP-relaxation of the path hitting problem:

$$\text{minimize} \quad \sum_{p \in \mathcal{H}} c_p x_p \tag{PH}$$

$$\text{subject to} \quad \sum_{p \in \mathcal{H} : p \cap q \neq \emptyset} x_p \geq 1 \qquad \forall\, q \in \mathcal{D} \tag{2.1}$$

$$x_p \geq 0 \qquad \forall\, p \in \mathcal{H} \tag{2.2}$$

In an integral solution, the variable x_p indicates whether we pick the path $p \in \mathcal{H}$, and constraint (2.1) ensures that each demand path is intersected by at least one of the hitting paths we pick. The dual of this linear program is:

$$\text{maximize} \quad \sum_{q \in \mathcal{D}} y_q \tag{DPH}$$

$$\text{subject to} \quad \sum_{q \in \mathcal{D}: q \cap p \neq \emptyset} y_q \leq c_p \qquad \forall\, p \in \mathcal{H} \tag{2.3}$$

$$y_q \geq 0 \qquad \forall\, q \in \mathcal{D} \tag{2.4}$$

To better understand (DPH), we first consider integral solutions to this program. The variables $\{y_q : q \in \mathcal{D}\}$ specify the number of copies we pick from each demand path q, and the objective is to maximize their sum. The primal costs now serve as capacities, and constraint (2.3) states that for each path $p \in \mathcal{H}$ the total number of copies of demand paths that are intersected by p does not exceed its capacity c_p. Therefore, the dual program can be viewed as a fractional *path packing* problem.

2.2 An Exact Algorithm for Descending Paths

In the following we consider the special case in which $\mathcal{D}$ and $\mathcal{H}$ consist of descending paths. We assume that the tree $T = (V, E)$ is rooted at an arbitrary vertex, which is fixed in advance. For $v \in V$, we denote by $\text{depth}(v)$ the length of the unique path in T connecting v to the root. In addition, for $u \neq v \in V$ we denote by $[u, v]$ the unique path in T connecting u and v. Finally, for each path $q \in \mathcal{D} \cup \mathcal{H}$ the endpoints of q are designated by u_q and l_q, with the convention that $\text{depth}(u_q) < \text{depth}(l_q)$.

The algorithm. Starting with an empty set of hitting paths P and the trivial dual solution $y = 0$, we proceed as follows.

1. For each $v \in V$, in non-increasing order of depth in T,
   (a) For each $q \in \mathcal{D}$ such that $u_q = v$, we increase the dual variable y_q as much as possible, without violating the capacity constraints.
   (b) We augment P by adding, in an arbitrary order, the paths in $\mathcal{H}$ whose dual constraint became tight during this iteration (henceforth, *saturated paths*).
2. For each $p \in P$, in reverse order of addition, if $P \setminus \{p\}$ is a feasible solution we eliminate p from P.

Analysis. Let P be the set of hitting paths produced by the algorithm, and let y be the corresponding dual solution. When step 1 terminates, each demand path $q \in \mathcal{D}$ is intersected by at least one saturated hitting path, or otherwise y_q could have been further increased, contradicting the maximality of y. It follows that P is indeed a feasible primal solution, as feasibility is maintained throughout step 2. In addition, since we never violate the capacity constraints, y is a feasible dual solution.

We now prove that P and y are optimal for (PH) and (DPH), respectively, by showing that these solutions satisfy complementary slackness conditions. In terms of P and y, the primal conditions state that for each $p \in \mathcal{H}$, if $p \in P$ then this path is saturated. The dual conditions state that for each $q \in \mathcal{D}$, if $y_q > 0$ then P contains exactly one hitting path that intersects q. The primal conditions are immediately implied by the construction of P, whereas the dual require a closer inspection of step 2.

Lemma 1. *Let $q \in \mathcal{D}$ be a path for which $y_q > 0$. Then q is intersected by exactly one path in P.*

Proof. Suppose that P contains two distinct paths, p_1 and p_2, that intersect q, where without loss of generality $\text{depth}(u_{p_1}) \leq \text{depth}(u_{p_2})$. Let $P|_{p_2}$ be the set of paths remaining in P just before p_2 is considered for elimination in step 2, and let $\mathcal{D}|_{p_2}$ be the set of demand paths for which p_2 is the only path in $P|_{p_2}$ that intersects them. We remark that p_1 and p_2 are members of $P|_{p_2}$, as these paths survived step 2.

We first show that for each $q' \in \mathcal{D}|_{p_2}$ we have $\text{depth}(u_{q'}) > \text{depth}(u_q)$ by considering two cases, depending on whether the edge set $p_1 \cap p_2 \cap q$ is empty or not.

Case I: $p_1 \cap p_2 \cap q = \emptyset$. min us .1emSince q is a descending path, there is a unique minimum depth edge in $p_1 \cap q$, which we denote by e_1. Similarly, e_2 is the unique minimum depth edge in $p_2 \cap q$. Clearly, $e_1 \neq e_2$, or otherwise $p_1 \cap p_2 \cap q \neq \emptyset$. In addition, by the assumption that $\text{depth}(u_{p_1}) \leq \text{depth}(u_{p_2})$ the edge e_2 must be deeper than e_1, and by definition of e_2 the upper endpoint of this edge is u_{p_2}, as shown in Figure 1a. Now suppose that $\text{depth}(u_{q'}) \leq \text{depth}(u_q)$. Note that q' is intersected by p_2 and contains $u_{q'}$, implying that q' traverses the edge e_1 (see Figure 1b). However, $e_1 \in p_1$ and therefore p_1 intersects q'. This contradicts the fact that p_2 is the only path in $P|_{p_2}$ that intersects q', since $p_1 \in P|_{p_2}$ and $p_1 \neq p_2$.

Case II: $p_1 \cap p_2 \cap q \neq \emptyset$. Let $e \in p_1 \cap p_2 \cap q$ and let r be the lower endpoint of e, as described in Figure 1c. As p_2 intersects q', exactly one of the following scenarios occurs:

1. $q' \cap [r, u_{p_2}] \neq \emptyset$. Since p_1 contains the edge e and the vertex u_{p_1}, $[r, u_{p_2}]$ is a subpath of p_1 and therefore p_1 intersects q'. Once again, this contradicts the fact that p_2 is the only path in $P|_{p_2}$ that intersects q'.
2. $q' \cap [r, u_{p_2}] = \emptyset$ and $q' \cap [r, l_{p_2}] \neq \emptyset$. Clearly, $\text{depth}(u_{q'}) \geq \text{depth}(r) > \text{depth}(u_q)$, where the second inequality holds since $e \in q$.

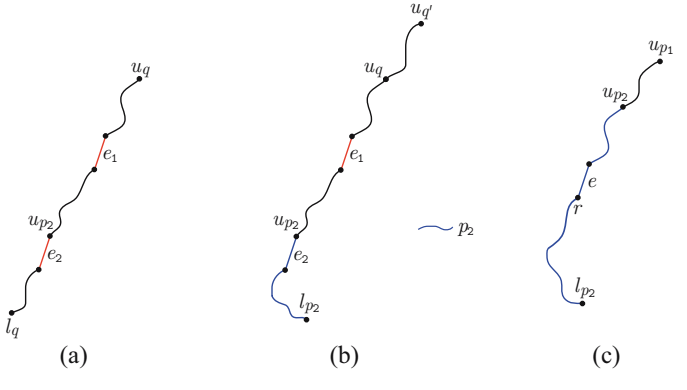


Fig. 1. (a) Case I: General configuration. (b) Any path that intersects p_2 and contains a vertex whose depth is at most that of u_q unavoidably traverses e_1. (c) Case II: General configuration.

We now show that p_2 should have been eliminated in step 2, by proving that $\mathcal{D}|_{p_2} = \emptyset$. Suppose that $\mathcal{D}|_{p_2} \neq \emptyset$ and let $q' \in \mathcal{D}|_{p_2}$. Since $\text{depth}(u_{q'}) > \text{depth}(u_q)$, the vertex $u_{q'}$ is processed in step 1 before u_q. Immediately after $u_{q'}$ is processed, P contains a path p' intersecting q', that became saturated in an earlier iteration or in the current one. Subsequently, when the iteration in which u_q is processed begins, the path p_2 is not saturated yet, since $y_q > 0$. Therefore, p_2 is added to P in this iteration or later. It follows that $p' \in P|_{p_2}$, since the paths in P are considered for elimination in reverse order of their addition, contradicting the assumption that $q' \in \mathcal{D}|_{p_2}$. □

Theorem 2. *When the paths in $\mathcal{D}$ and $\mathcal{H}$ are descending, the linear program* (PH) *has an integral optimal solution. This solution can be computed in polynomial time.*

2.3 An Algorithm for Arbitrary Paths

We now exploit the integrality result described in Theorem 2 to design an approximation algorithm for the general case, in which $\mathcal{D}$ and $\mathcal{H}$ may contain non-descending paths. For a path p in T, we denote by v_p^1 and v_p^2 the endpoints of this path and by $\text{LCA}(v_p^1, v_p^2)$ the lowest common ancestor of v_p^1 and v_p^2.

The algorithm. Let x^* be an optimal fractional solution to the linear program (PH).

1. We create a new set of hitting paths $\mathcal{H}'$, by replacing each $p \in \mathcal{H}$ with the pair of descending paths $[v_p^1, \text{LCA}(v_p^1, v_p^2)]$ and $[v_p^2, \text{LCA}(v_p^1, v_p^2)]$. Each of these paths is given a cost c_p.
2. We define a set of demand paths $\mathcal{D}'$ as follows. For each $q \in \mathcal{D}$, we add the path $q_1 = [v_q^1, \text{LCA}(v_q^1, v_q^2)]$ to $\mathcal{D}'$ if $\sum_{p \in \mathcal{H}: p \cap q_1 \neq \emptyset} x_p^* \geq \frac{1}{2}$ and otherwise we add $q_2 = [v_q^2, \text{LCA}(v_q^1, v_q^2)]$.
3. We apply the exact algorithm for descending paths to find a minimum cost subset $P' \subseteq \mathcal{H}'$ that hits $\mathcal{D}'$. The resulting solution is translated to a solution P for the original problem by picking $p \in \mathcal{H}$ if at least one of its replacements belongs to P'.

Analysis. We first argue that P is indeed a feasible solution. This claim follows from observing that the construction of $\mathcal{D}'$ in step 2 guarantees that it contains a subpath of each demand path in $\mathcal{D}$. Therefore, the paths in P' collectively hit those in $\mathcal{D}$, and this property is clearly preserved when P' is translated back to the original problem. We now show that the total cost of the paths in P is within factor 4 of optimum.

Lemma 3. $\sum_{p \in P} c_p \leq \sum_{p \in P'} c_p \leq 4 \cdot \text{OPT(PH)}$.

Theorem 4. *Path hitting in trees can be approximated to within a factor of* 4.

3 Edge Cover with Assignment

We temporarily deviate from the general theme of this paper and study the edge cover with assignment problem, formally defined in Subsection 1.2. This section introduces several tools that will allow us to obtain an improved approximation guarantee when the

input graph is a spider, as well as significantly simplify the presentation. We demonstrate that edge cover with assignment can be reduced to an equivalent edge cover problem, by modifying the given multigraph and its edge costs. We also consider a simple LP-relaxation of this problem and prove that its integrality gap is $\frac{4}{3}$.

A reduction to edge cover. Let I be an instance of the edge cover with assignment problem, consisting of a multigraph $G = (V, E)$ with edge costs c_e for each $e \in E$ and assignment costs $s_{v,e}$ for each $v \in V$ and $e \in \delta(v)$. Consider the following instance I' of the edge cover problem:

1. The new multigraph is G', with a vertex set $V' = V \cup \{a, b\}$ and an edge set $E' = E \cup \{(a, b)\} \cup \{(u, a) : u \in V\}$. In other words, we add a new edge (a, b) and connect a to each original vertex.
2. The cost of an original edge $e = (u, v)$ is $c'_e = c_e + s_{u,e} + s_{v,e}$; the cost of a new edge (u, a) is $c'_{(u,a)} = \min_{e \in \delta_G(u)}(c_e + s_{u,e})$; and the cost of (a, b) is $c'_{(a,b)} = 0$.

Lemma 5. *For each solution to I there exists a corresponding solution to I' of no greater cost, and vice versa.*

Theorem 6. *Edge cover with assignment can be solved as an edge cover problem.*

A simple LP-relaxation. Essential to our analysis will be the following LP-relaxation of edge cover with assignment:

$$\text{minimize} \quad \sum_{e \in E} c_e x_e + \sum_{v \in V} \sum_{e \in \delta(v)} s_{v,e} y_{v,e} \tag{ECA}$$

$$\text{subject to} \quad \sum_{e \in \delta(v)} y_{v,e} \geq 1 \qquad \forall\, v \in V \tag{3.1}$$

$$y_{v,e} \leq x_e \qquad \forall\, v \in V,\, e \in \delta(v) \tag{3.2}$$

$$x_e,\, y_{v,e} \geq 0 \qquad \forall\, v \in V,\, e \in \delta(v) \tag{3.3}$$

In an integral solution, the variable x_e indicates whether we pick the edge e, whereas $y_{v,e}$ indicates whether the vertex v is assigned to e. Constraint (3.1) ensures that each vertex is assigned to some adjacent edge, and constraint (3.2) states that the assignment of vertices is restricted to edges that we pick.

Theorem 7. *The integrality gap of* (ECA) *is exactly* $\frac{4}{3}$.

4 Path Hitting in Spiders

The main result of this section is an improved algorithm for path hitting in spiders that constructs a solution whose cost is within factor 3,219 of optimum. We initially consider the special case in which all demand paths are descending, and show how to exploit a number of structural properties in order to formulate this problem as edge cover with assignment. To approximate the general problem, we combine this formulation together with new insight into some of the results presented in Sections 2 and 3.

Notation. We assume that the given spider $S = (V, E)$ is rooted at its center r, and use $\mathcal{A}$ to denote the set of paths emerging from r, to which we also refer as the *arms* of S. The subset $\mathcal{H}^1 \subseteq \mathcal{H}$ is the set of hitting paths that are contained in a single arm of S, of which $\mathcal{H}^1_a$ are those contained in $a \in \mathcal{A}$. Similarly, $\mathcal{H}^2 \subseteq \mathcal{H}$ is the set of hitting paths that go through two arms of S, of which $\mathcal{H}^2_a$ are those that go through $a \in \mathcal{A}$ and an additional arm. For a path $p \in \mathcal{H}^2_a$, we use $\text{depth}_a(p)$ to denote the depth of the endpoint of p that resides on a. A demand path q that is contained in a is said to be located *below* a hitting path $p \in \mathcal{H}^2_a$ if the depth of the upper endpoint of q is at least $\text{depth}_a(p)$. In this case, p and q are edge-disjoint.

4.1 A Reformulation of Descending Demand Paths

In what follows we consider the special case in which the objective is to find a minimum cost subset of $\mathcal{H}$ that hits a collection of descending paths $\mathcal{D}'$. Let $\mathcal{H}^* \subseteq \mathcal{H}$ be an optimal solution to this problem. We observe that, with respect to each arm $a \in \mathcal{A}$, the set of paths $\mathcal{H}^*$ satisfies two structural properties.

1. If $\mathcal{H}^* \cap \mathcal{H}^2_a = \emptyset$, then the set of paths $\mathcal{H}^* \cap \mathcal{H}^1_a$ is an optimal solution to the problem of hitting the paths in $\mathcal{D}'$ that are contained in a using a subset of $\mathcal{H}^1_a$.
2. If $\mathcal{H}^* \cap \mathcal{H}^2_a \neq \emptyset$, let p_a be the path that maximizes $\text{depth}_a(p)$ over all paths in $\mathcal{H}^* \cap \mathcal{H}^2_a$. Then $\mathcal{H}^* \cap \mathcal{H}^1_a$ is an optimal solution to the problem of hitting the demand paths below p_a using a subset of $\mathcal{H}^1_a$.

These observations enable us to reformulate the subproblem we consider as an instance of edge cover with assignment in an auxiliary multigraph, by interpreting the cost of each subset $\mathcal{H}^* \cap \mathcal{H}^1_a$ as an assignment cost. Specifically, we begin by finding for each arm $a \in \mathcal{A}$ an optimal solution to the problem of hitting the paths in $\mathcal{D}'$ that are contained in a using a subset of $\mathcal{H}^1_a$. Since both $\mathcal{D}'$ and $\mathcal{H}^1_a$ consist of descending paths, this solution can be obtained by applying the algorithm described in Subsection 2.2, and we denote its cost by $\text{OPT}_{a,\emptyset}$. We then find for each $a \in \mathcal{A}$ and $p \in \mathcal{H}^2_a$ an optimal solution to the problem of hitting the demand paths below p using a subset of $\mathcal{H}^1_a$. Once again, we apply the algorithm for descending paths and use $\text{OPT}_{a,p}$ to denote the cost of the resulting solution. Based on these computations, we define an instance of edge cover with assignment on a multigraph G as follows:

1. The set of vertices is $\mathcal{A} \cup \{v_\emptyset\}$.
2. For each $a', a'' \in \mathcal{A}$ and $p \in \mathcal{H}^2_{a'} \cap \mathcal{H}^2_{a''}$, we add an edge $e_p = (a', a'')$ with cost c_p.
3. For each $a \in \mathcal{A}$, we add an edge $(a, v_\emptyset)$ of zero cost.
4. The assignment costs are: $s_{a,e_p} = \text{OPT}_{a,p}$ for each $a \in \mathcal{A}$ and $p \in \mathcal{H}^2_a$; $s_{a,(a,v_\emptyset)} = \text{OPT}_{a,\emptyset}$ for each $a \in \mathcal{A}$; and $s_{v_\emptyset,(a,v_\emptyset)} = 0$ for each $a \in \mathcal{A}$.

We now prove that the above reduction is indeed correct, by clarifying the equivalence between the original instance of path hitting and the resulting instance of edge cover with assignment.

Lemma 8. *There exist an edge cover $F \subseteq E(G)$ and an assignment function $\phi : V(G) \to F$ whose total cost is at most $\sum_{p \in \mathcal{H}^*} c_p$.*

Lemma 9. *Let $F \subseteq E(G)$ be an edge cover in G, and let $\phi : V(G) \to F$ be an assignment function. Then (F, ϕ) can be translated to a subset of $\mathcal{H}$ that hits $\mathcal{D}'$ with an identical cost.*

4.2 An Algorithm for Arbitrary Paths

In the following we design an improved algorithm for path hitting in spiders. The current algorithm departs from our approach for general trees in two aspects. First, we skip the path splitting step and add an elimination step in which redundant demand paths are discarded. Second, we solve the problem of hitting descending paths using the reduction to edge cover with assignment.

The algorithm. Let x^* be an optimal fractional solution to the linear program (PH).

1. We define a set of descending demand paths $\mathcal{D}'$ by[2]:
   (a) For each $q \in \mathcal{D}$, we add the path $q_1 = [v_q^1, \mathrm{LCA}(v_q^1, v_q^2)]$ to $\mathcal{D}'$ if $\sum_{p \in \mathcal{H}: p \cap q_1 \neq \emptyset} x_p^* \geq \frac{1}{2}$ and otherwise we add $q_2 = [v_q^2, \mathrm{LCA}(v_q^1, v_q^2)]$.
   (b) While there is a pair $q' \neq q'' \in \mathcal{D}'$ such that q' is a subpath of q'', we eliminate q'' from $\mathcal{D}'$.
2. We find a minimum cost subset $P \subseteq \mathcal{H}$ that hits $\mathcal{D}'$ as follows:
   (a) We reduce this problem to edge cover with assignment (see Subsection 4.1).
   (b) We find an optimal solution to the resulting instance (see Theorem 6) and translate it back to the original problem, as specified in Lemma 9.

4.3 Analysis

We first observe that P is indeed a feasible solution. The construction of $\mathcal{D}'$ in step 1a guarantees that it contains a subpath of each demand path in $\mathcal{D}$. In addition, for each path q'' that is eliminated in step 1b we leave in $\mathcal{D}'$ a witness ensuring that q'' is hit by P, in the form of a subpath of q''. In what follows we prove the next theorem.

Theorem 10. *The cost of P is at most* $\frac{4(1+\sqrt{2})}{3}\mathrm{OPT(PH)} < 3.219 \cdot \mathrm{OPT(PH)}$.

Let I denote the instance of edge cover with assignment that is produced by step 2a of the algorithm. Using notation similar to that in Subsection 4.1, consider the specialization of the LP-relaxation (ECA) for I, after picking in advance some edge $(a, v_\emptyset)$ and assigning $v_\emptyset$ to this edge (with zero cost):

$$\text{minimize} \quad \sum_{p \in \mathcal{H}^2} c_p x_{e_p} + \sum_{a \in \mathcal{A}} \sum_{p \in \mathcal{H}_a^2} \mathrm{OPT}_{a,p} y_{a,e_p} + \sum_{a \in \mathcal{A}} \mathrm{OPT}_{a,\emptyset} y_{a,(a,v_\emptyset)} \qquad (\mathrm{ECA}_I)$$

$$\text{subject to} \quad \sum_{p \in \mathcal{H}_a^2} y_{a,e_p} + y_{a,(a,v_\emptyset)} \geq 1 \qquad \forall\, a \in \mathcal{A} \qquad (4.1)$$

$$y_{a,(a,v_\emptyset)} \leq x_{(a,v_\emptyset)} \qquad \forall\, a \in \mathcal{A} \qquad (4.2)$$

$$y_{a,e_p} \leq x_{e_p} \qquad \forall\, a \in \mathcal{A},\ p \in \mathcal{H}_a^2 \qquad (4.3)$$

$$x_{e_p},\ x_{(a,v_\emptyset)},\ y_{a,e_p},\ y_{a,(a,v_\emptyset)} \geq 0 \qquad \forall\, a \in \mathcal{A},\ p \in \mathcal{H}^2 \qquad (4.4)$$

To avoid confusion, we denote by $\mathrm{OPT}_f(\cdot)$ the cost of an optimal fractional solution to a given linear program. Similarly, $\mathrm{OPT}_i(\cdot)$ is the cost of an optimal integral solution.

[2] Recall that v_q^1 and v_q^2 denote the endpoints of a given path q.

As specified in step 2b, the set of paths P is a translation of an optimal integral solution to (ECA$_I$). According to Lemma 9 this translation is cost-preserving, therefore $\sum_{p\in P} c_p = \text{OPT}_i(\text{ECA}_I)$. In addition, Theorem 7 states that the integrality gap of (ECA) is $\frac{4}{3}$, and we have $\sum_{p\in P} c_p \le \frac{4}{3}\text{OPT}_f(\text{ECA}_I)$. It follows that we can prove Theorem 10 by showing that the linear program (ECA$_I$) has a fractional solution $(\hat{x},\hat{y})$ whose cost is at most $(1+\sqrt{2})\text{OPT}_f(\text{PH})$.

Constructing $(\hat{x},\hat{y})$. Let $\theta \ge 2$ be a *fitting parameter*, whose value will be determined later. Using x^*, the optimal solution to (PH), we define $(\hat{x},\hat{y})$ as follows:

1. The fractional edges we pick are $\hat{x}_{e_p} = \theta x^*_p$ for each $p \in \mathcal{H}^2$, and $\hat{x}_{(a,v_\emptyset)} = 1$ for each $a \in \mathcal{A}$.
2. The fractional assignment of each $a \in \mathcal{A}$ is determined according to two cases. Let $\mathcal{H}^2_a = \{p_1,\ldots,p_k\}$, where we assume that these paths are indexed by non-increasing order of depth in a.
   (a) If $\sum_{i=1}^{k} x^*_{p_i} \ge \frac{1}{\theta}$, let $I(a)$ be the minimal index for which $\sum_{i=1}^{I(a)} x^*_{p_i} \ge \frac{1}{\theta}$. Then $\hat{y}_{a,(a,v_\emptyset)} = 0$, $\hat{y}_{a,e_{p_i}} = \theta x^*_{p_i}$ for $1 \le i \le I(a)-1$, $\hat{y}_{a,e_{p_i}} = 1-\theta\sum_{j=1}^{I(a)-1} x^*_{p_j}$ for $i = I(a)$, and $\hat{y}_{a,e_{p_i}} = 0$ otherwise.
   (b) If $\sum_{i=1}^{k} x^*_{p_i} < \frac{1}{\theta}$, then $\hat{y}_{a,(a,v_\emptyset)} = 1-\theta\sum_{i=1}^{k} x^*_{p_i}$, and $\hat{y}_{a,e_{p_i}} = \theta x^*_{p_i}$ for $1 \le i \le k$.

Lemma 11. *$(\hat{x},\hat{y})$ is a feasible solution to* (ECA$_I$).

Bounding the assignment cost of $(\hat{x},\hat{y})$. We are now concerned with bounding the total fractional assignment cost of some $a \in \mathcal{A}$ in terms of θ and x^*. Specifically, we relate this cost to that of the hitting paths in $\mathcal{H}^1_a$.

Lemma 12.

$$\text{OPT}_{a,\emptyset}\hat{y}_{a,(a,v_\emptyset)} + \sum_{p\in\mathcal{H}^2_a} \text{OPT}_{a,p}\hat{y}_{a,e_p} \le \begin{cases} \sup_{\alpha\in[0,\frac{1}{\theta})}\left\{\frac{2-\alpha(2+\theta)}{(1-\alpha)(1-2\alpha)}\right\}\sum_{p\in\mathcal{H}^1_a} c_p x^*_p & \text{if } \sum_{p\in\mathcal{H}^2_a} x^*_p < \frac{1}{\theta} \\ \frac{1}{1-1/\theta}\sum_{p\in\mathcal{H}^1_a} c_p x^*_p & \text{if } \sum_{p\in\mathcal{H}^2_a} x^*_p \ge \frac{1}{\theta} \end{cases}$$

Bounding the overall cost of $(\hat{x},\hat{y})$. We now show that the fitting parameter θ can be determined such that the total cost of $(\hat{x},\hat{y})$ is at most $(1+\sqrt{2})\sum_{p\in\mathcal{H}} c_p x^*_p$, completing the proof of Theorem 10. Let ρ denote the ratio between the fractional costs of the paths in $\mathcal{H}^2$ and the paths in $\mathcal{H}$, that is, $\rho = \frac{\sum_{p\in\mathcal{H}^2} c_p x^*_p}{\sum_{p\in\mathcal{H}} c_p x^*_p} \in [0,1]$. By definition of $\hat{x}$ and the bounds in Lemma 12, the cost of $(\hat{x},\hat{y})$ with respect to (ECA$_I$) is

$$\begin{aligned}
&\sum_{p\in\mathcal{H}^2} c_p\hat{x}_{e_p} + \sum_{a\in\mathcal{A}}\sum_{p\in\mathcal{H}^2_a} \text{OPT}_{a,p}\hat{y}_{a,e_p} + \sum_{a\in\mathcal{A}} \text{OPT}_{a,\emptyset}\hat{y}_{a,(a,v_\emptyset)} \\
&\quad\le \theta\sum_{p\in\mathcal{H}^2} c_p x^*_p + \max\left\{\sup_{\alpha\in[0,\frac{1}{\theta})}\left\{\frac{2-\alpha(2+\theta)}{(1-\alpha)(1-2\alpha)}\right\}, \frac{1}{1-1/\theta}\right\}\sum_{a\in\mathcal{A}}\sum_{p\in\mathcal{H}^1_a} c_p x^*_p \\
&\quad= \theta\sum_{p\in\mathcal{H}^2} c_p x^*_p + \sup_{\alpha\in[0,\frac{1}{\theta})}\left\{\frac{2-\alpha(2+\theta)}{(1-\alpha)(1-2\alpha)}\right\}\sum_{a\in\mathcal{A}}\sum_{p\in\mathcal{H}^1_a} c_p x^*_p \\
&\quad= \left(\rho\theta + (1-\rho)\sup_{\alpha\in[0,\frac{1}{\theta})}\left\{\frac{2-\alpha(2+\theta)}{(1-\alpha)(1-2\alpha)}\right\}\right)\sum_{p\in\mathcal{H}} c_p x^*_p \ ,
\end{aligned}$$

where the first equality follows from the observation that $\sup_{\alpha\in[0,\frac{1}{\theta})}\{\frac{2-\alpha(2+\theta)}{(1-\alpha)(1-2\alpha)}\} \geq \frac{1}{1-1/\theta}$. Although we do not control the parameter ρ, we can bound the ratio between the cost of $(\hat{x},\hat{y})$ and $\sum_{p\in\mathcal{H}} c_p x_p^*$ by considering the worst possible choice for ρ. Using elementary calculus, it is easy to verify that

$$\max_{\rho\in[0,1]} \min_{\theta\geq 2} \left(\rho\theta + (1-\rho) \sup_{\alpha\in[0,\frac{1}{\theta})} \left\{ \frac{2-\alpha(2+\theta)}{(1-\alpha)(1-2\alpha)} \right\} \right) = 1 + \sqrt{2} \ .$$

References

1. J. Aráoz, W. H. Cunningham, J. Edmonds, and J. Green-Krótki. Reductions to 1-matching polyhedra. *Networks*, 13:455–473, 1983.
2. R. Carr, T. Fujito, G. Konjevod, and O. Parekh. A $2\frac{1}{10}$-approximation algorithm for a generalization of the weighted edge-dominating set problem. *Journal of Combinatorial Optimization*, 5(3):317–326, 2001.
3. S. Chawla, R. Krauthgamer, R. Kumar, Y. Rabani, and D. Sivakumar. On the hardness of approximating multicut and sparsest-cut. In *20th CCC*, pages 144–153, 2005.
4. J. Edmonds and E. L. Johnson. Matching: A well-solved class of integer linear programs. In *Combinatorial Structures and their Applications*, pages 89–92. Gordon and Breach, New York, 1970.
5. G. Even, J. Feldman, G. Kortsarz, and Z. Nutov. A 3/2-approximation algorithm for augmenting the edge-connectivity of a graph from 1 to 2 using a subset of a given edge set. In *4th APPROX*, pages 90–101, 2001.
6. U. Feige. A threshold of $\ln n$ for approximating set cover. *Journal of the ACM*, 45(4):634–652, 1998.
7. G. N. Frederickson and J. JáJá. Approximation algorithm for several graph augmentation problems. *SIAM Journal on Computing*, 10(2):270–283, 1981.
8. T. Fujito and H. Nagamochi. A 2-approximation algorithm for the minimum weight edge dominating set problem. *Discrete Applied Mathematics*, 118(3):199–207, 2002.
9. N. Garg, V. V. Vazirani, and M. Yannakakis. Approximate max-flow min-(multi)cut theorems and their applications. *SIAM Journal on Computing*, 25(2):235–251, 1996.
10. N. Garg, V. V. Vazirani, and M. Yannakakis. Primal-dual approximation algorithms for integral flow and multicut in trees. *Algorithmica*, 18(1):3–20, 1997.
11. M. X. Goemans and D. P. Williamson. A general approximation technique for constrained forest problems. *SIAM Journal on Computing*, 24(2):296–317, 1995.
12. K. Jain. A factor 2 approximation algorithm for the generalized Steiner network problem. *Combinatorica*, 21(1):39–60, 2001.
13. D. S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9:256–278, 1974.
14. S. Khuller and R. Thurimella. Approximation algorithms for graph augmentation. *Journal of Algorithms*, 14(2):214–225, 1993.
15. A. Levin and D. Segev. Partial multicuts in trees. In *3rd WAOA*, pages 320–333, 2005.
16. L. Lovász. On the ratio of optimal integral and fractional covers. *Discrete Mathematics*, 13:383–390, 1975.
17. K. G. Murty and C. Perin. A 1-matching blossom type algorithm for edge covering problems. *Networks*, 12:379–391, 1982.
18. O. Parekh. Edge dominating and hypomatchable sets. In *13th SODA*, pages 287–291, 2002.

Minimum Transversals in Posi-modular Systems

Mariko Sakashita[1], Kazuhisa Makino[2],
Hiroshi Nagamochi[3], and Satoru Fujishige[4]

[1] Graduate School of Informatics, Kyoto University, Kyoto, 606-8501, Japan
sakasita@amp.i.kyoto-u.ac.jp
[2] Graduate School of Information Science and Technology, University of Tokyo, Tokyo, 113-8656, Japan
makino@mist.i.u-tokyo.ac.jp
[3] Graduate School of Informatics, Kyoto University, Kyoto, 606-8501, Japan
nag@amp.i.kyoto-u.ac.jp
[4] Research Institute for Mathematical Sciences, Kyoto University, Kyoto, 606-8502, Japan
fujishig@kurims.kyoto-u.ac.jp

Abstract. Given a system (V, f, d) on a finite set V consisting of two set functions $f : 2^V \to \mathbb{R}$ and $d : 2^V \to \mathbb{R}$, we consider the problem of finding a set $R \subseteq V$ of the minimum cardinality such that $f(X) \geq d(X)$ for all $X \subseteq V - R$, where the problem can be regarded as a natural generalization of the source location problems and the external network problems in (undirected) graphs and hypergraphs. We give a structural characterization of minimal deficient sets of (V, f, d) under certain conditions. We show that all such sets form a tree hypergraph if f is posi-modular and d is modulotone (i.e., each nonempty subset X of V has an element $v \in X$ such that $d(Y) \geq d(X)$ for all subsets Y of X that contain v), and that conversely any tree hypergraph can be represented by minimal deficient sets of (V, f, d) for a posi-modular function f and a modulotone function d. By using this characterization, we present a polynomial-time algorithm if, in addition, f is submodular and d is given by either $d(X) = \max\{p(v) \mid v \in X\}$ for a function $p : V \to \mathbb{R}_+$ or $d(X) = \max\{r(v, w) \mid v \in X, w \in V - X\}$ for a function $r : V^2 \to \mathbb{R}_+$. Our result provides first polynomial-time algorithms for the source location problem in hypergraphs and the external network problems in graphs and hypergraphs. We also show that the problem is intractable, even if f is submodular and $d \equiv \mathbf{0}$.

1 Introduction

Given a system (V, f, d) on a finite set V consisting of two set functions $f : 2^V \to \mathbb{R}$ and $d : 2^V \to \mathbb{R}$ with $f(\emptyset) \geq d(\emptyset)$, we consider the problem of finding a set $R \subseteq V$ of minimum cardinality such that $f(X) \geq d(X)$ for all $X \subseteq V - R$. The problem can be regarded as a natural generalization of the source location problems and the external network problems with edge-connectivity requirements in (undirected) graphs and hypergraphs [1, 6, 7, 13]; we will discuss these problems in Section 6. We give an interesting structural characterization of minimal *deficient* sets of (V, f, d), i.e., minimal sets $X \subseteq V$ such that $f(X) < d(X)$, under

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 576–587, 2006.

certain conditions. We show that all such sets form a tree hypergraph if f is posi-modular and d is modulotone (i.e., each nonempty subset X of V has an element $v \in X$ such that $d(Y) \geq d(X)$ for all subsets Y of X containing v), and that conversely any tree hypergraph can be represented by minimal deficient sets of (V, f, d) for a posi-modular function f and a modulotone function d. By using this characterization, we present a polynomial-time algorithm if, in addition, f is submodular and d is given by either $d(X) = \max\{p(v) \mid v \in X\}$ for a function $p : V \to \mathbb{R}_+$ or $d(X) = \max\{r(v, w) \mid v \in X, w \in V - X\}$ for a function $r : V^2 \to \mathbb{R}_+$.

As applications of our algorithm, we present first polynomial-time algorithms for the following problems:

1. The source location problem in hypergraphs with edge-connectivity requirements.
2. The external network problems in graphs and hypergraphs with edge-connectivity requirements.

We also show that the problem is intractable even if f is submodular and $d \equiv \mathbf{0}$. Namely, we show that the problem is NP-hard if a submodular function f is given as a functional form, and it requires at least $2^{\frac{n}{2}}$ time in the worst case, if f is given implicitly by an oracle, where $n = |V|$.

Our approach partly follows the idea by Bárász *et al.* [2] that is used to construct a polynomial time algorithm for the source location problem with a uniform demand function in *directed* networks. They introduced a new concept of *solid sets* for cut functions of directed graphs, proved that solid sets form a tree hypergraph, and gave an efficient algorithm for computing the underlying tree of the tree hypergraph by introducing a technique to reduce the size of solid sets required to obtain the tree. However, their proof is based on the properties of cut functions f of directed graphs (which cannot be generalized to submodular functions, as will be observed in Proposition 1) and the uniformness of demand functions (which cannot also be generalized to modulotonicity), while our proof is based on the posi-modularity of f and modulotonicity of demand functions.

The rest of this paper is organized as follows. In Section 2 we formulate our problem, introduce some known results, and present our structural result on the minimal deficient sets. Section 3 shows the intractability of the problem, even if f is submodular. Section 4 reveals the structural properties of posi-modular systems and gives a proof of our new hypertree characterization (Theorem 4). Section 5 describes a polynomial-time algorithm if f is submodular and posi-modular. Section 6 addresses applications of our problem.

Due to the space limitation, some of the proofs can be omitted.

2 Preliminaries

2.1 Hypergraphs

Let V be a finite set. A family $\mathcal{E} \subseteq 2^V$ is called a *Sperner family* if for arbitrary two distinct sets $E, E' \in \mathcal{E}$, neither $E \subseteq E'$ nor $E' \subseteq E$ holds. For a family

$\mathcal{E} \subseteq 2^V$, the hypergraph $(V, \mathcal{E})$ is simply written as $\mathcal{E}$. For a hypergraph $\mathcal{E}$, a subset $R \subseteq V$ is called a *transversal* (or *hitting set*) of $\mathcal{E}$ if $R \cap E \neq \emptyset$ for all $E \in \mathcal{E}$. Let $\tau(\mathcal{E})$ denote the *transversal number* of $\mathcal{E}$, i.e.,

$$\tau(\mathcal{E}) = \min\{|R| \mid R \text{ is a transversal of } \mathcal{E}\}.$$

A subfamily $\mathcal{E}' \subseteq \mathcal{E}$ is called a *matching* of $\mathcal{E}$ if $E \cap E' = \emptyset$ for arbitrary two distinct sets $E, E' \in \mathcal{E}'$, and let $\nu(\mathcal{E})$ denote the *matching number* of $\mathcal{E}$, i.e.,

$$\nu(\mathcal{E}) = \max\{|\mathcal{E}'| \mid \mathcal{E}' \text{ is a matching of } \mathcal{E}\}.$$

A hypergraph $\mathcal{E}$ is called a *tree hypergraph* (or *hypertree*) if there exists a tree T with a vertex set V such that each hyperedge $E \in \mathcal{E}$ induces a subtree of T. Such a tree T is called a *basic tree* for the hypergraph $\mathcal{E}$. For a tree hypergraph, the following result is known.

Theorem 1. (e.g., [3]) *Let $\mathcal{E}$ be a tree hypergraph. Then $\mathcal{E}$ satisfies the König property, i.e., $\tau(\mathcal{E}) = \nu(\mathcal{E})$.*

We review two characterizations of tree hypergraphs. A hypergraph $\mathcal{E}$ is said to have the *Helly property* if every subfamily of pairwise intersecting hyperedges has a nonempty intersection. The *line* (or *intersecting*) *graph* $L(\mathcal{E})$ of a hypergraph $\mathcal{E}$ is a graph in which the vertices correspond to the hyperedges, two of them being adjacent if the corresponding hyperedges have a nonempty intersection. An undirected graph is called *chordal* if every cycle of length at least 4 has a chord.

Theorem 2. (e.g., [10]) *A family $\mathcal{E} \subseteq 2^V$ is a tree hypergraph if and only if $\mathcal{E}$ has the Helly property and its line graph $L(\mathcal{E})$ is chordal.*

Another characterization is known as follows. Define a weight function $c(u, v)$ on the edge set of the complete graph on V as follows. For every pair $\{u, v\}$ of elements in V, let $c(u, v)$ be the number of hyperedges containing both u and v.

Theorem 3. (Bárász et al.[2]) *A family $\mathcal{E}$ is a tree hypergraph if and only if a spanning tree of maximum c-weight has weight $\sum_{E \in \mathcal{E}}(|E| - 1)$. Furthermore, such a spanning tree is a basic tree for $\mathcal{E}$.*

It follows from Theorem 3 that any maximum spanning tree algorithm can be used to compute a basic tree for a tree hypergraph.

2.2 Transversals over Set Functions

In this paper, we consider a system (V, f, d) on a finite set V consisting of two set functions $f : 2^V \to \mathbb{R}$ and $d : 2^V \to \mathbb{R}$ with $f(\emptyset) \geq d(\emptyset)$, and introduce the following problem:

$$\begin{array}{ll} \text{Minimize} & |R| \\ \text{subject to} & f(X) \geq d(X) \text{ for all } X \subseteq V - R \\ & R \subseteq V. \end{array} \tag{1}$$

Here $f(\emptyset) \geq d(\emptyset)$ is necessary for the problem to have a feasible solution. A vertex subset $X \subseteq V$ is called *deficient* if $f(X) < d(X)$. A deficient set X is called *minimal* if no proper subset of X is deficient. Let $\mathcal{W}(f, d)$ denote the family of all minimal deficient sets of (V, f, d). Then the constraint in problem (1) is equivalent to

$$R \cap X \neq \emptyset \quad \text{for all } X \in \mathcal{W}(f, d). \tag{2}$$

Therefore, it follows that the problem we consider is to compute a minimum transversal R of $\mathcal{W}(f, d)$. We remark that $\mathcal{W}(f, d)$ is not given explicitly and $|\mathcal{W}(f, d)|$ may be exponential in $n = |V|$.

A set function $f : 2^V \to \mathbb{R}$ is called *submodular* if

$$f(X) + f(Y) \geq f(X \cup Y) + f(X \cap Y) \tag{3}$$

for arbitrary two subsets X, Y of V, and *posi-modular* if

$$f(X) + f(Y) \geq f(X - Y) + f(Y - X) \tag{4}$$

for arbitrary two subsets X, Y of V. We call a function $d : 2^V \to \mathbb{R}$ *modulotone* if each nonempty $X \subseteq V$ has an element $v \in X$ such that $d(Y) \geq d(X)$ for all $Y \subseteq X$ containing v.

One of the main contributions of this paper is to derive the following new characterization of tree hypergraphs in terms of set functions.

Theorem 4. *A Sperner family $\mathcal{E} \subseteq 2^V$ is a tree hypergraph if and only if $\mathcal{E} = \mathcal{W}(f, d)$ holds for a posi-modular function $f : 2^V \to \mathbb{R}$ and a modulotone function $d : 2^V \to \mathbb{R}$.*

3 Submodular Systems

This section shows that problem (1) with a submodular function f is intractable in general and $\mathcal{W}(f, d)$ may not be a tree hypergraph. We first show that every Sperner hypergraph $\mathcal{E}$ can be represented by $\mathcal{W}(f, d)$ of a submodular function f and a constant function d.

Lemma 1. *For a Sperner hypergraph $\mathcal{E} \subseteq 2^V$, let $d, f : 2^V \to \mathbb{R}$ be functions defined by*

$$d(X) = 0 \quad \text{for all } X \subseteq V, \tag{5}$$
$$f(X) = -|\mathcal{E}(X)| \quad \text{for all } X \subseteq V, \tag{6}$$

where $\mathcal{E}(X) = \{E \in \mathcal{E} \mid E \subseteq X\}$. Then f is submodular and it holds that

$$\mathcal{E} = \mathcal{W}(f, d).$$

Since it is NP-hard to compute a minimum transversal of a general Sperner hypergraph, Lemma 1 implies the following hardness result.

Theorem 5. *Let d be a function defined by $d \equiv \mathbf{0}$. For a given Sperner hypergraph $\mathcal{E} \subseteq 2^V$, let f be a submodular function given by* (6)*. Then it is* NP*-hard to compute a minimum transversal of $\mathcal{W}(f, d)$.*

We also have the following complexity result if f is given by an oracle, i.e., we can invoke the oracle for the evaluation of $f(X)$ for any $X \subseteq V$ and use the function value $f(X)$.

Theorem 6. *Let f be a submodular function given by an oracle. Then problem* (1) *requires at least $2^{\frac{n}{2}}$ calls to the oracle in the worst case, where $n = |V|$.*

Lemma 1 also implies that $\mathcal{W}(f, d)$ is not a tree hypergraph even if f is submodular, which contrasts to the result on posi-modular functions f.

Proposition 1. *The family $\mathcal{W}(f, d)$ is not always a tree hypergraph, even if f is a submodular function and d is given by $d \equiv \mathbf{0}$.*

4 Posi-modular Systems

This section discusses problem (1) for posi-modular functions f. We first prove the sufficiency of Theorem 4, where the necessity will be shown in Section 4.2.

4.1 Structure of Posi-modular Functions

Let V be a finite set and $f : 2^V \to \mathbb{R}$ be a posi-modular function.

Basic Properties of posi-modular Functions. This section gives two properties of posi-modular functions.

Lemma 2. *Let $X_0, X_1, \ldots, X_{h-1}, X_h$ $(= X_0)$ $(h \geq 3)$ be subsets of V such that $X_i \cap X_j \neq \emptyset$ if and only if i and j are consecutive integers. For each $i = 0, \ldots, h-1$, let $Y_i = X_i \cap X_{i+1}$. Then any posi-modular function f satisfies*

$$\sum_{i=0}^{h-1} f(X_i) \geq \sum_{i=0}^{h-1} f(Y_i).$$

Let $\mathcal{X} = \{X_1, \ldots, X_h\} \subseteq 2^V$ and $I = \{1, \ldots, h\}$. For each subset J of I, let

$$Z_J = \bigcap_{j \in J} X_j. \tag{7}$$

Lemma 3. *If $\mathcal{X}$ is pairwise intersecting, i.e., $X_i \cap X_j \neq \emptyset$ for all $i, j \in I$, then for any posi-modular function f on 2^V, we have*

$$\sum_{i=1}^{h} f(X_i) \geq \sum_{i=1}^{h} f(Z_{I \setminus \{i\}} - Z_I).$$

Solid Sets. For an element $v \in V$, we call a nonempty subset X of V *v-solid* (with respect to f) if $v \in X$ and $f(X) < f(Y)$ for all nonempty proper subsets Y of X containing v. For each $v \in V$, we denote by $\mathcal{S}_v$ the family of all v-solid sets. Let $\mathcal{S}(f) = \bigcup_{v \in V} \mathcal{S}_v$. We prove that $\mathcal{S}(f)$ is a tree hypergraph if f is posi-modular. For each subset $X \subseteq V$, let $A_X = \{v \in X \mid X \in \mathcal{S}_v\}$.

Lemma 4. *Let X and Y be sets in $\mathcal{S}(f)$ such that $X \cap Y \neq \emptyset$. Then A_X or A_Y is included in $X \cap Y$, if f is a posi-modular function.*

Proof. We suppose that $X - Y$ and $Y - X$ are both nonempty, since the lemma clearly holds if $X \subseteq Y$ or $Y \subseteq X$. By the posi-modularity, we have

$$f(X) \geq f(X - Y) \quad \text{or} \quad f(Y) \geq f(Y - X).$$

By symmetry, we assume without loss of generality that $f(X) \geq f(X - Y)$. Then X cannot be v-solid for any $v \in X - Y$ since $X - Y$ is a nonempty proper subset of X. Therefore all elements $v \in X$ such that $X \in \mathcal{S}_v$ belong to $X \cap Y$, i.e., $A_X \subseteq X \cap Y$. □

Lemma 5. *The line graph L of $\mathcal{S}(f)$ is chordal, if f is a posi-modular function.*

Proof. Assuming that L is not chordal, we derive a contradiction. Let X_0, $X_1, \ldots, X_{h-1}, X_h$ $(= X_0)$ be a chordless cycle in $\mathcal{S}(f)$ of length at least 4. For each $i = 0, \ldots, h-1$, let $Y_i = X_i \cap X_{i+1}$. Then we have $Y_i \neq \emptyset$ for all $i = 0, \ldots, h-1$ and $Y_i \cap Y_j = \emptyset$ for all i and j with $i \neq j$. It follows from Lemma 4 that

$$A_{X_0} \subseteq Y_0 \quad \text{or} \quad A_{X_1} \subseteq Y_0.$$

By symmetry, we assume without loss of generality that $A_{X_1} \subseteq Y_0$. Then by applying Lemma 4 to X_1 and X_2, $A_{X_2} \subseteq Y_1$ holds, since $Y_0 \cap Y_1 = \emptyset$. By repeating this argument, we have

$$A_{X_{i+1}} \subseteq Y_i \quad \text{for } i = 0, 1, \ldots, h-1.$$

From this, $f(Y_i) > f(X_{i+1})$ holds for $i = 0, 1, \ldots, h-1$, since Y_i is a proper subset of X_{i+1} containing some v with $X_{i+1} \in \mathcal{S}_v$. Therefore we have

$$\sum_{i=0}^{h-1} f(Y_i) > \sum_{i=0}^{h-1} f(X_i),$$

which contradicts Lemma 2. □

We can also show the Helly property of $\mathcal{S}(f)$ by extending Lemma 4.

Lemma 6. *Let f be a posi-modular function, and let $\mathcal{X} = \{X_i \mid i \in I = \{1, \ldots, h\}\}$ be a pairwise intersecting subfamily of $\mathcal{S}(f)$. Then it has a set X_i such that $A_{X_i} \subseteq Z_I$, where Z_I is given as* (7).

Lemma 6 directly implies the following lemma.

Corollary 1. *If f is posi-modular, then $\mathcal{S}(f)$ has the Helly property.*

Lemma 5 and Corollary 1 imply the following theorem.

Theorem 7. *If f is posi-modular, then $\mathcal{S}(f)$ is a tree hypergraph.*

We are ready to prove the sufficiency of Theorem 4.

Lemma 7. *If f is a posi-modular function and d is a modulotone function, then $\mathcal{W}(f,d) \subseteq \mathcal{S}(f)$.*

Proof. Let X be a member of $\mathcal{W}(f,d)$. Then $f(X) < d(X)$ and $f(Y) \geq d(Y)$ for all nonempty proper subsets Y of X. From the assumption on d, there is an element $v \in X$ such that $d(Y) \geq d(X)$ for all $Y \subseteq X$ containing v. Therefore we have $f(Y) \geq d(Y) \geq d(X) > f(X)$ for all proper subsets Y of X containing v. That is, X is v-solid and hence $X \in \mathcal{S}(f)$ holds. □

Theorem 7 together with Lemma 7 implies the sufficiency of Theorem 4.

4.2 Necessity of Theorem 4

Let us show the necessity of Theorem 4.

For a hypergraph $\mathcal{E} \subseteq 2^V$, let $w : \mathcal{E} \to \mathbb{R}_+$ be a nonnegative weight function on $\mathcal{E}$. Let us define $f, d : 2^V \to \mathbb{R}$ by

$$\begin{aligned} f(X) &= \textstyle\sum\{w(E) \mid E \in \mathcal{E}, E \cap X \neq \emptyset, E - X \neq \emptyset\}, \\ d(X) &= \max_{v \in X} d(v), \end{aligned} \tag{8}$$

where $d(\emptyset) = 0$ and $d(v) = \sum\{w(E) \mid v \in E \in \mathcal{E}\}$. Note that f is a cut function of the hypergraph. This implies that f is symmetric submodular, and hence it is posi-modular. It is clear that $d : 2^V \to \mathbb{R}_+$ is modulotone.

Theorem 8. *Let $\mathcal{E} \subseteq 2^V$ be a tree Sperner hypergraph. Then there is a weight function $w : \mathcal{E} \to \mathbb{R}_+$ such that $\mathcal{E} = \mathcal{W}(f,d)$ holds for two set functions f and d defined as above.*

5 Submodular and Posi-modular Systems

In this section, we assume that a function f is posi-modular and submodular and a function d is given by one of the following two forms, since applications discussed in the subsequent section satisfy this assumption.

1. For a given function $p : V \to \mathbb{R}_+$, let

$$d(X) = \begin{cases} \max\{p(v) \mid v \in X\} & \text{if } X \neq \emptyset \\ 0 & \text{if } X = \emptyset. \end{cases} \tag{9}$$

2. For a given function $r : V^2 \to \mathbb{R}_+$, let

$$d(X) = \begin{cases} \max\{r(u,v) \mid u \in X, v \in V - X\} & \text{if } X \neq \emptyset, V \\ 0 & \text{if } X = \emptyset \text{ or } V. \end{cases} \tag{10}$$

It is not difficult to see that d is modulotone in both cases.

Theorem 8 together with the result in [5, 6] implies that problem (1) with a posi-modular function f and a modulotone function d is solvable in $O(n^3\rho(n))$ time, if the feasibility (i.e., a given $R \subseteq V$ satisfies $f(X) \geq d(X)$ for all $X \subseteq V - R$) can be checked in $O(\rho(n))$ time. We first show that the feasibility (transversal) check is possible in polynomial time if f is posi-modular and submodular and d is given by either (9) or (10), which implies polynomiality of the problem. We then improve the complexity by using maximal s-avoiding t-solid sets, which will be defined later, where a similar technique can be found in [2].

We remark that it is open whether the feasibility check is possible in polynomial time for a posi-modular function f and a modulotone function d.

5.1 Transversal Check

We consider how to check whether a given set $R \subseteq V$ is a transversal.

Let us first consider a function d of the form (10). In this case, a subset $R \subseteq V$ is a transversal, i.e., $f(X) \geq d(X)$ for each $X \subseteq V - R$ if and only if

$$\min\{f(X) \mid u \in X \subseteq V - (R \cup \{v\})\} \geq r(u, v) \tag{11}$$

for each ordered pair $(u, v) \in V^2$. The value of the left-hand side of (11) is the minimum value of the submodular function $f' : 2^{V-(R\cup\{u,v\})} \to \mathbb{R}_+$ defined by

$$f'(X) = f(X \cup \{u\}). \tag{12}$$

Therefore we can check whether R is a transversal by minimizing the submodular function f' for every ordered pair $(u, v) \in V^2$. Since the submodular function minimization is solved in $O((n^6\gamma + n^7)\log n)$ time [8], where γ denotes the time required to compute the function value for each subset X, problem (1) can be solved in $O(n^3 \times n^2 \times (n^6\gamma + n^7)\log n) = O((n^{11}\gamma + n^{12})\log n)$ time.

Similarly, for functions d given by (9), the problem can be solved in $O((n^{10}\gamma + n^{11})\log n)$ time.

In the subsequent sections, we reduce these complexities.

5.2 Computing s-Avoiding Solid Sets

For $s, t \in V$ with $s \neq t$, by an *s-avoiding t-solid* set X we mean a t-solid subset of $V - \{s\}$. An s-avoiding t-solid set X is called *maximal* if X is not included in any other s-avoiding t-solid set. For each $s \in V$, let $\mathcal{S}^{(s)}$ be the family of maximal s-avoiding t-solid sets for $t \in V - \{s\}$, and let $\mathcal{S}^*(f) = \bigcup_{s\in V} \mathcal{S}^{(s)}$.

We consider minimizing a submodular function f, in particular, finding a subset X of $V - \{s\}$ containing t such that

$$f(X) = \min\{f(Y) \mid t \in Y \subseteq V - \{s\}\}. \tag{13}$$

From the submodularity of f, the family of the minimizers is closed under taking union and intersection. Let N_t^s denote a unique minimal member of this family.

Lemma 8. *For $s, t \in V$ with $s \neq t$, N_t^s is a unique maximal s-avoiding t-solid set.*

Based on this, the family $\mathcal{S}^{(s)}$ can be obtained by computing all sets N_t^s for $t \in V - \{s\}$. We note that a unique minimal minimizer for a submodular function can be computed by using (strongly) polynomial algorithms for submodular function minimization (e.g., [8, 9, 11]) once. The best known algorithm due to [8] computes a maximal minimizer. Since the minimal minimizer can be obtained by executing it for the submodular function f^* defined by $f^*(X) = f(V - X)$ for all $X \subseteq V$, N_t^s can be computed in $O((n^6\gamma + n^7)\log n)$ time [8]. Thus $\mathcal{S}^*(f)$ can be computed in $O((n^8\gamma + n^9)\log n)$ time.

5.3 Computing a Basic Tree for $\mathcal{S}(f)$

From Theorem 3, given a tree hypergraph with the explicit list of the hyperedges, we can compute a basic tree and the algorithm is polynomial in $n = |V|$ and $m = |\mathcal{E}|$ [2].

Since $|\mathcal{S}^*(f)|$ is at most n^2 as mentioned above, we can compute a basic tree T for $\mathcal{S}^*(f)$ in polynomial time. Moreover, we can show that T is also a basic tree for $\mathcal{S}(f)$, where a similar proof can be found in [2].

Lemma 9. *If T is a basic tree for $\mathcal{S}^*(f)$, then it is basic for $\mathcal{S}(f)$.*

5.4 Computing a Minimum Transversal

In this section, we consider the problem of computing a minimum transversal R of $\mathcal{W}(f, d)$, i.e., a minimum size set $R \subseteq V$ such that $R \cap X \neq \emptyset$ for all $X \in \mathcal{W}(f, d)$.

Theorem 4 implies that $\mathcal{W}(f, d)$ is a tree hypergraph, and it follows from Lemmas 7 and 9 that a basic tree T for $\mathcal{W}(f, d)$ can be computed in polynomial time. It is known (e.g., [2]) that if a basic tree T is available, we can compute a minimum transversal by the following simple algorithm which uses the transversal check as a subroutine.

Choose an arbitrary element r of T, and regard T as an arborescence with a root r. Here $T[U]$ denotes the subtree of T induced by a vertex set U.

Algorithm MinTransversal

Input: A posi-modular function $f : 2^V \to \mathbb{R}$, a modulotone function $d : 2^V \to \mathbb{R}$ with $f(\emptyset) \geq d(\emptyset)$, and a basic tree T for $\mathcal{W}(f, d)$.

Output: A minimum transversal R of $\mathcal{W}(f, d)$.

Step 1. Initialize $R := \emptyset$ and $U := V$.

Step 2. If U is empty, then output R and halt.

Step 3. Choose a leaf v of $T[U]$ and $U := U - \{v\}$.

Step 4. If $R \cup U$ is not a transversal then $R := R \cup \{v\}$. Go to Step 2. □

Lemma 10. *Algorithm* MinTransversal *outputs a minimum transversal of $\mathcal{W}(f, d)$.*

5.5 Complexity

Given a posi-modular and submodular function $f : 2^V \to \mathbb{R}_+$ and a function $d : 2^V \to \mathbb{R}_+$ given by either (9) or (10), the algorithm outlined above for finding a minimum-size set $R \subseteq V$ such that $f(X) \geq d(X)$ for each $X \subseteq V - R$ consists of the following three steps:

1. Computing the family $\mathcal{S}^*(f)$.
2. Computing a basic tree T for $\mathcal{S}^*(f)$.
3. Computing a minimum transversal R of the family $\mathcal{W}(f, d)$ of all minimal deficient sets using T (Algorithm MinTransversal).

As discussed in Section 5.2, Step 1 can be done in $O((n^8\gamma + n^9)\log n)$ time. In Step 2, we first determine the weight function c in Theorem 3 and then construct a maximum weight spanning tree T. These can be executed in $O(n^2|\mathcal{S}^*(f)|) = O(n^4)$ time, since $|\mathcal{S}^*(f)| \leq n^2$. Since the time-consuming part of Step 3 (i.e., Algorithm MinTransversal) is to check whether $R \cup U$ is a transversal of $\mathcal{W}(f, d)$ for every $v \in V$, Step 3 can be performed in $O((n^8\gamma + n^9)\log n)$ time and $O((n^9\gamma + n^{10})\log n)$ time for functions d given by (9) and (10), respectively. The time bound of Step 3 dominates the time complexity of the entire algorithm.

Theorem 9. *Let f be a posi-modular and submodular function. Then problem* (1) *can be solved in $O((n^8\gamma + n^9)\log n)$ and $O((n^9\gamma + n^{10})\log n)$ time if d is given by* (9) *and* (10)*, respectively.*

We note that the complexities above are essentially $O(n^2 SFM(n))$ and $O(n^3 SFM\ (n))$, where $SFM(n)$ denotes the time complexity for minimizing a submodular function on 2^V with $n = |V|$. We also note that the algorithms are quadratically faster than the ones based on [5, 6] (See Section 5.1).

6 Applications of Problem (1)

In this section, we briefly discuss applications of our problem, where we focus on the source location problem and the external network problem in undirected graphs.

Let $G = (V, E)$ be an undirected graph with a capacity function $c : E \to \mathbb{R}_+$. It has a demand function $p : V \to \mathbb{R}_+$. Then the source location problem with edge-connectivity requirements in undirected graphs [1, 7, 12, 13] is given as

$$\begin{array}{ll} \text{Minimize} & |S| \\ \text{subject to} & \lambda_G(S, v) \geq p(v) \text{ for all } v \in V \\ & S \subseteq V, \end{array} \tag{14}$$

where $\lambda_G(S, v)$ denotes the maximum flow value (i.e., edge-connectivity) between S and v in G, and we define $\lambda_G(S, v) = +\infty$ if $v \in S$. This problem has been studied as a location problem concerned with network reliability.

Suppose that we are asked to locate a set S of multiple servers which can provide a certain service in a multimedia network $\mathcal{N}$. A user at vertex v can receive a service by connecting to a server in S through a path in $\mathcal{N}$. To ensure the quality of the service to v even if certain number $p(v) - 1$ of links become out of order, we should select S so that the edge-connectivity between S and v is at least $p(v)$. Therefore, this kind of fault-tolerant setting can be formulated as the source location problem.

For each $X \subseteq V$, let $f(X)$ denote the sum of edge capacities between X and $V - X$, i.e.,

$$f(X) = \sum\{c(u, v) \mid u \in X, v \in V - X, (u, v) \in E\}. \tag{15}$$

It is well known that f is called a cut function and is posi-modular and submodular. Then by the max-flow min-cut theorem, the source location problem can be regarded as problem (1) when f and d are respectively given as (15) and (9).

Let us next consider that the external network problem with edge-connectivity requirements in undirected graphs [6].

Let $G = (V, E)$ be an undirected graph with a capacity function $c : E \to \mathbb{R}_+$. It has a demand function $r : V^2 \to \mathbb{R}_+$. Then the external network problem is given as

$$\begin{array}{ll} \text{Minimize} & |S| \\ \text{subject to} & \lambda_{G/S}(u, v) \geq r(u, v) \text{ for all } (u, v) \in V^2 \\ & S \subseteq V, \end{array}$$

where G/S denotes the graph obtained from G by identifying vertex set S with a single vertex s, and if $u \in S$, we define $\lambda_{G/S}(u, v) = \lambda_{G/S}(s, v)$. Similarly to the source location problem, this has been studied as a network reliability problem [6]. Let f and d be respectively given as (15) and (10). Then by the max-flow min-cut theorem, the external network problem can be formulated as problem (1).

Let us now apply the algorithm given in Section 5.5 to these problems. In Step 1, for each s and t, N_t^s can be computed in $O(nm \log(n^2/m))$ time [4], where $n = |V|$ and $m = |E|$. Thus Step 1 can be done in in $O(n^3 m \log(n^2/m))$ time. As mentioned in Section 5.5, Step 2 can be executed in $O(n^4)$ time. Since each transversal check in Step 3 is, respectively, possible in $O(n^2 m \log(n^2/m))$ and $O(n^3 m \log(n^2/m))$ time for d given by (9) and (10), Step 3 can be done in $O(n^3 m \log(n^2/m))$ and $O(n^4 m \log(n^2/m))$ time, respectively.

Therefore, by using our general framework given in Section 5.5, we have the following result.

Corollary 2. *The source location problem and the external network problem are solvable in $O(n^3 m \log(n^2/m))$ and $O(n^4 m \log(n^2/m))$ time, respectively.*

We remark that this is the first polynomial-time algorithm for the external network problem and it is known that the source location problem can be solved in $O(n^2 m \log(n^2/m))$ time by using its own specific properties [1]. We also note

that our general framework is applicable for the source location problem and the external network problem for not only graphs $G = (V, E)$ but also hypergraphs $(V, \mathcal{E})$, where f is given as

$$f(X) = \sum\{c(E) \mid E \cap X, E \cap (V - X) \neq \emptyset, E \in \mathcal{E}\}. \tag{16}$$

Acknowledgements. This research was partially supported by the Scientific Grant-in-Aid from Ministry of Education, Science, Sports and Culture of Japan.

References

1. K. Arata, S. Iwata, K. Makino, and S. Fujishige: Locating sources to meet flow demands in undirected networks, *J. Algorithms*, **42** (2002), 54–68.
2. M. Bárász, J. Becker, and A. Frank: An algorithm for source location in directed graphs, *Operations Research Letters*, **33** (2005), 221–230.
3. A. Brandstädt, V. B. Le, and J. P. Spinrad: *Graph Classes: A Survey*, SIAM Monographs on Discrete Mathematics and Applications, SIAM, Philadelphia (1999).
4. J. Hao and J. B. Orlin: A faster algorithm for finding the minimum cut in a graph *J. Algorithms*, **17** (1994), 424–446.
5. J. van den Heuvel and M. Johnson: Transversals of subtree hypergraphs and the source location problem in digraphs, *CDAM Research Report*, LSE-CDAM-2004-10, London School of Economics.
6. J. van den Heuvel and M. Johnson: The external network problem with edge- or arc-connectivity requirements, *Workshop on Combinatorial and Algorithmic Aspects of Networking, CAAN 2004*, Lecture Notes in Computer Science **3405**, Springer (2004), 114–126.
7. H. Ito, H. Uehara, and M. Yokoyama: A faster and flexible algorithm for a location problem on undirected flow networks, *IEICE Trans.*, **E83-A** (2000), 704–712.
8. S. Iwata: A faster scaling algorithm for minimizing submodular functions, *SIAM J. Comput.*, **32** (2003), 833–840.
9. S. Iwata, L. Fleischer, and S. Fujishige: A combinatorial strongly polynomial algorithm for minimizing submodular functions, *J. of ACM*, **48** (2001), 761–777.
10. T. A. Mckee and F. R. McMorris: *Topics in Intersection Graph Theory*, SIAM Monographs on Discrete Mathematics and Applications, SIAM, Philadelphia (1999).
11. A. Schrijver: A combinatorial algorithm minimizing submodular functions in strongly polynomial time, *Journal of Combinatorial Theory, Series B*, **80** (2000), 346–355.
12. H. Tamura, M. Sengoku, S. Shinoda, and T. Abe: Some covering problems in location theory on flow networks, *IEICE Trans.*, **E75-A** (1992), 678–683.
13. H. Tamura, H. Sugawara, M. Sengoku, and S. Shinoda: Plural cover problem on undirected flow networks, *IEICE Trans.*, **J81-A** (1998), 863–869 (in Japanese).

An LP-Designed Algorithm for Constraint Satisfaction

Alexander D. Scott[1] and Gregory B. Sorkin[2]

[1] Mathematical Institute, University of Oxford, Oxford OX1 3LB, UK
scott@maths.ox.ac.uk
[2] Department of Mathematical Sciences, IBM T.J. Watson Research Center, Yorktown Heights NY 10598, USA
sorkin@watson.ibm.com

Abstract. The class Max $(r, 2)$-CSP consists of constraint satisfaction problems with at most two r-valued variables per clause. For instances with n variables and m binary clauses, we present an $\widetilde{O}(r^{19m/100})$-time algorithm. It is the fastest algorithm for most problems in the class (including Max Cut and Max 2-Sat), and in combination with "Generalized CSPs" introduced in a companion paper, also allows counting, sampling, and the solution of problems like Max Bisection that escape the usual CSP framework. Linear programming is key to the design as well as the analysis of the algorithm.

1 Introduction

A recent line of research has been to speed up exponential-time algorithms (deterministic or randomized) for maximization problems such as Max 2-Sat and Max Cut. For example, Gramm, Hirsch, Niedermeier and Rossmanith solve Max 2-Sat in time $\widetilde{O}(2^{m/5})$ and use this to solve Max Cut in time $\widetilde{O}(2^{m/3})$ [GHNR03], while Kulikov and Fedin solve Max Cut in time $\widetilde{O}(2^{m/4})$ [KF02], where m is the number of constraints or edges. (The $\widetilde{O}(\cdot)$ notation is defined in Section 2.)

The typical method is to repeatedly *transform* an instance to a smaller one or *split* it into several smaller ones (whence the exponential running time) until trivial instances are reached; the reductions are then reversed to recover a solution to the original instance. In [SS03] we introduced a new such method, distinguished by the fact that reducing an instance of Max Cut, for example, results in a problem that is not Max Cut, but where the reductions are closed over the larger class Max 2-CSP. This allowed the reductions to be simpler, fewer, and more powerful. The algorithm ran in time $\widetilde{O}(r^{m/5})$ (time $\widetilde{O}(2^{m/5})$ for binary-valued problems), making it the fastest for Max Cut, but tied for Max 2-Sat.

Results. The present $\widetilde{O}(r^{19m/100})$ algorithm is the fastest for Max Cut, Max 2-Sat, Max Dicut, weighted versions of these problems, less often considered problems like Max Ones 2-Sat, Max 2-Lin, and of course general Max 2-CSP; more efficient algorithms are known only for a few problems such as Maximum Independent Set (MIS). (For discussion of MIS and references, see the full version, which also addresses polynomial factors in an efficient implementation.)

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 588–599, 2006.

In combination with a "Generalized CSP" approach described in a companion paper [SS06], the algorithms here also enable (still in time $\widetilde{O}(r^{19m/100})$) counting CSP solutions of each possible cost; randomly sampling from optimal solutions, or according to the Gibbs measure or other distributions; and solving problems that do not fall into the Max 2-CSP framework, like Max Bisection, Sparsest Cut, judicious partitioning, Max Clique (without blowing up the input size), and multi-objective problems.

Techniques. We focus throughout on the graph supporting a CSP instance. The key step in our earlier $\widetilde{O}(r^{m/5})$ analysis was to use a linear program (LP) to show that the number of splitting reductions for an m-edge graph is $\leq m/5$. Consideration of an example which achieves that bound shows that any improvement must exploit *connected components* of the CSP's underlying graph. Conceptually, treatment of separate components sits uneasily with the LP analysis, which considers the (indivisible) degree sequence of the full graph: the usual argument that in case of component division "we are done, by induction" cannot be applied. However, a simple observation will sweep away the difficulty.

The LP was essential in the *design* of the new algorithm as well as its analysis. Its *primal* solution shows which reductions contribute to the worst case. We can easily exclude a bad reduction from the LP to see if an improved bound would result, and only then think hard about whether the reduction can be avoided.

The LP method presented is certainly applicable to reductions other than our own, and we hope to see it applied to algorithm design and analysis in contexts other than exponential-time algorithms and CSPs. (For a different use of LPs in automating an extremal construction, see [TSSW00].)

Literature Survey. The (a, b)-CSP model is extensively exploited for example in Beigel and Eppstein's [BE05]. An early version of our results was given in technical report [SS04]. We have already mentioned the Max 2-Sat algorithm of [GHNR03] and the Max Cut algorithm of [KF02]; [SS03] improved on the latter, and the present result improves on both. The lovely algorithm of Williams [Wil04], like ours, applies to all of 2-CSP. It runs in time $\widetilde{O}(n^{\omega/3})$, where $\omega < 2.376$ is the matrix-multiplication exponent. Depending on n rather than m, this algorithm is faster than ours if the average degree is above $200/(19\omega) < 4.430$. However, it requires exponential *space* of order $2^{2n/3}$.

Outline. In the next section we define the class Max 2-CSP, and in Section 3 we introduce the reductions our algorithms will use. In Section 4 we define and analyze the $\widetilde{O}(r^{m/5})$ algorithm as a relatively gentle introduction to the tools, including the LP analysis. The $\widetilde{O}(r^{19m/100})$ algorithm is presented in Section 5.

2 Max $(r, 2)$-CSP

The problem Max Cut is to partition the vertices of a given graph into two classes so as to maximize the number of edges "cut" by the partition. Think of each *edge* as being a *function* on the classes (or "colors") of its endpoints, with

value 1 if the endpoints are of different colors, 0 if they are the same: Max Cut is equivalent to finding a 2-coloring of the vertices which maximizes the sum of these edge functions. This view naturally suggests a generalization.

An *instance* (G, S) of Max $(r, 2)$-CSP is given by an "underlying" graph $G = (V, E)$ and a set S of "score" functions. Writing $[r] = \{1, \ldots, r\}$ for the set of available colors, we have a "dyadic" score function $s_e : [r]^2 \to \mathbb{R}$ for each edge $e \in E$, a "monadic" score function $s_v : [r] \to \mathbb{R}$ for each vertex $v \in V$, and finally a single "niladic" score "function" $s_\emptyset : [r]^0 \to \mathbb{R}$ which takes no arguments and is just a constant convenient for bookkeeping.

A *candidate solution* is a function $\phi : V \to [r]$ assigning "colors" to the vertices (we call ϕ an "assignment" or "coloring"), and its *score* is

$$s(\phi) := s_\emptyset + \sum_{v \in V} s_v(\phi(v)) + \sum_{uv \in E} s_{uv}(\phi(u), \phi(v)). \tag{1}$$

An *optimal solution* ϕ is one which maximizes $s(\phi)$.

Notation. We reserve the symbols G for the underlying graph of a Max $(r, 2)$-CSP instance, n and m for its numbers of vertices and edges, $[r] = \{1, \ldots, r\}$ for the allowed colors of each vertex, and $L = 1 + nr + mr^2$ for the input length. Since a CSP instance with $r < 2$ is trivial, we will assume $r \geq 2$ as part of the definition. For brevity, we write "d-vertex" for "vertex of degree d". The notation $\widetilde{O}(\cdot)$ suppresses polynomial factors in any parameters, so for example $\widetilde{O}(r^{cn})$ may mean $O(r^3 n\, r^{cn})$.

Remarks. The class Max $(r, 2)$-CSP is surprisingly flexible, and in addition to Max Cut and Max 2-Sat includes problems like MIS and minimum vertex cover that are not at first inspection structured around pairwise constraints. Readers familiar with the class $\mathcal{F}$-Sat will see that when the arity of $\mathcal{F}$ is limited to 2, Max $(r, 2)$-CSP also contains $\mathcal{F}$-Sat, $\mathcal{F}$-Max-Sat and $\mathcal{F}$-Min-Sat (e.g., Max 2-Sat and Max 2-Lin) and $\mathcal{F}$-Max-Ones.

3 Reductions

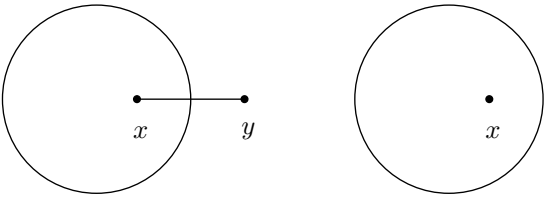
As with most of the works surveyed above, our algorithms are based on progressively reducing the instance to one with fewer vertices and edges until the instance becomes trivial. Because we work in the general class Max $(r, 2)$-CSP rather than trying to stay within a smaller class such as Max 2-Sat, our reductions are simpler and fewer than is typical. For example, [GHNR03] uses seven reduction rules; we have just three (plus a trivial "0-reduction" that other works may treat implicitly). The first two reductions each produce equivalent instances with one vertex fewer, while the third produces a set of r instances, each with one vertex fewer, one of which is equivalent to the original instance. We expand the previous notation (G, S) for an instance to (V, E, S), where $G = (V, E)$.

Reduction 0 (transformation). This is a trivial "pseudo-reduction". If a vertex y has degree 0 (so it has no dyadic constraints), then set $s_\emptyset = s_\emptyset + \max_{C \in [r]} s_y(C)$ and delete y from the instance entirely.

Reduction I. Let y be a vertex of degree 1, with neighbor x. Reducing (V, E, S) on y results in a new problem (V', E', S') with $V' = V \setminus y$ and $E' = E \setminus xy$. S' is the restriction of S to V' and E', except that for all colors $C \in [r]$ (and in total time $O(r^2)$) we set

$$s'_x(C) = s_x(C) + \max_{D \in [r]} \{s_{xy}(CD) + s_y(D)\}.$$

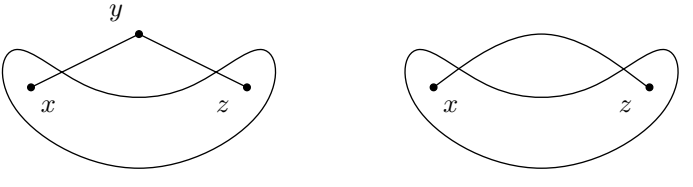
Note that any coloring ϕ' of V' can be extended to a coloring ϕ of V in r ways, depending on the color assigned to y. Writing (ϕ', D) for the extension in which $\phi(y) = D$, the defining property of the reduction is that $S'(\phi') = \max_D S(\phi', D)$. In particular, $\max_{\phi'} S'(\phi') = \max_\phi S(\phi)$, and an optimal coloring ϕ' for the instance (V', E', S') can be extended to an optimal coloring ϕ for (V, E, S).



Reduction II (transformation). Let y be a vertex of degree 2, with neighbors x and z. Reducing (V, E, S) on y results in a new problem (V', E', S') with $V' = V \setminus y$ and $E' = (E \setminus \{xy, yz\}) \cup \{xz\}$. S' is the restriction of S to V' and E', except that for $C, D \in [r]$ (and in total time $O(r^3)$) we set

$$s'_{xz}(CD) = s_{xz}(CD) + \max_{F \in [r]} \{s_{xy}(CF) + s_{yz}(FD) + s_y(F)\} \qquad (2)$$

if there was already an edge xz, discarding the first term $s_{xz}(CD)$ if there was not. As in Reduction I, any coloring ϕ' of V' can be extended to V in r ways, according to the color F assigned to y, and the defining property of the reduction is that $S'(\phi') = \max_F S(\phi', F)$. In particular, $\max_{\phi'} S'(\phi') = \max_\phi S(\phi)$, and an optimal coloring ϕ' for (V', E', S') can be extended to an optimal coloring ϕ for (V, E, S).



Reduction III (splitting). Let y be a vertex of degree 3 *or higher*. Where reductions I and II each had a single reduction of (V, E, S) to (V', E', S'), here we define r different reductions: for each color C there is a reduction of (V, E, S) to (V', E', S^C) corresponding to assigning the color C to y. We

define $V' = V \setminus y$, and E' as the restriction of E to $V \setminus y$. S^C is the restriction of S to $V \setminus y$, except that we set

$$(s^C)_0 = s_\emptyset + s_y(C),$$

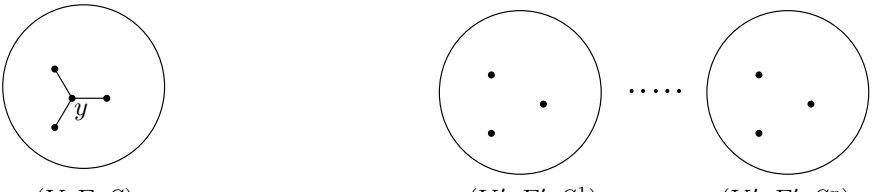
and, for every neighbor x of y and every $D \in [r]$,

$$(s^C)_x(D) = s_x(D) + s_{xy}(DC).$$

As in the previous reductions, any coloring ϕ' of $V \setminus y$ can be extended to V in r ways, (ϕ', C) where color C is given to y, and now (this is different!) $S^C(\phi') = S(\phi', C)$. Furthermore,

$$\max_C \max_{\phi'} S^C(\phi') = \max_\phi S(\phi),$$

and an optimal coloring on the left *is* an optimal coloring on the right.



(V, E, S) (V', E', S^1) (V', E', S^r)

4 An $\widetilde{O}(r^{m/5})$ Algorithm

As a warm-up to our $\widetilde{O}(r^{19m/100})$ algorithm, in this section we will present Algorithm A, which will run in time $O(nr^{3+m/5})$ and space $O(L)$. (Recall that $L = 1 + nr + mr^2$ is the input length.) Roughly speaking, a simple recursive algorithm for solving an input instance could work as follows. Begin with the input problem instance.

Given an instance $\mathcal{M} = (G, S)$:

1. If any reduction of type 0, I or II is possible (in that order of preference), apply it to reduce $\mathcal{M}$ to $\mathcal{M}'$, recording certain information about the reduction. Solve $\mathcal{M}'$ recursively, and use the recorded information to reverse the reduction and extend the solution to one for $\mathcal{M}$.
2. If only a type III reduction is possible, reduce (in order of preference) on a vertex of degree 5 or more, 4, or 3. For $i \in [r]$, recursively solve each of the instances $\mathcal{M}^i$ in turn, select the solution with the largest score, and use the recorded information to reverse the reduction and extend the solution to one for $\mathcal{M}$.
3. If no reduction is possible then the graph has no vertices, there is a unique coloring (the empty coloring), and the score is $s_\emptyset$ (from the niladic score function).

If the recursion depth is ℓ, the recursive algorithm's running time is $\widetilde{O}(r^\ell)$, and the preference order over type III reductions is needed to obtain the bound $\ell \leq m/5$ of Lemma 1; we prove this bound in Section 4. Numerous complications in optimizing the polynomial factors are addressed in the full paper.

Phases. While a III-reduction produces r different subinstances, all have the same underlying graph: the original graph with one vertex deleted. Type 0, I and II CSP reductions also change the underlying graph in a way independent of the score functions, so all the CSP reductions have graph-reduction counterparts depending only on the underlying graph and the reduction vertex. Our algorithm runs in three phases. The first, Algorithm A.1, finds a sequence of *n graph* reductions; this can be done in linear time and space, and our focus is to show that it has $\ell \leq m/5$ III-reductions. Details of this phase will allow us to assume the graph is always simple. The second phase finds an optimal cost, and the third produces a corresponding coloring.

Recursion Depth. The crux of the analysis is the following lemma.

Lemma 1. *Algorithm A.1 reduces a graph G with n vertices and m edges to an empty graph after $\ell \leq m/5$ III-reductions.*

Proof. While the graph has maximum degree 5 or more, Algorithm A.1 III-reduces only on such a vertex, destroying at least 5 edges; any I- or II-reductions included in the same step only increase the number of edges destroyed. Thus, it suffices to prove the lemma for graphs with maximum degree 4 or less. Since the reductions never increase the degree of any vertex, the maximum degree will always remain at most 4.

In this paragraph, we give some intuition for the rest of the argument. Algorithm A.1 III-reduces on 4-vertices as long as possible, before III-reducing on 3-vertices, whose neighbors must then all be of degree 3 (degrees 0, 1 or 2 would trigger a 0-, I- or II-reduction in preference to the III-reduction). Note that each III-reduction on a 3-vertex destroys 6 edges if we imagine immediately following up with II-reductions on its neighbors; similarly, reduction on a 4-vertex destroys at least 5 edges unless the 4-vertex has no degree-3 neighbor. The only problem comes from reductions on 4-vertices whose neighbors are also all of degree 4, as these destroy only 4 edges. Our LP analysis will show that because such reductions create degree-3 vertices, and the algorithm terminates with none, these bad reductions cannot occur too often.

We proceed by considering the various types of reductions and their effect on the number of edges and the number of 3-vertices. The reductions are catalogued in Table 1. The first row, for example, shows that III-reducing on a 4-vertex with 4 neighbors of degree 4 (and thus none of degree 3), destroys 4 edges, and (changing the neighbors from degree 4 to 3) destroys 5 4-vertices (including itself) and creates 4 3-vertices. The remaining rows up to the table's separating line similarly illustrate the other III-reductions. Below the line, II-reductions and I-reductions are decomposed into parts. As shown just below the line, a II-reduction, regardless of the degrees of the neighbors, first destroys 1 edge

Table 1. Tabulation of the effects of various reductions in Algorithm A.1

deg	#nbrs of deg				destroys					steps
	4	3	2	1	e	4	3	2	1	
4	4	0	0	0	4	5	−4	0	0	1
4	3	1	0	0	4	4	−2	−1	0	1
4	2	2	0	0	4	3	0	−2	0	1
4	1	3	0	0	4	2	2	−3	0	1
4	0	4	0	0	4	1	4	−4	0	1
3	0	3	0	0	3	0	4	−3	0	1
2					1	0	0	1	0	0
$\frac{1}{2}\,e$	1	0	0	0	$\frac{1}{2}$	1	−1	0	0	0
$\frac{1}{2}\,e$	0	1	0	0	$\frac{1}{2}$	0	1	−1	0	0
$\frac{1}{2}\,e$	0	0	1	0	$\frac{1}{2}$	0	0	1	−1	0
$\frac{1}{2}\,e$	0	0	0	1	$\frac{1}{2}$	0	0	0	1	0

and 1 2-vertex, and counts as 0 steps (steps count only III-reductions). In the process, the II-reduction may create a parallel edge, which may at some stage be deleted by Algorithm A.1. Since the exact effect of an edge deletion depends on the degrees of its neighbors, to minimize the number of cases we treat an edge deletion as two half-edge deletions, each destroying $\frac{1}{2}$ an edge, and whose effect depends on the degree of the half-edge's incident vertex. For example the table's next line shows deletion of a half-edge incident to a 4-vertex, changing it to a 3-vertex and destroying half an edge. The last four rows of the table also capture I-reductions. 0-reductions are irrelevant to the table, which does not consider vertices of degree 0.

The sequence of reductions reducing G to an empty graph can be parametrized by an 11-vector $\boldsymbol{n}$ giving the number of reductions (and partial reductions) indexed by the rows of the table, so for example its first element is the number of III-reductions on 4-vertices whose neighbors are also all 4-vertices. Since the reductions destroy all m edges, the dot product of $\boldsymbol{n}$ with the table's column "destroys e" (call it $\boldsymbol{e}$) must be precisely m. Since all 4-vertices are destroyed, the dot product of $\boldsymbol{n}$ with the column "destroys 4" (call it $\boldsymbol{d_4}$) must be ≥ 0, and the same goes for the "destroy" columns 3, 2 and 1. The number of III-reductions is the dot product of $\boldsymbol{n}$ with the "steps" column, $\boldsymbol{n} \cdot \boldsymbol{s}$. How large can the number of III-reductions $\boldsymbol{n} \cdot \boldsymbol{s}$ possibly be?

To find out, let us maximize $\boldsymbol{n} \cdot \boldsymbol{s}$ subject to the constraints that $\boldsymbol{n} \cdot \boldsymbol{e} = m$ and that $\boldsymbol{n} \cdot \boldsymbol{d_4}$, $\boldsymbol{n} \cdot \boldsymbol{d_3}$, $\boldsymbol{n} \cdot \boldsymbol{d_2}$ and $\boldsymbol{n} \cdot \boldsymbol{d_1}$ are all ≥ 0. Instead of maximizing over proper reduction collections $\boldsymbol{n}$, which seem hard to characterize, we maximize over the larger class of non-negative real vectors $\boldsymbol{n}$, giving an upper bound on the proper maximum. Maximizing the linear function $\boldsymbol{n} \cdot \boldsymbol{s}$ of $\boldsymbol{n}$ subject to a set of linear constraints (such as $\boldsymbol{n} \cdot \boldsymbol{e} = m$ and $\boldsymbol{n} \cdot \boldsymbol{d_4} \geq 0$) is simply solving a linear program (LP); the LP's constraint matrix and objective function are the part of Table 1 right of the double line. To avoid dealing with "m" in the LP, we set $\boldsymbol{n'} = \boldsymbol{n}/m$, and solve the LP with constraints $\boldsymbol{n'} \cdot \boldsymbol{e} = 1$, and as before $\boldsymbol{n'} \cdot \boldsymbol{d_4} \geq 0$, etc., to maximize $\boldsymbol{n'} \cdot \boldsymbol{s}$. The "$\boldsymbol{n'}$" LP is a small linear program (11

variables and 5 constraints) and its maximum is precisely 1/5, showing that the number of III-reduction steps — $\boldsymbol{n} \cdot \boldsymbol{s} = m\boldsymbol{n}' \cdot \boldsymbol{s}$ — is at most $m/5$.

This establishes that the number of type-III reductions can be at most 1/5th the number of edges m, concluding the proof.

Theorem 2. *A Max* $(r, 2)$*-CSP instance on* n *variables with* m *dyadic constraints and length* L *can be solved in time* $O(nr^{3+m/5})$ *and space* $O(L)$.

Proof. The theorem is an immediate consequence of Lemma 1 and the polynomial-factor considerations ignored in this version of the paper.

The LP's *dual* solution gives a "potential function" proof of Lemma 1. The dual assigns "potentials" to the graph's edges and to vertices according to their degrees, such that the number of steps counted for a reduction is at most its change to the potential. Since the potential is initially at most $0.20m$ and finally 0, the number of steps is at most $m/5$. The *primal* solution of the LP uses (proportionally) 1 III-reduction on a 4-vertex with all 4-neighbors, 1 III-reduction on a 3-vertex, and 3 II-reductions; reducing a K_5 realizes these values.

5 An $\widetilde{O}(r^{19m/100})$ Algorithm

The analysis of Algorithm A contains the seeds of its improvement. First, since reductions on vertices 5-vertices may destroy only 5 edges, we cannot ignore them and improve on $m/5$. This simply means including them in the LP.

Second, were this the only change we made, we would find that the LP solution is the same as before, with support on a reduction on a 4-vertex with all 4-neighbors (a "bad" reduction destroying only 4 edges), and harmless reductions (III-reduction on a 3-vertex and the I- and II-reductions it enables). This suggests that we should focus on eliminating the bad reduction. Indeed, if we exclude it from the LP, the LP cost decreases to 23/120 (about 0.192), and the new solution shows support on a reduction on a degree-5 vertex with all degree-5 neighbors and a degree-4 vertex with one degree-3 neighbor (each resulting in the destruction of 5 edges). If the first of these cases could also be eliminated, the LP would have cost 19/100, precisely what our algorithm will achieve. Improving beyond this would require addressing the remaining bad cases of a 5-vertex with neighbors of degree 5 except for one of degree 4, and a 4-vertex with neighbors of degree 4 except for one of degree 3.

Finally, a collection of many disjoint K_5s requires $m/5$ III-reductions in total. To beat $\widetilde{O}(r^{m/5})$ we will have to use the fact that an optimum solution to a disconnected CSP is a union of solutions of its components, and thus the $m/5$ reductions can in some sense be done in parallel, rather than sequentially. Correspondingly, where Algorithm A.1 built a *sequence* of reductions of length at most $m/5$, Algorithm B.1 will build a reduction *tree* whose *depth* is at most $2 + 19m/100$. The depth bound is proved by showing that in any sequence of reductions in a component on a fixed vertex, all but at most two "bad" reductions

can be paired with other reductions, and for the good reductions (including the paired ones), the LP has maximum 19/100.

Algorithm B: First Phase. As with Algorithm A, a first phase Algorithm B.1 of Algorithm B performs only graph reductions. Like Algorithm A, Algorithm B preferentially performs type 0, I or II reductions, but it is more particular about the vertices on which it III-reduces. When forced to perform a type III reduction, Algorithm B selects a vertex in the following decreasing order of preference:

- a vertex of degree ≥ 6;
- a vertex of degree 5 with at least 1 neighbor of degree 3 or 4;
- a vertex of degree 5 whose neighbors all have degree 5;
- a vertex of degree 4 with at least 1 neighbor of degree 3;
- a vertex of degree 4 whose neighbors all have degree 4;
- a vertex of degree 3.

When Algorithm B makes any such reduction with any degree-3 neighbor, it immediately follows up with II-reductions on all those neighbors.

Because Algorithm B treats graph components individually, the sequence of reductions must be organized into a *reduction tree*. The defining property of the reduction tree is that if reduction on a vertex v divides the graph into k components, then a corresponding tree node v has k children, one for each component, the child node corresponding to the first vertex reduced upon in that component (the first vertex in the reduction sequence restricted to the set of vertices in the component). If the graph is initially disconnected, the reduction "tree" is really a forest, but since this case presents no additional issues we will speak in terms of a tree. We remark that the number of children k is necessarily 1 for I- and II-reductions, can be 1 or more for a III-reduction, and is 0 for a 0-reduction.

Call the maximum number of III-reduction nodes in any root-to-leaf path in the reduction tree its "III-reduction" depth. Lemma 3 characterizes an efficient construction of the tree, but it is clear that it can be done in polynomial time and space. The crux of the matter is Lemma 4, which relies on the reduction preference order set forth above, but not on the algorithmic details of Algorithm B.1.

Lemma 3. *A reduction tree on n vertices which has III-reduction depth d can be constructed in time $O(dn + n)$ and space $O(m + n)$.*

Splitting-Tree Depth. Analogous to Lemma 1 characterizing Algorithm A, the next lemma is the heart of the analysis of Algorithm B.

Lemma 4. *For a graph G with m edges, the reduction tree's III-reduction depth is $d \leq 2 + 19m/100$.*

Proof. It suffices to prove the lemma for graphs with maximum degree ≤ 5. As in the proof of Lemma 1, for each type of reduction we will count the number of edges and vertices of various degrees it destroys, and its depth: the depth

is normally 1 for a III-reduction and 0 otherwise, but we will now introduce "paired" pseudo-reductions counting for depth 2. Recall that in Algorithm B we immediately follow each III-reduction with a II-reduction on each 2-vertex it produces.

Define a "bad" reduction to be one on a 5-vertex all of whose neighbors are also of degree 5, or on a 4-vertex all of whose neighbors are of degree 4. (These two reductions destroy 5 and 4 edges respectively, while, except for reducing on a 4-vertex with three 4-neighbors and one 3-neighbor, every other reduction, coupled with the II-reductions it enables, destroys at least 6 edges.) The analysis is aimed at controlling the number of these reductions.

For shorthand, we write reductions in terms of the degree of the vertex on which we are reducing followed by the numbers of neighbors of degrees 5, 4, and 3, so for example the "bad" reduction on a 5-vertex is written (5|500).

Within a component, a (5|500) reduction is performed only if there is no 5-vertex adjacent to a 3- or 4-vertex; this means the component *has* no 3- or 4-vertices, since otherwise a path from such a vertex to the 5-vertex would include an edge incident on a 5-vertex and a 3- or 4-vertex. We track the component containing one vertex, say vertex 1, as it is reduced. If the component necessitates a bad 5-reduction, one of four things must be true:

1. *This is the first degree-5 reduction in this branch of the splitting tree.* This case can occur only once. Weakening this constraint, we will allow it to occur any number of times, but we will count its depth contribution as 0, and add 1 to the depth at the end. For this reason, the first bold row in Table 2 has depth 0 not 1.
2. *The previous III-reduction (which because of our preference order must also have been a degree-5 reduction) was on a* (5|005) *vertex, and left no vertices of degree 3 or 4.* In this case we pair the bad (5|500) reduction with its preceding (5|005) reduction. This defines a new "pair" pseudo-reduction shown as the second bold row of the table: it counts for 2 steps, destroys 15 edges, etc. (Other, non-paired (5|005) reductions are still allowed as before.)
3. *The previous III-reduction was on a 5-vertex and produced vertices of degree 3 or 4 in this component, but they were destroyed by I- and II-reductions.* In this case we similarly pair the (5|500) reduction with a I- or II-reduction, but we cannot say specifically with which sort. The "forces" column of Table 2 will constrain each (5|500) reduction for this case to be accompanied by at least one I- or II-reduction (or two "half-edge" reductions) of any sort.
4. *The previous III-reduction was on a 5-vertex and produced vertices of degree 3 or 4, but split them all off into other components.* In this case, the (5|500) reduction produces a non-empty side component destroyed with the usual reductions but adding depth 0 for the component of interest. These reductions can be expressed as a nonnegative combination of half-edge reductions, so we can pair the (5|500) reduction with any two of these, much as in case (3).

Table 2 summarizes the reductions. Together, the four cases above let us exclude (5|500) reductions, replacing them with less harmful possibilities represented by the first three bold rows in the table. Reasoning identically for bad

Table 2. Tabulation of the effects of various reductions in Algorithm B

deg	#nbrs of deg					destroys					forces	depth
	5	4	3	2	1	e	4	3	2	1		
5	0	0	5	0	0	10	0	5	0	0	0	1
5	0	1	4	0	0	9	1	3	0	0	0	1
.	.	.	.	.	.	.	.	.	.	.	.	.
5	4	1	0	0	0	5	−3	−1	0	0	0	1
5	**5**	**0**	**0**	**0**	**0**	**5**	**−5**	**0**	**0**	**0**	**0**	**0**
5 + 5	**5**	**0**	**5**	**0**	**0**	**15**	**−5**	**5**	**0**	**0**	**0**	**2**
5	**5**	**0**	**0**	**0**	**0**	**5**	**−5**	**0**	**0**	**0**	**−1**	**1**
4	0	0	4	0	0	8	1	4	0	0	0	1
4	0	1	3	0	0	7	2	2	0	0	0	1
4	0	2	2	0	0	6	3	0	0	0	0	1
4	0	3	1	0	0	5	4	−2	0	0	0	1
4	**0**	**4**	**0**	**0**	**0**	**4**	**5**	**−4**	**0**	**0**	**0**	**0**
4 + 4	**0**	**4**	**4**	**0**	**0**	**12**	**6**	**0**	**0**	**0**	**0**	**2**
4	**0**	**4**	**0**	**0**	**0**	**4**	**5**	**−4**	**0**	**0**	**−1**	**1**
3	0	0	3	0	0	6	0	4	0	0	0	1
2	0	0	0	0	0	1	0	0	1	0	1	0
$\frac{1}{2}\,e$	1	0	0	0	0	$\frac{1}{2}$	−1	0	0	0	$\frac{1}{2}$	0
$\frac{1}{2}\,e$	0	1	0	0	0	$\frac{1}{2}$	1	−1	0	0	$\frac{1}{2}$	0
$\frac{1}{2}\,e$	0	0	1	0	0	$\frac{1}{2}$	0	1	−1	0	$\frac{1}{2}$	0
$\frac{1}{2}\,e$	0	0	0	1	0	$\frac{1}{2}$	0	0	1	−1	$\frac{1}{2}$	0
$\frac{1}{2}\,e$	0	0	0	0	1	$\frac{1}{2}$	0	0	0	1	$\frac{1}{2}$	0

reductions on 4-vertices contributes the other three bold rows. In analyzing a branch of the splitting tree, let vector $\boldsymbol{n}'$ count the (normalized) number of reductions of each type, as in the proof of Lemma 1. Constraining the "forces" column's dot product with $\boldsymbol{n}'$ forces the pairing of a I- or II-reduction with each bad reduction. Everything else goes as before. The LP maximum is 19/100, which (accounting for case (1) occurrences for a 4- and a 5-vertex) proves the reduction depth to be $\leq 2 + 19m/100$.

Corollary 5. *Algorithm B solves a Max* $(r, 2)$*-CSP instance* (G, S)*, where* G *has* n *vertices and* m *edges, in time* $O(nr^{5+19m/100})$ *and in linear space.*

Motivated by a tree-decomposition CSP approach in a recent report by Kneis and Rossmanith [KR05], we note the following corollary of Lemma 4.

Corollary 6. *A graph* G *with* m *edges has treewidth at most* $3 + 19m/100$*.*

6 Conclusions

The LP is key to our algorithm design as well as the analysis. We begin with a collection of reductions, and a preference order on them, guided by intuition. The preference order both excludes some cases (e.g., reducing on high-degree vertices

first, we do not need to worry about a reduction vertex having a neighbor of larger degree) and determines an LP. Solving the LP pinpoints the "bad" reductions that determine the bound. We then try to ameliorate these cases: in the present paper we showed that each could be paired with another reduction to give a less bad combined reduction, but we might also have taken some other course such as changing the preference order to eliminate bad reductions. Using the LP as a black box is a convenient way to engage in this cycle of algorithm analysis and improvement, an approach that should be applicable to other problems.

Our methods seem not to extend to 3-variable CSPs, since a II-reduction would combine two 3-variable clauses into a 4-variable clause.

The improvement from $m/5$ to $19m/100$ is significant in that $m/6$ appears to be a natural barrier: In a random cubic graph, a III-reduction results in the deletion of 6 edges and a new cubic graph, and to beat $m/6$ requires either distinguishing the new graph from random cubic, or targeting many III-reductions so as to divide the graph into components. Such an approach would require new ideas outside the scope of the local properties we consider.

References

[BE05] Richard Beigel and David Eppstein, *3-coloring in time $O(1.3289^n)$*, J. Algorithms **54** (2005), no. 2, 168–204.

[GHNR03] Jens Gramm, Edward A. Hirsch, Rolf Niedermeier, and Peter Rossmanith, *Worst-case upper bounds for MAX-2-SAT with an application to MAX-CUT*, Discrete Appl. Math. **130** (2003), no. 2, 139–155.

[KF02] A. S. Kulikov and S. S. Fedin, *Solution of the maximum cut problem in time $2^{|E|/4}$*, Zap. Nauchn. Sem. S.-Peterburg. Otdel. Mat. Inst. Steklov. (POMI) **293** (2002), no. Teor. Slozhn. Vychisl. 7, 129–138, 183.

[KR05] Joachim Kneis and Peter Rossmanith, *A new satisfiability algorithm with applications to Max-Cut*, Tech. Report AIB-2005-08, Department of Computer Science, RWTH Aachen, 2005.

[SS03] Alexander D. Scott and Gregory B. Sorkin, *Faster algorithms for MAX CUT and MAX CSP, with polynomial expected time for sparse instances*, Proc. 7th International Workshop on Randomization and Approximation Techniques in Computer Science, RANDOM 2003, Lecture Notes in Comput. Sci., vol. 2764, Springer, August 2003, pp. 382–395.

[SS04] ———, *A faster exponential-time algorithm for Max 2-Sat, Max Cut, and Max k-Cut*, Tech. Report RC23456 (W0412-001), IBM Research Report, December 2004, See http://domino.research.ibm.com/library/cyberdig.nsf.

[SS06] ———, *Generalized constraint satisfaction problems*, Tech. Report RC23935, IBM Research Report, April 2006, See http://domino.research.ibm.com/library/cyberdig.nsf.

[TSSW00] Luca Trevisan, Gregory B. Sorkin, Madhu Sudan, and David P. Williamson, *Gadgets, approximation, and linear programming*, SIAM J. Comput. **29** (2000), no. 6, 2074–2097.

[Wil04] Ryan Williams, *A new algorithm for optimal constraint satisfaction and its implications*, Proc. 31st International Colloquium on Automata, Languages and Programming (ICALP), 2004.

Approximate k-Steiner Forests Via the Lagrangian Relaxation Technique with Internal Preprocessing*

Danny Segev[1] and Gil Segev[2]

[1] School of Mathematical Sciences, Tel-Aviv University, Tel-Aviv 69978, Israel
segevd@post.tau.ac.il
[2] Department of Computer Science and Applied Mathematics
Weizmann Institute of Science, Rehovot 76100, Israel
gil.segev@weizmann.ac.il

Abstract. An instance of the *k-Steiner forest* problem consists of an undirected graph $G = (V, E)$, the edges of which are associated with non-negative costs, and a collection $\mathcal{D} = \{(s_i, t_i) : 1 \leq i \leq d\}$ of distinct pairs of vertices, interchangeably referred to as *demands*. We say that a forest $\mathcal{F} \subseteq G$ *connects* a demand (s_i, t_i) when it contains an s_i-t_i path. Given a requirement parameter $k \leq |\mathcal{D}|$, the goal is to find a minimum cost forest that connects at least k demands in $\mathcal{D}$. This problem has recently been studied by Hajiaghayi and Jain [SODA '06], whose main contribution in this context was to relate the inapproximability of k-Steiner forest to that of the *dense k-subgraph* problem. However, Hajiaghayi and Jain did not provide any algorithmic result for the respective settings, and posed this objective as an important direction for future research.

In this paper, we present the first non-trivial approximation algorithm for the k-Steiner forest problem, which is based on a novel extension of the Lagrangian relaxation technique. Specifically, our algorithm constructs a feasible forest whose cost is within a factor of $O(\min\{n^{2/3}, \sqrt{d}\} \cdot \log d)$ of optimal, where n is the number of vertices in the input graph and d is the number of demands.

1 Introduction

An instance of the *k-Steiner forest* problem consists of an undirected graph $G = (V, E)$, whose edges are associated with non-negative costs specified by a real-valued function $c : E \to \mathbb{R}_+$. An additional ingredient of the input is a collection $\mathcal{D} = \{(s_i, t_i) : 1 \leq i \leq d\}$ of distinct pairs of vertices, interchangeably referred to as *demands*. In adherence to standard terminology, we say that a forest $\mathcal{F} \subseteq G$ *connects* a demand (s_i, t_i) when it contains an s_i-t_i path. Given a requirement parameter k, the objective is to find a minimum cost forest that connects at least k demands in $\mathcal{D}$. It is important to note that there is no loss of generality in restricting the discussion to forests, rather than allowing arbitrary subgraphs, as any edge-minimal solution to the problem under consideration is necessarily acyclic.

* Due to space limitations, some proofs and technical details are omitted from this extended abstract. We refer the reader to the full version of this paper (currently available online at http://www.math.tau.ac.il/~segevd), in which all missing information is provided.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 600–611, 2006.

The k-Steiner forest problem has recently been introduced and studied by Hajiaghayi and Jain [12], who pointed out that both *Steiner forest* and *k-MST* can be interpreted as special cases of this problem, implying its APX-hardness [3, 15]. Their main contribution in this context is to relate the inapproximability of k-Steiner forest to that of the *dense k-subgraph* problem, in which given an undirected graph we wish to identify a subset of k vertices whose induced subgraph has a maximum number of edges. Specifically, this relation states that a polynomial-time $\alpha(n)$-approximation for k-Steiner forest *on stars* can be employed as a subroutine to efficiently find a k-vertex subgraph whose density is at least $\frac{1}{2}\alpha^{-2}(n)$ times that of an optimal solution. We remark that the currently best approximation guarantee for the dense k-subgraph problem is $O(n^{-\delta})$, for some universal constant $\delta < 1/3$, due to Feige, Kortsarz and Peleg [7]; this long-standing bound will be immediately improved as a consequence of achieving an $o(n^{\delta/2})$ factor for k-Steiner forest. Hajiaghayi and Jain [12] did not provide any algorithmic result for the latter, and posed this objective as an important open problem for future research.

1.1 Results and Techniques

In this paper, we present the first non-trivial approximation algorithm for the k-Steiner forest problem, which is based on a novel extension of the Lagrangian relaxation technique. Our main result is the following.

Theorem 1. *There is a polynomial-time algorithm that approximates the k-Steiner forest problem to within a factor of* $O(\min\{n^{2/3}, \sqrt{d}\} \cdot \log d)$*, where n is the number of vertices in the given graph and d is the number of demands.*

A slightly different view. The algorithm we propose and its analysis are based on viewing k-Steiner forest as a partial covering problem with exponentially many "sets". For this purpose, let $\mathcal{T}$ denote the collection of all trees in the input graph G. Then, one can think of the k-Steiner forest problem as that of computing a minimum cost subset of trees $\mathcal{F} \subseteq \mathcal{T}$ that connects at least k demands in $\mathcal{D}$. Although $\mathcal{F}$ is clearly a forest in any optimal solution, it would be imperative to allow the trees in this subset to overlap in both vertices and edges; therefore, $\mathcal{F}$ will be referred to as a *collection of trees* rather than as a forest.

A seemingly useful method. Suppose that the requirement to connect at least k demands is not strictly enforced; instead, if the collection of trees we construct leaves a demand (s_i, t_i) unconnected, we incur a penalty of $\pi(i)$. The *prize-collecting Steiner forest* problem asks to find a collection $\mathcal{F} \subseteq \mathcal{T}$ that minimizes the cost of $\mathcal{F}$ plus the penalties of the unconnected demands. As we demonstrate in the full version of this paper, connections between the prize-collecting and the partial variants of numerous optimization problems have been the subject of an ever-growing line of work, in which the Lagrangian relaxation technique plays an instrumental role. Schematically speaking, this technique assembles a near-optimal solution to the partial variant by employing successive calls to an approximation algorithm for the prize-collecting variant. However, in all previous applications the latter algorithm had to satisfy two structural properties, stated here in terms of prize-collecting Steiner forest:

1. **Pay penalties at the same rate as OPT.** For every instance I, the solution we obtain satisfies $C + \alpha\Pi \leq \alpha \cdot \mathrm{OPT}(I)$, where C is the total cost of the trees picked by the algorithm, and Π is the sum of penalties over all unconnected demands. Intuitively, an inequality of this form guarantees an α-approximation even when all penalties are inflated by a factor of α.
2. **Allow solutions to be combined.** Lagrangian duality, in conjunction with the first property we mention, establishes that any optimal solution to the original k-Steiner forest problem can be approximated by a *convex combination* of prize-collecting solutions. Nevertheless, such a characterization does not appear to be of much help, unless there is an efficient method for combining these solutions into an approximate integral one.

Once again, existing algorithms are not applicable. It is not difficult to verify that the LP-rounding technique suggested by Bienstock, Goemans, Simchi-Levi and Williamson [5] can be adapted to approximate prize-collecting Steiner forest to within a factor of 3. In fact, Hajiaghayi and Jain [12] have recently proposed a primal-dual algorithm for this problem that achieves a similar approximation guarantee, and have also derived an improved factor of 2.54 by means of randomized rounding. Unfortunately, penalties are not paid-for at the same rate as OPT by any of these algorithms. Furthermore, we argue that this difficulty is not the primary factor limiting the applicability of previously advocated methods; rather, the fundamental question is: How do we combine the prize-collecting solutions?

It is worth noting that, regardless of the problem-specific scheme we may apply to combine these solutions, most algorithms that follow the Lagrangian relaxation framework acquire an additional lower bound on the optimal cost through *preprocessing*. Specifically, an exhaustive search is conducted in order to "guess" certain attributes of an arbitrary optimal solution, according to which the given instance is modified in advance. Examples for attributes that were found to be useful in approximating directly related problems include the optimal diameter [9], a constant number of edges in the optimal solution [14, 16], or a combination of both vertices and edges [2].

"Discarding" expensive trees via internal preprocessing. Intuitively, the k-Steiner forest problem would be much easier to approximate, given that our prize-collecting algorithm avoids picking overly-priced trees. However, we are not aware of any way of achieving this objective by utilizing the above-mentioned form of preprocessing. The new approach we propose does not involve a preliminary step of preprocessing; instead, an analogous effect is obtained by adding extra requirements to internal procedures. To clarify this statement, suppose that $\Delta \geq 0$ estimates the minimum cost of a k-Steiner forest to within some constant factor. Then, we would like the prize-collecting algorithm to behave as if all trees with cost greater than Δ were explicitly eliminated from $\mathcal{T}$, a hypothetical scenario whose optimal cost is denoted by OPT_Δ. In Section 2, we show that this task can be accomplished in bicriteria fashion, establishing the next theorem.

Theorem 2. *There is a polynomial-time algorithm that finds a collection $\mathcal{F} \subseteq \mathcal{T}$, consisting of trees with individual costs of at most $4\min\{n^{2/3}, \sqrt{d}\}\Delta$, such that*

$$C(\mathcal{F}) + 12\min\left\{n^{2/3}, \sqrt{d}\right\}\mathcal{H}(d) \cdot \Pi(\mathcal{F}) \leq 12\min\left\{n^{2/3}, \sqrt{d}\right\}\mathcal{H}(d) \cdot \mathrm{OPT}_\Delta \ .$$

Here, $C(\mathcal{F})$ is the total cost of the trees in $\mathcal{F}$, $\Pi(\mathcal{F})$ is the sum of penalties over all demands left unconnected by $\mathcal{F}$, and $\mathcal{H}(d)$ is the d-th harmonic number.

Putting it all together. In Section 3, we formulate the k-Steiner forest problem as an integer program, and augment it with additional valid constraints stating that trees in $\mathcal{T}$ with a cost greater than Δ cannot be picked. While these constraints are clearly redundant with respect to the original problem, they play an important role in its Lagrangian relaxation, by enabling us to make use of the unorthodox algorithm described in Theorem 2. Consequently, the prize-collecting solutions we construct pay penalties at an optimal rate, and at the same time consist of trees whose cost can be bounded in terms of Δ. The strategy we apply to combine these solutions has its roots in a greedy procedure suggested by Levin and Segev [14] for partially covering general set systems.

1.2 Related Work

We proceed by demonstrating that k-Steiner forest generalizes and unifies two of the most fundamental problems in combinatorial optimization. Noting that the undermentioned problems have received a great deal of attention in the operations research and computer science communities, it is beyond the scope of this writing to present an exhaustive overview. We refer the reader to the full version of this paper for a more comprehensive review of the literature.

When $k = |\mathcal{D}|$, we obtain the Steiner forest problem, in which the goal is to compute a minimum cost forest connecting all given demands. This problem is known to be APX-hard [3, 15], since it contains *Steiner tree* as a special case. On the positive side, Agrawal, Klein and Ravi [1] devised the currently best approximation algorithm, achieving a performance guarantee of $2(1 - 1/d)$. This result was extended to a broader class of network design problems by Goemans and Williamson [11].

Now suppose that the set of demands is $\mathcal{D} = \{(r, v) : v \in V\}$, where r is some specified vertex. In this case, k-Steiner forest captures the *rooted k-MST* problem, asking to find a minimum cost tree that spans at least k vertices, one of which is r. We remark that this version of the problem is equivalent to its classic version, in which no root vertex is specified (see, for example, [10]). Following a sequence of initial results, Blum, Ravi and Vempala [6] were the first to obtain a constant-factor approximation for k-MST. This factor was improved to 3 by Garg [9], later to $2 + \epsilon$ by Arora and Karakostas [2], and finally to 2 by Garg [10].

As previously mentioned, the approximability of k-Steiner forest is closely related to that of the dense k-subgraph problem. The currently best approximation guarantee for the latter problem is $O(n^{-\delta})$, for some universal constant $\delta < 1/3$, due to Feige, Kortsarz and Peleg [7]. Additional approaches whose performance depends on the ratio k/n have emerged over the years, for example, a greedy heuristic proposed by Asahiro, Iwama, Tamaki and Tokuyama [4], and SDP-based algorithms developed by Feige and Langberg [8] and by Han, Ye and Zhang [13].

1.3 Notation

We conclude this section by introducing some notation and terminology. Given a tree $T \in \mathcal{T}$, we use $c(T) = \sum_{e \in E(T)} c(e)$ to denote the total cost of the edges in T.

Furthermore, when $\mathcal{F}$ is a collection of trees, the notation $c(\mathcal{F})$ is used as a shorthand for $\sum_{T\in\mathcal{F}} c(T)$. Finally, for a collection $\mathcal{F} \subseteq \mathcal{T}$, we denote by $\mathcal{D}(\mathcal{F})$ the set of demands connected by at least one tree in $\mathcal{F}$, excluding the case where $\mathcal{F}$ consists of a single tree, which is abbreviated by writing $\mathcal{D}(T)$ instead of $\mathcal{D}(\{T\})$.

2 A Bicriteria Prize-Collecting Algorithm

The main result of this section is a constructive proof of Theorem 2. We remind the reader that a prize-collecting instance consists of an undirected graph $G = (V, E)$ with non-negative edge costs specified by $c : E \to \mathbb{R}_+$. An additional ingredient of the input is a collection of demands $\mathcal{D} = \{(s_i, t_i) : 1 \leq i \leq d\}$, where the penalty we incur for leaving a demand (s_i, t_i) unconnected is $\pi(i)$. Now suppose that the individual cost of every tree in the constructed solution should not exceed Δ, a given budget; we denote by OPT_Δ the cost of an optimal solution satisfying this extra restriction.

2.1 Constructing Dense Trees with Small Costs

In what follows, we examine the intrinsic structure of connectivity under budget constraints, and develop an essential tool that will allow us to considerably simplify the proof of Theorem 2. For this purpose, we define the *density* of a tree $T \in \mathcal{T}$ to be the ratio between its cost and the number of demands it connects, and let $\mathcal{T}_\Delta$ denote the collection of trees in $\mathcal{T}$ whose cost is at most Δ. Having introduced this notation, we claim that the minimum density of a tree in $\mathcal{T}_\Delta$ can be efficiently approximated, while keeping the factor by which the budget Δ is exceeded within an acceptable magnitude. This result is formally described in the next theorem.

Theorem 3. *There is a polynomial-time algorithm that finds a tree $T \in \mathcal{T}$ whose density is at most $12\min\{n^{2/3}, \sqrt{d}\}$ times the minimum density of a tree in $\mathcal{T}_\Delta$, ensuring that $c(T) \leq 4\min\{n^{2/3}, \sqrt{d}\}\Delta$ at the same time.*

Due to space limitations, we prove a simplified version of the above theorem, in which the term $\min\{n^{2/3}, \sqrt{d}\}$ is replaced by $n^{2/3}$. Following similar arguments, an analogous proof for the $\sqrt{d}$-dependent bound is provided in the full version of this paper. For sake of simplicity, we do not attempt to optimize constant factors in this extended abstract.

Initial assumptions. To avoid special treatment of degenerate cases, it would be convenient to assume throughout this section that the input graph contains at least one tree with $c(T) \leq \Delta$ and $\mathcal{D}(T) \neq \emptyset$. As explained in Section 3, this requirement can be easily enforced. We therefore consider the case where $\min_{T\in\mathcal{T}_\Delta} \mathrm{density}(T) < \infty$, let T^* be a tree of minimum density over all trees in $\mathcal{T}_\Delta$, and let $q = |\mathcal{D}(T^*)|$. By conducting an exhaustive search, we may also assume that the number of connected demands q and an arbitrary vertex $r \in V(T^*)$ are known in advance.

The vertex-augmentation lemma. Noting that the demands in $\mathcal{D}$ are distinct, T^* must be comprised of many vertices whenever q is sufficiently large. By combining this intuitive observation with further structural properties, we demonstrate in Lemma 4 how to extend a given tree to a new tree that connects $\Omega(q)$ demands or contains $\Omega(\sqrt{q})$

additional vertices. To bound the overall cost of the augmenting edges, our approach depends upon a constant-factor approximation for the rooted *quota-MST* problem. In this generalization of k-MST, each vertex $v \in V$ is associated with a non-negative profit $p(v)$, and the objective is to compute a minimum cost tree rooted at r that collects a total profit of at least P, a specified quota. It is easy to ascertain that the proof of Lemma 4 requires to approximate instances with $p(v) \in \{0, \ldots, d\}$, a subproblem that reduces back to k-MST in a straightforward way. Consequently, such instances can be approximated to within a factor of 2 [10].

Lemma 4. *Let T be a tree that contains r. Then, we can find in polynomial time a tree T^+ satisfying $T \subseteq T^+$, $c(T^+) \leq c(T) + 2c(T^*)$ and at least one of the following properties:*

1. $|V(T^+)| \geq |V(T)| + \frac{\sqrt{q}}{2}$.
2. $|\mathcal{D}(T^+)| \geq \frac{3q}{8}$.

Proof. We assume without loss of generality that $|\mathcal{D}(T)| < q/2$, since the claim can be established in the opposite case by defining $T^+ = T$. For $0 \leq j \leq 2$, let A_j be the set of demands in $\mathcal{D}(T^*)$ with exactly j endpoints in $V(T)$, that is, $A_j = \{(s_i, t_i) \in \mathcal{D}(T^*) : |\{s_i, t_i\} \cap V(T)| = j\}$. The proof proceeds by considering two cases, depending on the cardinality of $V(T^*) \setminus V(T)$.

Case 1: $|V(T^*) \setminus V(T)| \geq \sqrt{q}/2$. This inequality implies, in particular, that T^* connects r to at least $\sqrt{q}/2$ vertices not belonging to T. Hence, we can obtain a tree $\tilde{T}$ that connects r to at least $\sqrt{q}/2$ vertices in $V \setminus V(T)$ and satisfies $c(\tilde{T}) \leq 2c(T^*)$ by approximating the following quota-MST instance: The vertices in $V \setminus V(T)$ are associated with unit profits, whereas those in $V(T)$ have zero profits; the quota is $\sqrt{q}/2$; and the root is r. We now define $T^+ = T \cup \tilde{T}$, and eliminate cycles in T^+ by removing edges from $\tilde{T}$. Clearly, $|V(T^+)| \geq |V(T)| + \sqrt{q}/2$.

Case 2: $|V(T^*) \setminus V(T)| < \sqrt{q}/2$. Since the demands in $\mathcal{D}$ are distinct, we have

$$|A_0| \leq \binom{|V(T^*) \setminus V(T)|}{2} \leq \binom{\lfloor \sqrt{q}/2 \rfloor}{2} \leq \frac{q}{8} \ ,$$

implying that

$$|A_1| = |\mathcal{D}(T^*)| - |A_0| - |A_2| \geq |\mathcal{D}(T^*)| - |A_0| - |\mathcal{D}(T)| \geq q - \frac{q}{8} - \frac{q}{2} = \frac{3q}{8} \ ,$$

where the first equation holds since $\{A_0, A_1, A_2\}$ is a partition of $\mathcal{D}(T^*)$, and the succeeding inequality is obtained by observing that $A_2 \subseteq \mathcal{D}(T)$. At this point, we approximate the following quota-MST instance: The profit $p(v)$ of each vertex $v \in V \setminus V(T)$ is set to be the number of demands in $\mathcal{D}$ consisting of v and an additional vertex from T; all vertices in $V(T)$ have zero profits; the quota is $3q/8$; and the root is r. As a result, we acquire a tree $\tilde{T}$ satisfying $c(\tilde{T}) \leq 2c(T^*)$, since T^* connects r to the vertex set $V(T^*) \setminus V(T)$, with $\sum_{v \in V(T^*) \setminus V(T)} p(v) \geq |A_1| \geq 3q/8$. Once again, we designate $T^+ = T \cup \tilde{T}$ and eliminate cycles in T^+, noting that $|\mathcal{D}(T^+)| \geq 3q/8$.

Needless to say, $|V(T^*) \setminus V(T)|$ and $\sqrt{q}/2$ cannot be compared without prior knowledge of T^*. To work around this difficulty, we try to approximate both quota-MST instances, whose construction is independent of T^*. If one of these attempts fails to generate a feasible solution, we can immediately distinguish between the pair of cases described above; otherwise, we pick the case in which $c(\tilde{T})$ is smaller. □

Finding a budgeted dense tree. A close inspection of Lemma 4 reveals that repeated applications of the algorithm it prescribes will terminate rather quickly with a tree connecting $\Omega(q)$ demands, provided that q is sufficiently large. Moreover, as each augmentation step increases the overall cost by at most $2c(T^*) \leq 2\Delta$, the resulting tree would be of near-optimal density, and its cost would not exceed the budget Δ by much. This observation suggests two separate tactics, depending on the order of q.

Case 1: $q < 9n^{2/3}$. Interpreting $c : E \to \mathbb{R}_+$ as a length function, we compute the shortest path P connecting any demand in $\mathcal{D}$. Note that the cost of this solution does not exceed Δ, since T^* connects at least one demand and $c(T^*) \leq \Delta$. In addition,

$$\text{density}(P) = \frac{c(P)}{|\mathcal{D}(P)|} \leq c(T^*) \leq 9n^{2/3} \cdot \frac{c(T^*)}{q} = 9n^{2/3} \cdot \frac{c(T^*)}{|\mathcal{D}(T^*)|} = 9n^{2/3} \cdot \text{density}(T^*) \ .$$

Case 2: $q \geq 9n^{2/3}$. Starting with a trivial tree T that consists of the singular vertex r, we repeatedly extend T by applying the algorithm proposed in Lemma 4, as long as $|\mathcal{D}(T)| < 3q/8$. In each step, we either add to T at least $\sqrt{q}/2$ new vertices, or discover that it already connects at least $3q/8$ demands. It follows that the resulting tree satisfies

$$c(T) \leq \left(\frac{n}{\sqrt{q}/2} + 1\right) \cdot 2c(T^*) \leq \left(\frac{2n^{2/3}}{3} + 1\right) \cdot 2c(T^*) \leq 4n^{2/3}c(T^*) \leq 4n^{2/3}\Delta \ ,$$

and at the same time

$$\text{density}(T) = \frac{c(T)}{|\mathcal{D}(T)|} \leq \frac{4n^{2/3}c(T^*)}{3q/8} \leq 11n^{2/3} \cdot \frac{c(T^*)}{|\mathcal{D}(T^*)|} = 11n^{2/3} \cdot \text{density}(T^*) \ .$$

2.2 A Greedy Prize-Collecting Approach

We are now ready to conclude the proof of Theorem 2, by enclosing the algorithm for constructing budgeted dense trees within a sensible greedy heuristic. The principal idea that guides our algorithm can be informally described as follows. In each step, we identify a tree $T \in \mathcal{T}$ of near-optimal density, whose cost does not significantly exceed Δ. However, rather than picking T right away, its density is compared to the minimum available penalty $\pi(i^*)$, scaled by some factor that will be specified later. Based on the outcome of this comparison, we decide whether to pick T or to tentatively pay the penalty $\pi(i^*)$ and leave the corresponding demand (s_{i^*}, t_{i^*}) unconnected. In an attempt to highlight the intimate relationship between Theorems 2 and 3, let $\alpha(n, d)$ be the approximation guarantee of the latter theorem with respect to the optimal density, and let $\beta(n, d)$ be the maximal factor by which the budget Δ is exceeded. As previously mentioned, $\alpha(n, d) = 12 \min\{n^{2/3}, \sqrt{d}\}$ and $\beta(n, d) = 4 \min\{n^{2/3}, \sqrt{d}\}$.

The algorithm. In what follows, $\mathcal{F}$ denotes the collection of trees we construct, while $\mathcal{R}$ denotes the set of remaining demands; $\mathcal{P}$ and price$(\cdot)$ are used for purposes of analysis.

1. Initialize $\mathcal{F} \leftarrow \emptyset$, $\mathcal{R} \leftarrow \mathcal{D}$ and $\mathcal{P} \leftarrow \emptyset$.
2. While $\mathcal{R} \neq \emptyset$
   (a) Apply the algorithm given in Theorem 3 to identify a tree $T \in \mathcal{T}$ that approximates the following instance: The underlying graph and edge costs are still $G = (V, E)$ and $c : E \rightarrow \mathbb{R}_+$, respectively; the collection of demands is $\mathcal{R}$; and the budget is Δ.
   (b) Let (s_{i^*}, t_{i^*}) be a demand that minimizes $\pi(i)$ over all demands in $\mathcal{R}$, breaking ties arbitrarily.
   (c) If density$(T) \leq \alpha(n,d)\mathcal{H}(d)\pi(i^*)$, add T to $\mathcal{F}$, eliminate from $\mathcal{R}$ all newly connected demands, and for each of them define price(i) = density(T). Otherwise, add i^* to $\mathcal{P}$, eliminate (s_{i^*}, t_{i^*}) from $\mathcal{R}$, and define price$(i^*) = \alpha(n,d)\mathcal{H}(d)\pi(i^*)$.
3. Return $\mathcal{F}$.

Analysis. We first argue that the collection $\mathcal{F}$ consists of trees with individual costs of at most $\beta(n,d)\Delta$. This follows from the observation that each of these trees was obtained during step 2a, implying that $c(T) \leq \beta(n, |\mathcal{R}|)\Delta \leq \beta(n,d)\Delta$ for every $T \in \mathcal{F}$, according to Theorem 3. In addition, the overall cost of the trees in $\mathcal{F}$ is $\sum_{T\in\mathcal{F}} c(T)$, whereas the demands left unconnected by $\mathcal{F}$ have a total penalty of at most $\sum_{i\in\mathcal{P}} \pi(i)$. We remark that the latter term is an upper bound on the sum of penalties, and not the exact sum, since it is quite possible that $\mathcal{F}$ connects one or more demands in $\{(s_i, t_i) : i \in \mathcal{P}\}$. Therefore, we can complete the proof of Theorem 2 by verifying that

$$\sum_{T\in\mathcal{F}} c(T) + \alpha(n,d)\mathcal{H}(d) \sum_{i\in\mathcal{P}} \pi(i) \leq \alpha(n,d)\mathcal{H}(d) \cdot \mathrm{OPT}_\Delta \ . \tag{2.1}$$

Let $\mathcal{F}^* \subseteq \mathcal{T}_\Delta$ be an optimal solution to the prize-collecting instance at hand, and let Q be the index set of demands not connected by $\mathcal{F}^*$, that is, $Q = \{i : (s_i, t_i) \notin \mathcal{D}(\mathcal{F}^*)\}$. Note that, by construction of Q, we have $\sum_{T\in\mathcal{F}^*} c(T) + \sum_{i\in Q} \pi(i) = \mathrm{OPT}_\Delta$. Bearing these definitions in mind, each index $i \notin Q$ is assigned to a tree in $\mathcal{F}^*$ that connects (s_i, t_i), making an arbitrary choice in case of multiple options. In the remainder of this section, we use $\phi : \{1, \ldots, d\} \setminus Q \rightarrow \mathcal{F}^*$ to denote the resulting assignment and $\phi^{-1}(T)$ to denote the inverse image of T under ϕ. We proceed by establishing two crucial properties of the suggested pricing method.

Lemma 5. $\sum_{i\in\phi^{-1}(T)} \mathrm{price}(i) \leq \alpha(n,d)\mathcal{H}(d)c(T)$ *for every* $T \in \mathcal{F}^*$.

Lemma 6. $\mathrm{price}(i) \leq \alpha(n,d)\mathcal{H}(d)\pi(i)$ *for every* $1 \leq i \leq d$.

Noting that our pricing method guarantees $\sum_{i\in\mathcal{P}} \mathrm{price}(i) = \sum_{i\in\mathcal{P}} \alpha(n,d)\mathcal{H}(d)\pi(i)$ and $\sum_{i\notin\mathcal{P}} \mathrm{price}(i) = \sum_{T\in\mathcal{F}} c(T)$, we derive the desired bound (2.1) by manipulating Lemmas 5 and 6:

$$\begin{aligned}
\sum_{T\in\mathcal{F}} c(T) + \alpha(n,d)\mathcal{H}(d) \sum_{i\in\mathcal{P}} \pi(i) &= \sum_{i\notin\mathcal{P}} \mathrm{price}(i) + \sum_{i\in\mathcal{P}} \mathrm{price}(i) = \sum_{i\notin Q} \mathrm{price}(i) + \sum_{i\in Q} \mathrm{price}(i) \\
&= \sum_{T\in\mathcal{F}^*} \sum_{i\in\phi^{-1}(T)} \mathrm{price}(i) + \sum_{i\in Q} \mathrm{price}(i) \leq \alpha(n,d)\mathcal{H}(d) \sum_{T\in\mathcal{F}^*} c(T) + \alpha(n,d)\mathcal{H}(d) \sum_{i\in Q} \pi(i) \\
&= \alpha(n,d)\mathcal{H}(d) \cdot \mathrm{OPT}_\Delta \ .
\end{aligned}$$

3 The k-Steiner Forest Algorithm

Having already laid the foundations of our approach, we turn to describe the main result of this paper, namely, a polynomial-time algorithm that approximates the k-Steiner forest problem to within a factor of $O(\min\{n^{2/3}, \sqrt{d}\} \cdot \log d)$. Given an instance of the problem under consideration, we use OPT to denote the minimum cost of a forest that connects at least k demands. By conducting an exhaustive search, we may assume that a constant-factor estimate $\Delta \in [\text{OPT}, 2 \cdot \text{OPT}]$ of the optimal cost is known in advance. Furthermore, straightforward arguments allow us to assume that each edge has a strictly positive cost and that the shortest path connecting any demand is of length at most Δ.

3.1 An Integer Program and Its Lagrangian Relaxation

As mentioned earlier, the algorithm we propose and its analysis are based on viewing k-Steiner forest as an exponential-size partial covering problem, with the objective of computing a minimum cost subset of trees $\mathcal{F} \subseteq \mathcal{T}$ that connects at least k demands. This perspective motivates a natural integer programming formulation, which is surprisingly augmented with additional valid constraints stating that trees with cost greater than Δ cannot be picked.

$$\begin{aligned}
\text{OPT} = \text{minimize} \quad & \sum_{T \in \mathcal{T}} c(T) x_T & & \\
\text{subject to} \quad & \sum_{i=1}^{d} z_i \leq d - k & & (3.1)\\
& \sum_{T:(s_i,t_i) \in \mathcal{D}(T)} x_T + z_i \geq 1 & \forall\, 1 \leq i \leq d & (3.2)\\
& x_T = 0 & \forall\, T \in \mathcal{T} : c(T) > \Delta & (3.3)\\
& x_T,\, z_i \in \{0,1\} & \forall\, T \in \mathcal{T},\, 1 \leq i \leq d & (3.4)
\end{aligned}$$

In this formulation, the variable x_T indicates whether the tree T is chosen for the collection we construct, whereas z_i indicates whether the demand (s_i, t_i) is not connected. Constraint (3.1) forces any feasible solution to connect at least k demands. Constraint (3.2) ensures that we either pick at least one tree that connects (s_i, t_i), or specify that this demand remains unconnected by setting $z_i = 1$. Last but not least, the additional constraint (3.3) appears to be completely redundant at the moment, since $\text{OPT} \leq \Delta$.

We now relax the complicating constraint (3.1), and lift it to the objective function multiplied by $\lambda \geq 0$. The resulting Lagrangian relaxation is:

$$\begin{aligned}
\text{LR}(\lambda) = \text{minimize} \quad & \sum_{T \in \mathcal{T}} c(T) x_T + \lambda \left(\sum_{i=1}^{d} z_i - (d-k) \right) & & \\
\text{subject to} \quad & \sum_{T:(s_i,t_i) \in \mathcal{D}(T)} x_T + z_i \geq 1 & \forall\, 1 \leq i \leq d & (3.5)\\
& x_T = 0 & \forall\, T \in \mathcal{T} : c(T) > \Delta & (3.6)\\
& x_T,\, z_i \in \{0,1\} & \forall\, T \in \mathcal{T},\, 1 \leq i \leq d & (3.7)
\end{aligned}$$

We remark that, excluding the constant term of $-\lambda(d-k)$ in the objective function, $\mathrm{LR}(\lambda)$ describes a closely related instance of the prize-collecting Steiner forest problem, in which all demands are coupled with a uniform penalty of λ. However, the current formulation retains the extra restriction imposing an upper bound of Δ on the individual cost of every tree picked, which turns out to be of great importance. Indeed, simple examples demonstrate that the optimal cost may fluctuate by a factor of $\Omega(d)$ should this restriction be discarded, rendering any prize-collecting scheme obsolete. We refer to the above-mentioned instance as $I_{\lambda,\Delta}$, and use $\mathrm{OPT}(I_{\lambda,\Delta})$ to denote its optimum value. It is easy to verify that $\mathrm{LR}(\lambda) = \mathrm{OPT}(I_{\lambda,\Delta}) - \lambda(d-k)$ provides a lower bound on OPT for any $\lambda \geq 0$, by observing that an optimal solution to the original problem is also a feasible solution to $\mathrm{LR}(\lambda)$, whose cost is at most OPT.

3.2 Setting Up the Prize-Collecting Solutions

Preliminaries. In what follows, we apply the techniques developed in Section 2 to approximate the cost of an optimal k-Steiner forest by a convex combination of prize-collecting solutions, each of which consists of trees whose individual costs do not significantly exceed Δ. At this point, we remind the reader that, given any $\lambda \geq 0$, the polynomial-time algorithm described in Theorem 2 provides a bicriteria approximation for $I_{\lambda,\Delta}$. To simplify the oncoming discussion, it would be convenient to interpret the resulting solution in terms of the indicators (x, z) defined above. For this purpose, let x^λ indicate which trees were picked by the algorithm, and let z^λ indicate which demands were left unconnected. Clearly, (x^λ, z^λ) satisfies the constraints (3.5) and (3.7), but not necessarily (3.6). However, recalling that $\alpha(n,d) = 12\min\{n^{2/3}, \sqrt{d}\}$ and $\beta(n,d) = 4\min\{n^{2/3}, \sqrt{d}\}$, Theorem 2 guarantees $c(T) \leq \beta(n,d)\Delta$ for every $T \in \mathcal{T}$ with $x^\lambda_T = 1$, while

$$\sum_{T\in\mathcal{T}} c(T)x^\lambda_T + \alpha(n,d)\mathcal{H}(d)\sum_{i=1}^{d}\lambda z^\lambda_i \leq \alpha(n,d)\mathcal{H}(d)\cdot\mathrm{OPT}(I_{\lambda,\Delta}) \ . \tag{3.8}$$

The binary search. Intuitively speaking, increasingly larger values of λ compel the prize-collecting algorithm to connect all demands, as even a single penalty becomes unaffordable. Similar reasoning suggests that very few demands would be connected when λ is sufficiently small. In the next lemma, we obtain concrete bounds for this asymptotic behavior.

Lemma 7. *When $\lambda > d\Delta$, the solution (x^λ, z^λ) connects all demands. On the other hand, we may assume without loss of generality that (x^0, z^0) connects strictly less than k demands.*

This observation determines an initial interval over which we conduct a binary search, consisting of a polynomially-bounded number of calls to the prize-collecting algorithm. As a result, we find $\lambda^- \leq \lambda^+$ along with approximate solutions $(x^{\lambda^-}, z^{\lambda^-})$ and $(x^{\lambda^+}, z^{\lambda^+})$ satisfying the following properties:

1. $\lambda^+ - \lambda^- \leq \frac{c_{\min}}{d}$, where $c_{\min} > 0$ denotes the minimum cost of an edge in the input graph.

2. The solution $(x^{\lambda^-}, z^{\lambda^-})$ connects $k^- \leq k$ demands, whereas $(x^{\lambda^+}, z^{\lambda^+})$ connects $k^+ \geq k$ demands.

For the remainder of this section, we use $\mathcal{F}^-$ and $\mathcal{F}^+$ to denote the collections of trees that were picked by $(x^{\lambda^-}, z^{\lambda^-})$ and $(x^{\lambda^+}, z^{\lambda^+})$, respectively. In addition, we assume without loss of generality that $k^- < k$, as $\mathcal{F}^-$ by itself provides an approximation factor of $\alpha(n,d)\mathcal{H}(d)$ when $k^- = k$; this claim follows from a straightforward application of the inequalities (3.8) and $\mathrm{LR}(\lambda^-) \leq \mathrm{OPT}$.

Fractionally approximating OPT. Even though we can exercise inequality (3.8) to show that the cost of $\mathcal{F}^-$ comes within a factor of $\alpha(n,d)\mathcal{H}(d)$ of optimal, this solution is clearly infeasible. The situation is quite the opposite with respect to $\mathcal{F}^+$, which is a feasible solution whose cost may be arbitrarily large in comparison to OPT. Having observed these facts, we argue that the cost of an optimal k-Steiner forest can be approximated by a convex combination of $\mathcal{F}^-$ and $\mathcal{F}^+$, an essential characterization on which the forthcoming analysis will depend.

Lemma 8. *Let ξ be the unique solution to $\xi k^+ + (1-\xi)k^- = k$, that is, $\xi = \frac{k-k^-}{k^+-k^-}$. Then,*

$$\xi \sum_{T \in \mathcal{F}^+} c(T) + (1-\xi) \sum_{T \in \mathcal{F}^-} c(T) \leq 2\alpha(n,d)\mathcal{H}(d) \cdot \mathrm{OPT} \ .$$

3.3 Assembling an Approximate Integral Solution

In the following, we focus our attention on combining $\mathcal{F}^-$ and $\mathcal{F}^+$ into a single collection of trees $\mathcal{F}$, trying to balance between two contradicting objectives. On the one hand, we would like $\mathcal{F}$ to connect a sufficient number of demands; on the other hand, the factor by which the cost of $\mathcal{F}$ deviates from OPT should be minimized. Roughly speaking, this collection is created by augmenting $\mathcal{F}^-$ with a carefully chosen subset $\mathcal{Q} \subseteq \mathcal{F}^+$, connecting at least $k - k^-$ demands that were left unconnected by $\mathcal{F}^-$. For this purpose, we specialize a greedy procedure that has been recently suggested by Levin and Segev [14] for partially covering general set systems.

Lemma 9. *There is a polynomial-time algorithm that finds a subset of trees $\mathcal{Q} \subseteq \mathcal{F}^+$ connecting at least $k - k^-$ demands in $\mathcal{D}(\mathcal{F}^+) \setminus \mathcal{D}(\mathcal{F}^-)$, such that*

$$\sum_{T \in \mathcal{Q}} c(T) \leq \xi \sum_{T \in \mathcal{F}^+} c(T) + \max_{T \in \mathcal{F}^+} c(T) \ .$$

We complete the construction of an approximate k-Steiner forest by defining $\mathcal{F} = \mathcal{F}^- \cup \mathcal{Q}$, noting that this collection constitutes a feasible solution, as it connects at least k demands. With the latter observation in mind, we are now ready to establish the main result of this paper, claiming that the cost of $\mathcal{F}$ is within a factor of $O(\min\{n^{2/3}, \sqrt{d}\} \cdot \log d)$ of optimal. The underlying idea is to decompose the overall cost into three parts, and separately bound each of them by utilizing previously stated results, including the upper bound on the individual cost of every tree in $\mathcal{F}^+$:

$$\begin{aligned}\sum_{T\in\mathcal{F}} c(T) &= \sum_{T\in\mathcal{F}^-} c(T) + \sum_{T\in Q} c(T)\\ &\le \xi \sum_{T\in\mathcal{F}^-} c(T) + \left((1-\xi)\sum_{T\in\mathcal{F}^-} c(T) + \xi \sum_{T\in\mathcal{F}^+} c(T)\right) + \max_{T\in\mathcal{F}^+} c(T)\\ &\le (\xi+2)\alpha(n,d)\mathcal{H}(d)\cdot \mathrm{OPT} + 2\beta(n,d)\cdot \mathrm{OPT}\\ &= O\left(\min\left\{n^{2/3}, \sqrt{d}\right\}\cdot \log d\right)\cdot \mathrm{OPT}\ .\end{aligned}$$

The first inequality is an immediate consequence of Lemma 9. The second inequality is implied by Lemma 8, the observation that $\sum_{T\in\mathcal{F}^-} c(T) \le \alpha(n,d)\mathcal{H}(d)\cdot \mathrm{OPT}$, and the fact that $c(T) \le \beta(n,d)\Delta \le 2\beta(n,d)\cdot \mathrm{OPT}$ for every $T \in \mathcal{F}^+$. The last equation holds since $\xi \in [0,1]$, $\alpha(n,d) = 12\min\{n^{2/3}, \sqrt{d}\}$ and $\beta(n,d) = 4\min\{n^{2/3}, \sqrt{d}\}$.

References

1. A. Agrawal, P. N. Klein, and R. Ravi. When trees collide: An approximation algorithm for the generalized Steiner problem on networks. *SIAM Journal on Computing*, 24(3):440–456, 1995.
2. S. Arora and G. Karakostas. A $2+\epsilon$ approximation algorithm for the k-MST problem. In *11th SODA*, pages 754–759, 2000.
3. S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and the hardness of approximation problems. *Journal of the ACM*, 45(3):501–555, 1998.
4. Y. Asahiro, K. Iwama, H. Tamaki, and T. Tokuyama. Greedily finding a dense subgraph. *Journal of Algorithms*, 34(2):203–221, 2000.
5. D. Bienstock, M. X. Goemans, D. Simchi-Levi, and D. P. Williamson. A note on the prize collecting traveling salesman problem. *Mathematical Programming*, 59:413–420, 1993.
6. A. Blum, R. Ravi, and S. Vempala. A constant-factor approximation algorithm for the k-MST problem. *Journal of Computer and System Sciences*, 58(1):101–108, 1999.
7. U. Feige, G. Kortsarz, and D. Peleg. The dense k-subgraph problem. *Algorithmica*, 29(3):410–421, 2001.
8. U. Feige and M. Langberg. Approximation algorithms for maximization problems arising in graph partitioning. *Journal of Algorithms*, 41(2):174–211, 2001.
9. N. Garg. A 3-approximation for the minimum tree spanning k vertices. In *37th FOCS*, pages 302–309, 1996.
10. N. Garg. Saving an epsilon: A 2-approximation for the k-MST problem in graphs. In *37th STOC*, pages 396–402, 2005.
11. M. X. Goemans and D. P. Williamson. A general approximation technique for constrained forest problems. *SIAM Journal on Computing*, 24(2):296–317, 1995.
12. M. Hajiaghayi and K. Jain. The prize-collecting generalized Steiner tree problem via a new approach of primal-dual schema. In *17th SODA*, pages 631–640, 2006.
13. Q. Han, Y. Ye, and J. Zhang. An improved rounding method and semidefinite programming relaxation for graph partition. *Mathematical Programming*, 92(3):509–535, 2002.
14. A. Levin and D. Segev. Partial multicuts in trees. In *3rd WAOA*, pages 320–333, 2005.
15. C. H. Papadimitriou and M. Yannakakis. Optimization, approximation, and complexity classes. *Journal of Computer and System Sciences*, 43(3):425–440, 1991.
16. R. Ravi and M. X. Goemans. The constrained minimum spanning tree problem (extended abstract). In *5th SWAT*, pages 66–75, 1996.

Balancing Applied to Maximum Network Flow Problems
(Extended Abstract)

Robert Tarjan*, Julie Ward, Bin Zhang, Yunhong Zhou, and Jia Mao**

Hewlett-Packard Laboratories
1501 Page Mill Rd, Palo Alto, CA 94304
{robert.tarjan, jward, bin.zhang2, yunhong.zhou}@hp.com

Abstract. We explore *balancing* as a definitional and algorithmic tool for finding minimum cuts and maximum flows in ordinary and parametric networks. We show that a standard monotonic parametric maximum flow problem can be formulated as a problem of computing a particular maximum flow that is *balanced* in an appropriate sense. We present a divide-and-conquer algorithm to compute such a balanced flow in a logarithmic number of ordinary maximum-flow computations. For the special case of a bipartite network, we present two simple, local algorithms for computing a balanced flow. The local balancing idea becomes even simpler when applied to the ordinary maximum flow problem. For this problem, we present a round-robin arc-balancing algorithm that computes a maximum flow on an n-vertex, m-arc network with integer arc capacities of at most U in $O(n^2 m \log(nU))$ time. Although this algorithm is slower by at least a factor of n than other known algorithms, it is extremely simple and well-suited to parallel and distributed implementation.

1 Introduction

In this paper we explore the idea of *balancing* as a definitional and algorithmic tool in the solution of maximum flow and minimum cut problems on capacitated networks. One motivation for introducing this concept is its application to monotone parametric flow problems. For such a problem, we define a λ-*balanced flow*, which is a maximum flow in which the flows on the parametrically-capacitated arcs are balanced in an appropriate way. From a λ-balanced flow it is easy to extract a maximum flow and a minimum cut for *any* value of the parameter. Thus a λ-balanced flow provides a succinct representation of the entire parameterized set of solutions of the parametric problem. We describe a divide-and-conquer algorithm that finds a λ-balanced flow in a logarithmic number of ordinary maximum flow computations.

Balancing can also be used as an algorithmic technique to develop simple, local algorithms for maximum flow problems. For the special case of the maximum flow problem on a bipartite parametric network, we develop two local algorithms

* Also Department of Computer Science, Princeton University.
** Dept. of Comp. Sci., UC San Diego. The work was done at HP Labs as an intern.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 612–623, 2006.

to compute a λ-balanced flow. One balances flow across a pair of arcs at a time, and the other balances flows across all arcs incident to a single vertex, a subgraph we call a *star*. For the ordinary maximum flow problem (not restricted to bipartite graphs), we develop an even simpler local algorithm that balances across one arc at a time. Such single-arc balancing was previously used by Awerbuch and Leighton [2, 3] in (more-complicated) approximation algorithms for finding multi-commodity flows. We provide a tight running-time analysis of our arc-balancing algorithm.

Our paper is organized as follows. The remainder of this section gives our basic network flow terminology. More specialized terminology is developed in later sections. Section 2 contains a discussion of some related work. Section 3 discusses the monotone parametric problem, discusses related work, defines λ-balanced flows, and gives our divide-and-conquer algorithm for finding such a flow. Section 4 presents our two local algorithms for finding a λ-balanced flow in a bipartite parametric network. Section 5 describes and analyzes our arc-balancing algorithm for ordinary maximum flow. Section 6 contains final remarks.

A *network* is a directed graph G with two distinguished vertices, a *source* s and a *sink* t, along with a non-negative capacity function c on the arcs. We allow an arc to have infinite capacity. We shall assume that G is *symmetric*: if (v, w) is an arc, so is (w, v). We sometimes refer to the pair (v, w), (w, v) as the *edge* $\{v, w\}$. We can make a network symmetric without affecting the maximum flow problem by adding an arc (w, v) with zero capacity for each arc (v, w) whose reversal (w, v) is not originally present. We assume (without loss of generality) that arcs into the source and out of the sink have zero capacity. In stating resource bounds we shall denote by n the number of vertices, by m the number of arcs, and by U the maximum arc capacity, assuming that the capacities are integers. We assume arbitrary-precision real arithmetic.

A *pseudoflow* on G is a real-valued function f on the arcs that is *antisymmetric*: $f(v, w) = -f(w, v)$ for every arc (v, w), and that obeys the *capacity constraints*: $f(v, w) \leq c(v, w)$ for every arc (v, w). Given a pseudoflow, the *excess* at a vertex v is $\sum\{f(u, v) \mid (u, v) \text{ is an arc}\}$. A pseudoflow is a *preflow* if $e(v)$ is non-negative for every vertex v other than s, and it is a *flow* if $e(v)$ is zero for every vertex v other than s and t. The *value* of a flow is $e(t)$. A flow is *maximum* if it has maximum value over all possible flows. The *maximum flow problem* is that of finding a maximum flow in a given network.

A problem dual to the maximum flow problem is the minimum (source-sink) cut problem. A *cut* is a partition of the vertex set into two parts, one containing the source, the other containing the sink. We shall denote a cut by the set X of vertices in the part of the partition containing the source. An arc (v, w) with v but not w in X *crosses* the cut. The *capacity* of the cut is $\sum\{c(v, w) \mid (v, w) \text{ crosses } X\}$. A cut is *minimum* if it has minimum capacity. The *minimum cut problem* is that of finding a minimum cut in a given network.

Given a pseudoflow, the *residual capacity* of an arc (v, w) is $r(v, w) = c(v, w) - f(v, w)$. An arc (v, w) is *saturated* if $r(v, w) = 0$; otherwise, it is *unsaturated* or *residual*. An *augmenting path* is a path of residual arcs; its capacity is the

minimum of the residual capacities of its arcs. A cut is *saturated* if every arc crossing it is saturated. The maximum-flow minimum-cut theorem implies that a flow is maximum and a cut is minimum if and only if the flow saturates every arc across the cut.

2 Related Work

Maximum Flows. The maximum flow problem is a central problem in network optimization. Goldberg and Rao [14] modified the blocking flow approach of Dinitz [7] by using a distance function based on the residual capacities (rather than just assigning each arc a length of one) and obtained an algorithm running in $O(\min\{n^{2/3}, m^{1/2}\}m \log(n^2/m) \log U)$ time on networks with integer arc capacities. For most interesting ranges of n, m and U, their algorithm is fastest. This algorithm uses dynamic trees [22] as well as other involved ideas.

Another approach that has led to both good theoretical bounds and fast practical implementations [6] is the *preflow-push method* of Goldberg and Tarjan [13]. The preflow push method is simple and local, although its theoretically fastest versions use sophisticated ideas. The use of dynamic trees gives an $O(nm \log(n^2/m))$ running time [13]. Combining excess-scaling with the use of dynamic trees gives an $O(nm \log((n/m)\sqrt{\log U} + 2))$ running time [1]. See [14] for a table showing a complete history of complexity improvements for the problem.

A third approach notable for its simplicity is the MA-ordering maximum flow algorithm of Fujishige [10], which runs in $O(n(m + n \log n) \log(nU))$ time, improvable to $O(nm \log U)$ by using scaling. A fast practical version of this algorithm has been obtained by adding the preflow idea [18].

Parametric Flows. The monotone parametric network flow problem is a generalization of the ordinary network flow problem in which each arc incident to the source has its capacity given by a monotone increasing function of λ, and each arc incident to the sink has its capacity given by a monotone decreasing function of λ, where λ is a common value in the range $[0, \infty)$. For any fixed value of λ, a parametric network has a minimum cut. The parametric minimum cut problem is to compute all minimum cuts for *every* possible value of λ. This problem has a variety of applications, including product selection [4, 21], repair kit selection [17], database record segmentation [8], and baseball team elimination [15, 19]. See Gallo et al. [11] and Hochbaum [16] for discussions of some of these applications. Gallo, Grigoriadis, and Tarjan [11] gave a divide-and-conquer method that for certain preflow-push maximum-flow algorithms has a time complexity asymptotically the same as one maximum flow computation. Their fastest algorithm runs in $O(nm \log(n^2/m))$ time. For a parametric bipartite network, their algorithm runs in time $O(km \log(k^2/m + 2))$ where k is the size of the smaller side of the bipartite graph. Gallo et al. also show that the capacities on the arcs into the sink can be made constant, with no loss of generality.

Balancing. The definition of a λ-balanced flow that we present in Section 3, and the two local algorithms for computing such a flow that we give in Section 4,

were originally developed by the second and third authors [24, 25]: their work was the genesis of this paper. Awerbuch and Leighton [2] gave a local balancing algorithm for approximating maximum multi-commodity flow. The round-robin arc-balancing algorithm for ordinary single-commodity maximum flow that we give in Section 5 is closely related to (but simpler than) this algorithm. Awerbuch and Leighton [3] later gave a more-complicated but more-efficient balancing algorithm for approximating maximum multi-commodity flows. They did not explore the case of a single commodity, but for this special case of their algorithm a time bound of $O((n^2m/\epsilon^3)\log^3(m/\epsilon))$ to compute a $(1+\epsilon)$-approximation follows from their general bound.

3 Parametric Minimum Cuts and Balanced Flows

For our purposes, a *monotone parametric network* is one in which the capacity of each arc (s, v) is a strictly increasing function $c(\lambda, v)$ of a single parameter λ, for λ in the range $[0, \infty)$, and every other arc has fixed capacity. For simplicity we assume $c(0, v) = 0$ for all arcs (s, v). We further assume that there is a cut S of finite capacity containing every vertex v such that (s, v) is an arc. (In particular, this means that (s, t) is not an arc.) An important special case is $c(\lambda, v) = \lambda$ for every arc (s, v). The problem we wish to solve is to compute a minimum cut for every value of λ. This problem and related ones have a variety of applications [4, 21, 8, 17, 15, 19], with the latest application in webgraph clustering [9]. Even though there are an infinite number of possible λ values, there are only a finite number of possible cuts. Furthermore, in solving this problem it is only necessary to consider nested families of cuts: as the value of λ increases, the capacities of the arcs out of the source increase, and the minimum cut contains more-and-more vertices (more precisely, the minimum cut with fewest vertices contains more-and-more vertices) [8, 23]. Indeed, we can assign to each vertex v a value $\lambda(v)$ such that $S = S(\lambda) = \{v \mid \lambda(v) \leq \lambda\}$ is a minimum cut for λ [23]. (Vertices v with $\lambda(v) = \lambda$ can actually be included in S or not.) Thus the problem we solve is that of computing such a set of vertex values, which we call a *cut function*. Not only does a cut function define a nested family of minimum cuts, it also allows easy computation of the minimum cut capacity as a function of λ.

One way to compute a cut function is to find maximum flows for appropriately chosen values of λ. A straightforward application of this idea requires at most $n - 2$ maximum flow computations. Gallo et al. [11] describe a complicated divide-and-conquer method that runs a preflow-push maximum-flow algorithm both forwards and backwards on each subproblem, and starts each subproblem with a preflow given by the solution to an enclosing subproblem. By a clever amortization argument, they show that the overall running time to compute a cut function is only a constant factor larger than the time bound for the preflow push algorithm. In particular, they obtain a bound of $O(nm\log(n^2/m))$ time to compute a cut function. For a bipartite network, they claim a time bound of $O(km\log(k^2/m+2))$, where k is the size of the smaller side of the bipartite

graph. The Gallo et al. algorithm is actually presented as a way to compute the minimum cut capacity function and it is restricted to affine arc capacity functions, but it is easily modified to compute a cut function, and Gusfield and Martel [15] show how to extend the algorithm to nonlinear arc capacity functions.

We reformulate the problem of computing a cut function as one of computing a particular maximum flow in the network formed by replacing all the parameterized capacities by ∞. Specifically, replace all capacities of arcs out of the source by ∞, and let f be a maximum flow of the resulting network. We can compute f by applying any maximum flow algorithm. For each arc (s, v), we define $\lambda(f, v)$ to be the unique value of λ such that $c(\lambda, v) = f(s, v)$. We call the flow f *λ-balanced* if there is no augmenting path avoiding s from a vertex v to a vertex w with $\lambda(f, v) < \lambda(f, w)$. The idea is that a λ-balanced flow is a maximum flow that, to the extent possible, equalizes the flows on the arcs out of s, where equalization is with respect to λ-values. In particular, if $c(\lambda, v) = \lambda$ for every arc (s, v), a λ-balanced flow is a maximum flow that, to the extent possible, equalizes the flows on the arcs out of s. In this special case, the notion of a λ-balanced flow is equivalent to that of a maximum flow that achieves the best possible lexicographic sharing. See Gallo et al. [11], Section 4.1 for a discussion of various notions of flow-sharing, including lexicographic sharing.

Given a λ-balanced flow f, we can obtain a cut function as follows. We set $\lambda(s) = 0$. For each arc (s, v), we let $\lambda(v) = \lambda(f, v)$. For each other vertex w, we let $\lambda(w) = \min\{\lambda(v) \mid \text{there is an augmenting path avoiding } s \text{ from } v \text{ to } w\}$, with the minimum over the empty set defined to be ∞. (In particular, $\lambda(t) = \infty$.) It is straightforward to compute this cut function in $O(n \log n + m)$ time by first sorting the vertices v such that (s, v) is an arc in non-decreasing order by $\lambda(v)$ and then doing an incremental graph search along residual arcs from vertices of small λ, increasing λ to the next possible value when the search runs out of arcs to traverse.

A straightforward way to attempt to compute a λ-balanced flow is to apply the definition. Specifically, begin by finding any maximum flow on the network with parameterized capacities replaced by ∞. Then repeat the following step until it no longer applies: find an augmenting path avoiding s from a vertex v to a vertex w with $\lambda(f, v) < \lambda(f, w)$. Send as much flow as possible along the cycle formed by this path and the arcs (w, s) and (s, v) without causing $\lambda(f, v)$ to exceed $\lambda(f, w)$. The amount of flow increase is the minimum of the capacity of the augmenting path and $c(\lambda, v) - f(s, v)$, where λ is the unique value such that $c(\lambda, v) - f(s, v) = f(s, w) - c(\lambda, w)$. Unfortunately, in general this algorithm does not terminate, although it leads to simple, efficient algorithms for the special case of bipartite networks, as we discuss in Section 4.

On the other hand, the Gallo et al. divide-and-conquer algorithm can be reformulated in a straightforward way to compute a λ-balanced flow, without affecting its asymptotic time bound. We develop a simpler version of this algorithm, which uses a maximum-flow computation as a black box. The initialization is to replace the capacities of the parameterized arcs by ∞ and compute a maximum flow. The main step of the algorithm computes a maximum flow on an auxiliary

network, thereby either making measurable progress toward producing a balanced flow or splitting the problem in two. Specifically, given a maximum flow, compute $\lambda(v)$ for every arc (s, v). If all these values are equal, the flow is balanced. Otherwise, choose a value λ strictly between the maximum and minimum of the $\lambda(v)$'s. Construct a dummy source s' and a dummy sink t', and construct dummy arcs as follows: if (s, v) is an arc with $\lambda(f, v) < \lambda$, construct a dummy arc (s', v) with capacity $c(\lambda, v) - f(s, v)$; if (s, v) is an arc with $\lambda(f, v) > \lambda$, construct a dummy arc (v, t') with capacity $f(s, v) - c(\lambda, v)$. Delete s and t and all incident arcs, and replace the capacity of all arcs not incident to s' and t' by their residual capacity. Find a maximum flow in the resulting auxiliary network. Add this flow (ignoring the flow on the new arcs) to the flow on the original network. Adjust the flow on the arcs out of s so that the new f is indeed a flow. (That is, restore flow conservation at each vertex v such that (s, v) is an arc.)

Note that this flow modification increases $\lambda(f, v)$ for any vertex v with $\lambda(f, v) < \lambda$ to at most λ, and decreases $\lambda(f, v)$ for any vertex v with $\lambda(f, v) > \lambda$ to at least λ. In the auxiliary network, one of three things can happen. The maximum flow on the original network corresponds to a saturated cut. If this cut is $\{s'\}$, then after the flow augmentation all arcs (s, v) have $\lambda(f, v) \geq \lambda$. If the cut is $V - \{t'\}$, then after the augmentation all arcs (s, v) have $\lambda(f, v) \leq \lambda$. In either of these cases the flow augmentation has reduced the range of the λ values, and we repeat the main step. In any other case, the problem can be split into two nontrivial subproblems, as follows. Let the saturated cut in the auxiliary network be S'. Let $S = S' \cup \{s\} - \{s'\}$, and let $\overline{S} = V - S$. Form one subproblem by contracting all vertices in S into s and then deleting all unparameterized arcs out of s. Form the other subproblem by contracting all vertices in $\overline{S}$ into t and deleting all parameterized arcs into t. The current maximum flow on the original network gives maximum flows on the subproblems. Convert these into λ-balanced flows by applying the method (minus the initialization) to each subproblem. The λ-balanced flows on the subproblems, plus the flows on the unparameterized contracted arcs, plus appropriate flows on the parameterized arcs, give a λ-balanced flow on the original network.

It remains to choose λ in each iteration of the main step. Since each subproblem contains an original arc (s, v), the number of subproblem splits is at most $n - 3$. If the value chosen for λ is such that the sum of the capacities of arcs out of s' equals the sum of arc capacities into t', then applying the main step either splits the problem or makes all λ's equal and hence completes the computation. Thus this choice of λ gives a bound of $n-3$ maximum flow computations. On the other hand, choosing λ equal to the average of the maximum and the minimum of the $\lambda(v)$'s guarantees that the difference between the maximum and the minimum of the $\lambda(v)$'s drops by a factor of two with each iteration of the main step, whether or not the problem is split. If we alternate between these two choices, the number of iterations of the main loop is $2 \min\{n - 3, 1 + \log R\}$, where R is the ratio between the initial difference between λ values and the minimum possible difference between λ values. In the important special case where all the parameterized capacities are equal to λ, R is at most n^2U, giving us a bound

of at most $2\min\{n-3, 1+\log(n^2U)\}$ on the number of maximum flow computations needed to find a λ-balanced flow. If we use the Goldberg-Rao algorithm for the maximum flow computations, we obtain a bound of

$$O\left(\min\left\{n^{2/3}, m^{1/2}\right\} \cdot m \cdot \log(n^2/m) \cdot \log U \cdot \min\{n, \log(nU)\}\right)$$

to find a λ-balanced flow. This bound is better than that of Gallo et al. for interesting ranges of the parameters. Goldberg and Rao [14] claim that their maximum flow algorithm in combination with ideas in the Gallo et al. paper [11] gives a bound of $O(\log U)$ times their maximum flow bound to find parametric flows. This bound is better than ours $O(\log(nU))$. But we can find no justification in the Gallo et al. paper for the Goldberg-Rao claim, even for simpler versions of the problem such as computing a maximum or minimum breakpoint of the minimum cut capacity function, and in a private conversation [12] Goldberg has retracted the claim.

As compared to the analysis of the Gallo et al. algorithm, our analysis is straightforward. In particular, it does not rely on starting the maximum flow computations on the subproblems with partially computed solutions, or on the fact that the sizes of the subproblems at a given level of recursion sum to at most the size of the original problem. We wonder if the $\log R$ factor in our time bound can be reduced or eliminated, either by doing a more sophisticated analysis, or by exploiting the internals of certain maximum flow algorithms. Another issue is to bound R for more-complicated parametric capacities, such as arbitrary linear or piecewise-linear ones.

4 Local Algorithms for Bipartite Parametric Flow-Balancing

In this section we restrict our attention to the special case of a parametric network that is bipartite; that is, the vertices can be partitioned into two sets A and B such that every arc has one end in A and one end in B. We further assume that the sink t is in A (so that every arc (v, t) has v in B) and the source is in B. We also assume that *every* vertex in A is incident to s. This restriction still covers most of the interesting applications. (See [11].)

In this setting, we can efficiently convert a maximum flow into a λ-balanced flow by doing local balancing operations based on the definition of a λ-balanced flow. We describe two algorithms, one of which uses two-arc augmenting paths, and the other of which balances a star at each step, where a star is the subgraph induced by the set of arcs incident on a single vertex. The main advantage of these algorithms is that they are very simple as compared to algorithms based on standard maximum-flow algorithms.

Two-Arc-Balancing Algorithm: Begin with a maximum flow on the network with parameterized capacities replaced by ∞. Repeat the following step until it no longer applies: find an augmenting path $((v, u), (u, w))$ with $u \neq s$ and

$\lambda(f, v) < \lambda(f, w)$. Add as much flow as possible to this path without violating the capacity constraints or causing $\lambda(f, v)$ to exceed $\lambda(f, w)$. Increase the flow on (s, v) and decrease the flow on (s, w) by the same amount.

If this algorithm terminates (which it need not), the resulting flow is balanced. That is, if the network is bipartite with all vertices in A incident to s, then the definition of a balanced flow need only include two-arc augmenting paths.

We do not explore this algorithm further here, since there is a better alternative, which is to simultaneously consider all two-arc paths for a given intermediate vertex u. This idea gives the following algorithm:

Star-Balancing Algorithm: Begin with a maximum flow on the network with parameterized capacities replaced by ∞. Repeat the following step until it no longer applies: find a vertex $u \neq s$ such that there is an augmenting path $((v, u), (u, w))$ with $\lambda(f, v) < \lambda(f, w)$. Modify the flows on all arcs into u, and on corresponding arcs out of s, so that there is no such augmenting path.

The time required to do a star-balancing step depends on the complexity of the parameterized capacities. In the special case where all such capacities equal λ, or more generally where they are affine functions of λ, it is possible to star-balance a vertex u in time proportional to the degree of u [5]. The *round-robin star-balancing algorithm* is the version of star-balancing that repeatedly iterates over the vertices in B doing star-balancing steps, until an entire pass does not change the flow, or until a suitable stopping condition holds. (Vertices joined by an augmenting path have λ-values close enough that a simple postprocessing phase will complete the computation.)

A variant of round-robin star-balancing that may be more efficient in practice is to maintain a *working set* W of vertices in B whose last examination resulted in a change in the flow. Initially $W = B$. During a pass over W, if an examination of w results in sufficiently small flow change or no flow change, w is dropped from W. When W becomes empty, it is reset to be B. If a pass over $W = B$ results in no flow change, the computation is complete. A more-refined idea is to periodically find saturated cuts, as in the divide-and-conquer algorithm, and to split the working set into separate subsets based on the cuts.

At least for the special case in which all parameterized capacities are equal to λ, the running time of the round-robin star-balancing algorithm can be analyzed using the techniques we use to analyze the round-robin arc-balancing algorithm described in the next section, resulting in a similar running time. We omit this analysis and a discussion of appropriate stopping rules here since the ordinary maximum flow problem provides a simpler analytical setting.

In the important special case in which all arcs from A to B have infinite capacity, it is easy to construct an initial maximum flow in linear time without using a maximum-flow algorithm, by saturating all arcs into the sink, assigning flow to the arcs from A to B to get flow conservation on the vertices in B, for example by averaging the outgoing flow over the incoming arcs, and assigning flow to the arcs out of s to get flow conservation on the vertices in A. It is also easier to do star balancing steps.

5 Ordinary Maximum Flows Via Arc-Balancing

The idea of balancing flows can be fruitfully applied to the ordinary maximum flow problem. We develop an algorithm that computes a maximum flow by maintaining a pseudoflow and repeatedly moving flow along individual arcs so as to balance flow excesses. The algorithm has the virtue of being extremely simple and local, and it provides a simpler setting than the parametric flow problem in which to do algorithmic analysis. The key observation that justifies the algorithm is that if there is no augmenting path from a vertex of positive excess or the source to a vertex of negative excess or the sink, then both a minimum cut and a maximum flow can easily be extracted from the pseudoflow.

Arc-Balancing Algorithm: Construct an initial pseudoflow by saturating every arc out of s and every arc into t, and assigning zero flow to every other arc. Repeat the following step until it no longer applies (or until a suitable stopping rule holds):

> **Move:** Choose a residual arc (v, w) with $e(v) > e(w)$, $w \neq s$, and $v \neq t$. Increase the flow on (v, w) by $\min\{r(v, w), (e(v) - e(w))/2\}$.

A move on an arc (v, w) leaves $e(v) \geq e(w)$. Either it makes $e(v)$ and $e(w)$ equal, in which case we call it a *balancing move*, or it saturates (v, w), in which case we call it a *saturating move*, or both. An *α-move* is a move on an arc of residual capacity at least α.

It is easy to construct examples on which this algorithm runs forever. Henceforth in this section we shall assume that the arc capacities are integers bounded by U. Given integer arc capacities, it suffices to stop doing moves when there are no $(1/n^2)$-moves. (We discuss another stopping rule below.) Furthermore one can restrict the moves to $(1/n^2)$-moves. Also, $\log(n^2U) + O(1)$ bits of precision suffice in the computations. (Unlike most maximum flow algorithms, our algorithm does *not* maintain integrality of flows.)

Two notions help in understanding the behavior of this algorithm. A *canonical cut* is a cut $S(a) = \{s\} \cup \{v \neq t \mid e(v) \geq a\}$ for some real value a. If the algorithm ever saturates a canonical cut, no move can ever again occur on any arc crossing the cut. A *section* $S(a, b)$ is a non-empty set of vertices $S(a) - S(b)$ such that both $S(a)$ and $S(b)$ are saturated. Once the algorithm creates a section, the minimum excess in the section can never decrease, the maximum excess in the section can never increase, and the *excess difference* in the section, defined to be the maximum excess minus the minimum excess, can never decrease. The initial pseudoflow defines a section containing all the vertices except s and t.

As the algorithm proceeds, it saturates canonical cuts, splitting the network into smaller-and-smaller sections. All moves take place within sections. To find a minimum cut, the only important section is the *active section*, defined to be the minimal section $S(a, b)$ with $a \leq 0$ and $b > 0$. It suffices to do moves only in the active section. Another stopping rule for integer capacities is to stop when the sum of positive excesses in the active section is less than one or the sum of negative excesses in the active section is greater than minus one. As a special

case, it suffices to stop when the maximum excess of the non-sink vertices is non-positive or the minimum excess of the non-source vertices is non-negative.

Once the arc-balancing algorithm stops, a minimum cut can be extracted as follows.

Minimum Cut Computation: If the main loop stops with the sum of positive excesses in the active section less than one, find the saturated cut $S = S(b)$ with minimum $b > 0$. Alternatively, if the main loop stops with the sum of negative excesses in the active section greater than minus one, find the saturated cut $S = S(a)$ with maximum $a \leq 0$.

The desired cut can be found in $O(m)$ time using graph search. It is a little more complicated to extract a maximum flow. The time to do so is $O(m \log n)$ using a dynamic tree data structure [22].

The arc-balancing algorithm is generic in the sense that the order of moves is unspecified. We can restrict the order of moves in a simple way that leads to a good theoretical time bound. We call the resulting method the *round-robin arc-balancing algorithm*. It consists merely of looping over a fixed list of the arcs, applying a move to each active arc, until an entire pass results in no $(1/n^2)$-moves. (Alternatively, we can stop when the active section has positive excesses summing to less than one or negative excesses summing to more than minus one.) The round-robin algorithm is particularly well-suited to massively parallel implementation.

Theorem 1. *The round-robin algorithm stops after $O(n^2 \log(nU))$ passes over the arcs, taking $O(m)$ time per pass, for a total time of $O(n^2 m \log(nU))$.*

The bound on passes in Theorem 1 is tight within a constant factor.

Theorem 2. *For any n and U, there is a network with n vertices and arc capacities bounded by U on which the round-robin algorithm does $\Omega(n^2 \log U)$ passes and $\Omega(n^3 \log U)$ moves.*

The example used to prove Theorem 2 shows that arc-balancing has trouble moving flow along long paths. Essentially the same example is bad for both the two-arc-balancing and the star-balancing round-robin algorithms for finding a λ-balanced flow on a parametric network. This motivates looking for longer-range balancing methods. We can use a modified form of star-balancing in place of arc-balancing in the ordinary maximum flow algorithm: the main change is that the excess of the vertex at the center of the star must be balanced against all those of the adjacent vertices. We have developed a linear-time algorithm for balancing a path of arbitrary length, and an $O(n \log^2 n)$-time algorithm for balancing an n-vertex tree. (That is, given an initial pseudoflow, modifying it so that no arc-balancing moves are possible.) Both of these methods can be adapted to work for the bipartite parametric problem. These methods could be used as steps in computations on more-general networks, but so far we have not found a way to obtain a better overall running time for the ordinary maximum flow problem, for example.

6 Remarks

As we have discussed in this paper, balancing can be used as a conceptual and algorithmic tool for finding ordinary and parametric maximum flows and minimum cuts. Other types of applications may be able to use balancing ideas. Here we only mention one potential application in bioinformatics. In protein classification, we are given a graph whose edges may have weights. Vertices represent proteins; edges represent associational strengths. Certain proteins are classified by function. We wish to classify the unclassified proteins based on their associational strengths with classified ones. One possible approach is to compute, for an unclassified protein and for each group of classified proteins, an arc-balanced pseudoflow with the classified proteins as sources, and to regard the final excess at the unclassified protein as a measure of its associativity. In applying this idea it makes sense to make the edge capacities increasing functions of their weights and decreasing functions of their distance from classified proteins. See Singh et al. [20] for an approach to this problem using a similar method.

The balanced flow notion offers a succinct description of the solution to a parametric maximum flow or minimum cut problem, and it leads to novel algorithms for solving both parametric and ordinary maximum flow problems that use balancing as a local operation. Although our balancing algorithms are very simple, their time bounds are not competitive with those of existing algorithms. Nevertheless, we are optimistic that our techniques will lead to improved algorithms that are competitive in practice, if not in theory, with existing algorithms. Indeed, preliminary experiments on large real-world datasets suggest that the star-balancing algorithm for bipartite parametric problems computes solutions for all parameter values within a factor of two of the time taken by the fastest ordinary max-flow code [6] to compute a solution for a single parameter value. We intend to do additional experimental work. The idea of balancing in the context of maximum flow offers a rich space for research.

Acknowledgement. We thank Robert Schreiber and Andrew Goldberg for helpful discussions. We thank Qi Feng, a summer intern at HP Labs, for work on the product selection project, initially using maximum flow methods.

References

1. R. K. Ahuja, J. B. Orlin, and R. E. Tarjan. Improved time bounds for the maximum flow problem. *SIAM Journal on Computing*, 18:939–954, 1989.
2. B. Awerbuch and F. T. Leighton. A simple local-control approximation algorithm for multicommodity flow. In *Proc. FOCS*, pages 459–468. IEEE, 1993.
3. B. Awerbuch and T. Leighton. Improved approximation algorithms for the multicommodity flow problem and local competitive routing in dynamic networks. In *STOC*, pages 487–496, 1994.
4. M. L. Balinski. On a selection problem. *Management Science*, 17(3):230–231, 1970.
5. M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.

6. B. V. Cherkassky and A. V. Goldberg. On implementing the push-relabel method for the maximum flow problem. *Algorithmica*, 19(4):390–410, 1997.
7. E. A. Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Math. Dokl.*, 11:1277–1280, 1970.
8. M. J. Eisner and D. G. Severance. Mathematical techniques for efficient record segmentation in shared databases. *Journal of the ACM*, 23(4):619–635, 1976.
9. G. W. Flake, R. E. Tarjan, and K. Tsioutsiouliklis. Graph clustering and minimum cut trees. *Internet Mathematics*, 1(4):385–408, 2005.
10. S. Fujishige. A maximum flow algorithm using MA ordering. *Operations Research Letters*, 31:176 – 178, 2003.
11. G. Gallo, M. D. Grigoriadis, and R. E. Tarjan. A fast parametric maximum flow algorithm and applications. *SIAM J. Computing*, 18(1):30–55, 1989.
12. A. Goldberg. Private communication, 2006.
13. A. Goldberg and R. Tarjan. A new approach to the maximum flow problem. *Journal of the ACM*, 35(4):921–940, 1988.
14. A. V. Goldberg and S. Rao. Beyond the flow decomposition barrier. *Journal of the ACM*, 45(5):783–797, 1998.
15. D. Gusfield and C. Martel. A fast algorithm for the generalized parametric minimum cut problem and applications. *Algorithmica*, 7:499–519, 1992.
16. D. Hochbaum. Selection, provisioning, shared fixed costs, maximum closure, and implications on algorithmic methods today. *Management Science*, 50(6):709–723, 2004.
17. J. Mamer and S. Smith. Optimizing field repair kits based on job completion rate. *Management Science*, 28(11):1328–1333, 1982.
18. Y. Matsuoka and S. Fujishige. Practical efficiency of maximum flow algorithms using MA orderings and preflows. *J. Oper. Res. Soc. of Japan*, 48:297–307, 2005.
19. S. T. McCormick. Fast algorithms for parametric scheduling come from extensions to parametric maximum flow. *Operations Research*, 47:744–756, 1999.
20. E. Nabieva, K. Jim, A. Agarwal, B. Chazelle, and M. Singh. Whole-proteome prediction of protein function via graph-theoretic analysis of interaction maps. *Bioinformatics*, 21:i302–i310, 2005.
21. J. M. W. Rhys. A selection problem of shared fixed costs and network flows. *Management Science*, 17(3):200–207, 1970.
22. D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(1):362–391, 1983.
23. H. Stone. Critical load factors in two-processor distributed systems. *IEEE Trans. Software Engineering*, 4:254–258, 1978.
24. B. Zhang, J. Ward, and Q. Feng. A simultaneous parametric maximum flow algorithm for finding the complete chain of solutions. Technical report, HP Labs, 2004. `http://www.hpl.hp.com/techreports/2004/HPL-2004-189.html`.
25. B. Zhang, J. Ward, and Q. Feng. Simultaneous parametric maximum flow algorithm with vertex balancing. Technical report, HP Labs, 2005. `http://www.hpl.hp.com/techreports/2005/HPL-2005-121.html`.

Out-of-Order Event Processing in Kinetic Data Structures

Mohammad Ali Abam[1,⋆], Pankaj K. Agarwal[2,⋆⋆], Mark de Berg[1,⋆], and Hai Yu[2,⋆⋆]

[1] Department of Mathematics and Computing Science,
TU Eindhoven, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
mabam@win.tue.nl, mdberg@win.tue.nl
[2] Department of Computer Science, Duke University, Durham, NC 27708, USA
pankaj@cs.duke.edu, fishhai@cs.duke.edu

Abstract. We study the problem of designing kinetic data structures (KDS's for short) when event times cannot be computed exactly and events may be processed in a wrong order. In traditional KDS's this can lead to major inconsistencies from which the KDS cannot recover. We present more robust KDS's for the maintenance of two fundamental structures, kinetic sorting and tournament trees, which overcome the difficulty by employing a refined event scheduling and processing technique. We prove that the new event scheduling mechanism leads to a KDS that is correct except for finitely many short time intervals. We analyze the maximum delay of events and the maximum error in the structure, and we experimentally compare our approach to the standard event scheduling mechanism.

1 Introduction

The recent advances in sensing and tracking technology have led researchers to investigate the problem of maintaining various geometric attributes of a set of moving objects, as evident from a large body of literature on kinetic geometric algorithms. Basch *et al.* [6] introduced the *kinetic data structure (KDS)* framework for designing and analyzing algorithms for continuously moving objects. The KDS framework consists of two parts: a combinatorial description of the attribute, and a set of *certificates* (each of which is a predicate) with the property that as long as the certificates remain valid, the maintained attribute remains valid. It is assumed that each object follows a known trajectory so that one can compute the failure time of each certificate. Whenever a certificate fails—we call this an *event*—the KDS must be updated. This involves updating the attribute and the set of certificates. The KDS then remains valid until the next event has to be processed, and so on.

To be able to process each event at the right time, a global event queue Q is maintained to process the events in the right (chronological) order. This is a priority queue on

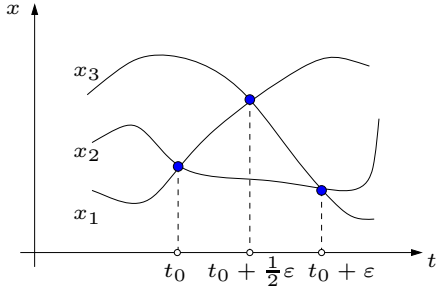
⋆ M.A. was supported by the Netherlands' Organisation for Scientific Research (NWO) under project no. 612.065.307. M.d.B. was supported by the Netherlands' Organisation for Scientific Research (NWO) under project no. 639.023.301.

⋆⋆ P. A. and H. Y. were supported 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.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 624–635, 2006.

the events, with the priority of an event being its failure time. Unfortunately, the event scheduling is not as easy as it seems. Suppose that a new certificate arises due to some event. When the failure time of the certificate lies in the past we should not schedule it, and when it lies in the future we should. But what if the event time is equal to the current time t_{curr}? In such a degenerate situation one has to be very careful to avoid an infinite loop. A more serious problem arises when the event times are not computed exactly. This will indeed be the case if the trajectories are polynomials of high degree or more complex curves. As a result, events may be processed in a wrong order, or we may fail to schedule an event because we think it has already taken place. This in turn may not only lead to serious errors in the geometric attribute the KDS is maintaining but also cause the algorithm to crash.

As a concrete example, consider the kinetic sorting problem: maintain the sorted order of a set S of points moving on the real line. We store S in a sorted array $A[1..n]$. For each $1 \leq i < n$ there is a certificate $\big[A[i] < A[i+1]\big]$. Whenever $A[j] = A[j+1]$, we have a certificate failure. At such an event we swap $A[j]$ and $A[j+1]$. Furthermore, at most three new certificates arise: $\big[A[j-1] < A[j]\big]$, $\big[A[j] < A[j+1]\big]$, and $\big[A[j+1] < A[j+2]\big]$. We compute the failure time of each of them, based on our knowledge of their current motions, and insert the failure times that are not in the past into the event queue Q. Some certificates may also disappear because the two points involved are no longer neighbors; they have to be deleted from Q. Now suppose that due to errors in the computed failure times the difference between the exact and the computed failure time of each certificate can be ε, for some $\varepsilon > 0$. Consider the three moving points x_1, x_2 and x_3 whose trajectories in the tx-plane are depicted in Figure 1. Table (i) shows what happens when we can compute the exact failure times. Table (ii) shows what happens when the computed failure times of the certificates $\big[x_1 < x_2\big]$, $\big[x_1 < x_3\big]$, and $\big[x_2 < x_3\big]$ are $t_0 + \varepsilon$, $t_0 + \frac{3}{2}\varepsilon$, and t_0 respectively: the KDS is not just temporarily incorrect, but gets into an incorrect state from which it never recovers.



t_{curr}	List	Certificates	Failure Time
0	x_1, x_2, x_3	$[x_1 < x_2]$ $[x_2 < x_3]$	t_0 $t_0 + \varepsilon$
t_0	x_2, x_1, x_3	$[x_1 < x_3]$	$t_0 + \frac{1}{2}\varepsilon$
$t_0 + \frac{1}{2}\varepsilon$	x_2, x_3, x_1	$[x_2 < x_3]$	$t_0 + \varepsilon$
$t_0 + \varepsilon$	x_3, x_2, x_1		

(i) without error

t_{curr}	List	Certificates	Failure Time
0	x_1, x_2, x_3	$[x_1 < x_2]$ $[x_2 < x_3]$	$t_0 + \varepsilon$ t_0
t_0	x_1, x_3, x_2	$[x_1 < x_3]$	$t_0 + \frac{3}{2}\varepsilon$
$t_0 + \frac{3}{2}\varepsilon$	x_3, x_1, x_2		

(ii) with error

Fig. 1. An example that numerical errors in the event times may cause errors in the KDS. Left: the trajectories of the points. Right: the status of the KDS at various times of execution.

This is a serious problem for the applicability of the KDS framework in practice. The goal of our paper is to address this issue: is it possible to do the event scheduling and processing in such a way that the KDS is more robust under errors in the computation of

event times? The KDS may process the events in a wrong order and thus may maintain a wrong geometric attribute from time to time, but we would like the KDS to detect these errors and fix them quickly.

Related work. The KDS framework [6] is a widely used algorithmic method for modeling motion. It has been applied to maintain a variety of geometric attributes of moving objects, including convex hull [5, 6], closest pair [6, 7], range searching structures [1, 2], extent measures [3, 4], and much more. See the survey by Guibas [13] and the references therein. There has been much work on modeling motion in many other fields as well, including computer graphics, spatial databases, robotics, and sensor networks.

There is a large body of work on robust computations in geometric algorithms [11, 20, 21], including geometric software libraries [8, 10]. The goal there is to implement various geometric primitives in a robust manner, including *predicates*, which test the sign of an arithmetic expression (e.g., ORIENTATION and INCIRCLE predicates), and *constructions*, which compute the value of an arithmetic expression (e.g., computing the intersection of two lines). There are two broad paradigms. The first approach, exact computation, performs computation with enough precision to ensure predicates can be evaluated correctly. One can of course use this approach to do the event scheduling in a KDS—see the papers by Guibas and co-workers [14, 15] for some research in this direction. Unfortunately, in practice a significant portion of the running time of a KDS is spent on computing certificate failure times [14, 16], and exact computation may lead to unacceptable performance (for example, when many events are very close to each other). An alternative is *controlled perturbation* [17, 12], where we perturb the initial positions of the objects by some amount δ so that with high probability the roots of all pertinent functions are sufficiently far away from each other. This does not seem to work well on KDS's because the large number of kinetic events makes the required perturbation bound δ fairly large.

Recently, Milenkovic and Sacks [19] studied the computation of arrangements of x-monotone curves in the plane using a plane sweep algorithm, under the assumption that intersection points of curves cannot be computed exactly. For infinite curves this boils down to the kinetic sorting problem, because one has to maintain the sorted order of the curves along the sweep line. In fact, our KDS for the kinetic sorting problem is very similar to their algorithm. The main difference is in the subroutine to compute intersection points of curves which we assume to have available; this subroutine is stronger than the subroutine they assume—see Section 2 for details. This allows us to ensure that we never process more events than the number of actual crossings, whereas Milenkovic and Sacks may process a quadratic number of events in the worst case even when there is only a linear number of crossings. The main difference between our and their papers, however, lies in the different view on the problem: since we are looking at the problem from a KDS perspective, we are especially interested in the delay of events and the error in the output for each snapshot of the motion, something that were not studied in [19]. Moreover, we study other KDS problems as well.

Our results. The main problem we face when event times are not computed exactly is that events may be processed in a wrong order. We present KDS's that are robust against this out-of-order processing, including kinetic sorting and kinetic tournaments.

Our algorithms are *quasi-robust* in the sense that the maintained attribute of the moving objects will be correct for most of the time, and when it is incorrect, it will not be far from the correct attribute. For the kinetic sorting problem, we obtain the following results:

- We prove that the KDS can only be incorrect when the current time lies inside an event interval.
- We prove that an event may be processed too late, but not by more than $O(n\varepsilon)$ time. This bound is tight in the worst case.
- We prove bounds on the geometric error of the structure—the maximum distance between the i-th point in the maintained list and the i-th point in the correct list—that depend on the velocities of the points.

We obtain similar results for kinetic tournaments. As a by-product of our approach, degeneracy problems (how to deal with multiple events occurring simultaneously) arising in traditional KDS algorithms naturally disappear, because our KDS no longer cares about in which order these simultaneous events are processed.

We have implemented the robust sorting and tournament KDS algorithms and tested them on a number of inputs, including highly degenerate ones. Our sorting algorithm works very well on these inputs: of course it does not get stuck and the final list is always correct (after all, this is what we proved), but the maximum delay is usually much less than the worst-case bound suggests (namely $O(\varepsilon)$ instead of $\Theta(n\varepsilon)$). This is in contrast to the classical KDS, which either falls into an infinite loop or misses many kinetic events along the way and maintains a list that deviates far from the true sorted list both geometrically and combinatorially. Our kinetic tournament algorithm is also robust and reduces the error by orders of magnitude.

Because of lack of space, many details including the proofs of lemmas and theorems are omitted from this abstract. We refer the reader to the full version for the details.

2 Our Model

We describe our model for computing the event times of certificates—how we cope with handling the events being processed out of order. Each certificate c is a predicate, and there is a characteristic function $\chi_c : \mathbb{R} \to \{1, 0, -1\}$ so that $\chi_c(t) = 1$ if c is true at time t, -1 if it is false at time t, and 0 if c is switching from being true to false, or vice-versa, at time t. The values of t at which χ_c is 0 are the event times of c. In our applications, $\chi_c(t) = \mathrm{sign}(\varphi_c, t)$ for some continuous function φ_c. For example, if $x(t), y(t)$ are two points, each moving in $\mathbb{R}^1$, then $\varphi_c(t) := y(t) - x(t)$ for the certificate $c := [x < y]$. For simplicity, we assume in this extended abstract that $\mathrm{sign}(\varphi_c, t) = 0$ for a finite number, s, of values of t.

We assume that the trajectory of each object is explicitly described by a function of time, which means in our applications that the function φ_c is also explicitly described, and that event times can be computed by computing the roots of the function φ_c. This is what is being done in traditional KDS's. In order to model the inaccuracy in computing event times, we fix a parameter $\varepsilon > 0$, which will determine the accuracy of the root computation. We assume there is a subroutine, denoted by $\textsc{crop}\big(f(t)\big)$, to compute the roots of a function $f(t)$, whose output is as follows:

(i) a set of disjoint, open *event intervals* $I_1, \cdots, I_k$, where $|I_i| \leq \varepsilon$ for each i, that cover all roots of $f(t)$;
(ii) the sign of $f(t)$ between any two consecutive event intervals;

For polynomial functions, Descartes' sign rule [9] and Sturm sequences [18] are standard approaches to implement such a subroutine. Observe that if $f(t)$ does not have any roots, then CROP$\big(f(t)\big)$ will tell us so. This is where our subroutine is more powerful than the subroutine of Milenkovic and Sacks [19], and this is why we can ensure that we only handle events if there is a real crossing of trajectories.

In our applications, we can ignore the event intervals whose two endpoints have the same sign and assume (pretend) that the sign does not change during such an interval. We will use $\mathcal{I} = \langle I_1, \ldots I_k \rangle$ to denote the set of open event intervals whose two endpoints have different signs, and ask CROP to only output these event intervals. Let λ_j (resp. ρ_j) denote the left (resp. right) endpoint of I_j, i.e., $I_j = (\lambda_j, \rho_j)$. As we will see below, we will schedule events at the right endpoints of these intervals.

We assume that CROP is deterministic: it always returns the same result when run on the same function. We also assume that tests as to whether a given time t lies inside an event interval computed by CROP are exact. We use t_{curr} to denote the current time of the KDS, which is the maximum computed event time over all processed events.

The pseudo-code of the algorithm for computing the failure time of a certificate c is given below. We first find the index, last, of the last event interval that lies to the left of t_{curr}. By our model, the sign of the function $\varphi_c(t)$ remains the same at all times during the interval $[\rho_{\mathsf{last}}, \lambda_{\mathsf{last}+1}]$; inside an event interval, we pretend that the sign is the same as at its left endpoint and that it changes at its right endpoint. Therefore, if t_{curr} lies in this interval, we can assume $\chi_c(t_{\mathsf{curr}}) = \chi_c(\rho_{\mathsf{last}})$.

Algorithm EVENTTIME (c)
1. $\mathcal{I} := \langle I_1 = (\lambda_1, \rho_1), \ldots, I_k = (\lambda_k, \rho_k) \rangle \leftarrow$ CROP (φ_c)
2. $\rho_0 \leftarrow -\infty$; $\rho_{k+1} \leftarrow +\infty$
3. last $\leftarrow$number of intervals in $\mathcal{I}$ to the left of t_{curr}
4. **if** $\chi_c(\rho_{\mathsf{last}}) = -1$ **then** return ρ_{last} **else** return $\rho_{\mathsf{last}+1}$

Note that if $\chi_c(\rho_{\mathsf{last}}) = -1$, then the event time returned by EVENTTIME(c) (i.e., ρ_{last}) is in the past. Still we insert this event into the queue because the certificate c is not valid and thus the combinatorial structure of the KDS is not correct. Apparently we missed an event, which we must still handle. As we will see in the next section, when we handle such an event in the past, we do not reset t_{curr}: the time t_{curr} will always be the maximum of the computed event times over all processed events. Finally, note that the above procedure has the following properties: If it returns a finite value ρ_i, then

(I1) ρ_i is the right endpoint of an event interval;
(I2) the certificate c is valid at λ_i and invalid at ρ_i, i.e., $\chi_c(\lambda_i) = 1$ and $\chi_c(\rho_i) = -1$.

3 Kinetic Sorting

Let S be a set of n points moving continuously on the real line. The value of a point $x \in S$ is a continuous function $x(t)$ of time t. We define $S(t) = \{x(t) : x \in S\}$. In the kinetic sorting problem, we want to maintain the sorted order of S during the motion.

The algorithm. As in the standard algorithm, we maintain an array A that stores the points in S. The events are stored in a priority queue Q, called global event queue. The certificates are standard as well: the certificate $c := \lceil x < y \rceil$ belongs to the current certificate set of the KDS if $x = A[k]$ and $y = A[k+1]$ for some k. We call these $n-1$ certificates *active*. We need the following notation regarding failure times.

$t_{\mathsf{cp}}(x, y)$: the computed failure time[1] of certificate $\lceil x < y \rceil$
$t_{\mathsf{pr}}(x, y)$: the time at which the failure of $\lceil x < y \rceil$ is actually processed

The new kinetic sorting algorithm is described below. The major difference with the standard algorithm is that we use the algorithm EVENTTIME to compute the failure time of a certificate. As such, unlike the standard algorithm, our algorithm may process events in the past. Note that t_{curr} remains unchanged when this happens (see line 4).

Algorithm KINETICSORTING
1. $t_{\mathsf{curr}} \leftarrow -\infty$; Initialize A and Q.
2. **while** $Q \neq \emptyset$
3. **do** $c : \lceil x < y \rceil \leftarrow$ DELETEMIN(Q)
4. $t_{\mathsf{curr}} \leftarrow \max\{t_{\mathsf{curr}}, t_{\mathsf{cp}}(x, y)\}$
5. Swap x and y (which are adjacent in A).
6. Remove from Q all certificates that become inactive.
7. $\mathcal{C} \leftarrow$ set of new certificates that become active.
8. **for** each $c : \lceil a < b \rceil \in \mathcal{C}$
9. **do** $t_{\mathsf{cp}}(a, b) \leftarrow$ EVENTTIME (c)
10. **if** $t_{\mathsf{cp}}(a, b) \neq \infty$
11. **then** Insert $\lceil a < b \rceil$ into Q, with $t_{\mathsf{cp}}(a, b)$ as failure time.

Basic properties. The status of the KDS *at time* t is defined as the status of the KDS after all events whose processing times are at most t have been processed. In the kinetic sorting problem, the status refers to the maintained array A. We say that a point x *precedes* a point y in the maintained array A if $x = A[k]$ and $y = A[l]$ for some $l > k$. If $l = k + 1$, then x *immediately precedes* y.

Since events may be processed in a wrong order, the above KDS could perhaps get into an infinite loop. However, if a certificate c is processed by the algorithm (line 5) at time t_0 and c becomes active again at time t_0, then EVENTTIME ensures that the failure time of c is in the future. This implies that the algorithm does not go into an infinite loop. We can also show the KDS almost always maintains a correctly sorted list in A.

Lemma 1. *If x immediately precedes y in A at time t_{curr}, then either*

(i) $x(t_{\mathsf{curr}}) < y(t_{\mathsf{curr}})$*, which means the order is correct, or*
(ii) $t_{\mathsf{cp}}(x, y) - \varepsilon \leq t_{\mathsf{curr}} < t_{\mathsf{cp}}(x, y)$*, where $t_{\mathsf{cp}}(x, y)$ is the computed failure time of the certificate $\lceil x < y \rceil$ currently stored in Q. Furthermore, $x(\gamma) = y(\gamma)$ for some $\gamma \in (t_{\mathsf{curr}} - \varepsilon, t_{\mathsf{curr}}]$.*

[1] This is a slight abuse of notation, because points can swap more than once, so the same certificates can fail multiple times. It will be convenient to treat these certificates as different. Formally we should write $t_{\mathsf{cp}}((x, y), t_{\mathsf{curr}})$ for the failure time of the certificate $\lceil x < y \rceil$ computed by the KDS at time t_{curr}. Since this is always clear from the context we omit the time parameter.

The following theorem is a straightforward consequence of the above lemma.

Theorem 1. *The ordering maintained by the kinetic sorting algorithm is correct except during at most E time intervals of length $\leq \varepsilon$, where E is the number of collisions of points in S over the entire motion.*

Delay of kinetic events. Theorem 1 shows that the ordering may be incorrect only near the event times, but many "event intervals" may cascade and thus an event may not be processed for a long time, thereby maintaining a wrong ordering for a long time. Next we bound the maximum delay of an event. The bound holds when every pair of points swaps at most s times for some parameter s.

Theorem 2. *Suppose that the trajectories of any two points $x, y \in S$ intersect at most s times. If an event with computed failure time $t_{\mathsf{cp}}(x, y)$ gets processed at time $t_{\mathsf{pr}}(x, y)$, then we have $t_{\mathsf{pr}}(x, y) - t_{\mathsf{cp}}(x, y) < (n-2)s \cdot \varepsilon + \varepsilon$. The bound is tight in the worst case.*

Remark. As t_{curr} advances, it might happen that an event interval (λ_i, ρ_i) corresponding to some certificate c becomes obsolete before it can be processed. This happens when by the time c becomes active, t_{curr} has already advanced past the next event interval $(\lambda_{i+1}, \rho_{i+1})$ of c. The above theorem implies that an event with computed failure time $t_{\mathsf{cp}}(x, y)$ either becomes obsolete or gets processed by the KDS before $t_{\mathsf{cp}}(x, y) + (n-2)s \cdot \varepsilon + \varepsilon$.

Error bounds. We turn our attention to the "error" in the array A. Combinatorially, Lemma 1 implies that if there are k event intervals containing t_{curr}, then the array A at time t_{curr} can be decomposed into at most $k+1$ (contiguous) subarrays, each of which is in sorted order. Next we discuss how far the maintained order can be from the correct order geometrically. In particular, we present a bound on the maximum distance between two points that are in the wrong order in the array and on how far away the k-th point in the maintained order—that is, the point $A[k]$—can be from the true point of rank k.

Theorem 3. *Let $\langle y_1, \cdots, y_n \rangle$ and $\langle z_1, \cdots, z_n \rangle$ be the sequence maintained by the algorithm and the correctly sorted sequence at some given time t_{curr}, respectively. Let V_{max} be the maximum velocity of any point in S over the time interval $[t_{\mathsf{curr}} - \varepsilon, t_{\mathsf{curr}}]$. Then for any $1 \leq i < j \leq n$, we have: (i) $y_i(t_{\mathsf{curr}}) - y_j(t_{\mathsf{curr}}) \leq n\varepsilon \cdot V_{\mathsf{max}}$, and (ii) $|y_i(t_{\mathsf{curr}}) - z_i(t_{\mathsf{curr}})| \leq n\varepsilon \cdot V_{\mathsf{max}}$.*

4 Kinetic Tournaments

A kinetic tournament [6] is a KDS that maintains the maximum of a set S of moving points in $\mathbb{R}$ by maintaining a tournament tree T over S. Each interior node u of T has a certificate of the form $\big[x < y\big]$, where $x, y \in S$ are the two points stored at the children of u, and y is also currently stored at u. To handle events, we need a subroutine that compares two points at time t_{curr} in a way that is consistent with EVENTTIME.

Algorithm COMPUTEMAX(x, y)
1. $\mathcal{I} := \langle I_1 = (\lambda_1, \rho_1), \ldots, I_k = (\lambda_k, \rho_k) \rangle \leftarrow$ CROP $(x(t) - y(t))$
2. $\rho_0 \leftarrow -\infty$
3. last $\leftarrow$number of intervals in $\mathcal{I}$ to the left of t_{curr}
4. **if** $\mathrm{sign}(x(\rho_{\mathsf{last}}) - y(\rho_{\mathsf{last}})) = 1$ **then** return x **else** return y

In the algorithm below, p_u denotes the point stored at node u, with $\mathrm{parent}(root) =$ **nil**.

Algorithm KINETICTOURNAMENT
1. $t_{\mathsf{curr}} \leftarrow -\infty$; Initialize T and Q.
2. **while** $Q \neq \emptyset$
3. **do** $c : [x < y] \leftarrow$ DELETEMIN(Q)
4. $t_{\mathsf{curr}} \leftarrow \max\{t_{\mathsf{curr}}, t_{\mathsf{cp}}(x, y)\}$
5. $u \leftarrow$ the node at which the certificate c fails.
6. **while** $u \neq$ **nil**
7. **do** Let z_1 and z_2 be the points stored at u's children.
8. $p_u \leftarrow$COMPUTEMAX(z_1, z_2); $u \leftarrow \mathrm{parent}(u)$
9. Remove from Q all certificates that become inactive.
10. $\mathcal{C} \leftarrow$ set of new certificates that become active.
11. **for** each $c : [a < b] \in \mathcal{C}$
12. **do** $t_{\mathsf{cp}}(a, b) \leftarrow$EVENTTIME (c)
13. **if** $t_{\mathsf{cp}}(a, b) \neq \infty$
14. **then** Insert $[a < b]$ into Q, with $t_{\mathsf{cp}}(a, b)$ as failure time.

The algorithm can be shown to maintain the following invariant:

Invariant: After an event has been processed, the point p_u stored at a node u is always one of the points stored at its children. Moreover, p_u is either the correct current maximum of the two children, or the trajectories of points stored at the two children cross during the period $(t_{\mathsf{curr}} - \varepsilon, t_{\mathsf{curr}}]$.

Following standard KDS terminology, we call an event in the kinetic tournament *external* when the maximum changes.

Lemma 2. *If the maximum maintained by the algorithm is incorrect at time t_{curr}, there must have been an external event during the period $(t_{\mathsf{curr}} - \varepsilon, t_{\mathsf{curr}}]$.*

The following result is an immediate consequence of Lemma 2.

Theorem 4. *The maximum maintained by the kinetic tournament is correct except during at most E time intervals of length $\leq \varepsilon$, where E is the number of external events.*

We now turn our attention to the geometric error of our KDS—the difference in value between the point stored in the root of the kinetic tournament tree and the real maximum—as a function of the maximum velocity. Interestingly, the geometric error is much smaller than in the sorting KDS, because it depends on the depth of the tournament tree. The following theorem makes this precise. It follows from the invariant, since the error between two consecutive nodes on the path from the root to the real maximum can be at most $\varepsilon \cdot 2V_{\mathsf{max}}$.

Theorem 5. *Let x denote the point stored in the root of the kinetic tournament tree at some time t_{curr}, and let y denote the point with the maximum value at time t_{curr}. Then $y(t_{\mathsf{curr}}) - x(t_{\mathsf{curr}}) \leq 2\varepsilon\lceil \log n \rceil \cdot V_{\mathsf{max}}$, where V_{max} is the maximum velocity of any point in S over the time interval $[t_{\mathsf{curr}} - \varepsilon, t_{\mathsf{curr}}]$.*

5 Experiments

We implemented our robust kinetic-sorting and kinetic-tournament algorithms and compared them to the traditional KDS event-scheduling approach. The programs are written in C++ and run in the Linux 2.4.20 environment.

Input data. We used the following synthetic datasets in our experiments, as illustrated in Figure 2. The inputs are low-degree motions because we have not yet implemented a full-fledged CROP procedure, and it becomes easier for us to compute delays of the events. Nonetheless, these inputs already cause trouble to traditional KDS's and are sufficient to illustrate the effectiveness of our algorithms.

- GRIDS: linear trajectories in whose dual points form a uniform grid;
- PARABOLA: congruent parabolic trajectories with apexes on a grid in the tx-plane;
- RANDDC: linear trajectories whose dual points are randomly distributed in a disk;
- RANDCR: linear trajectories whose dual points are randomly distributed on a circle.

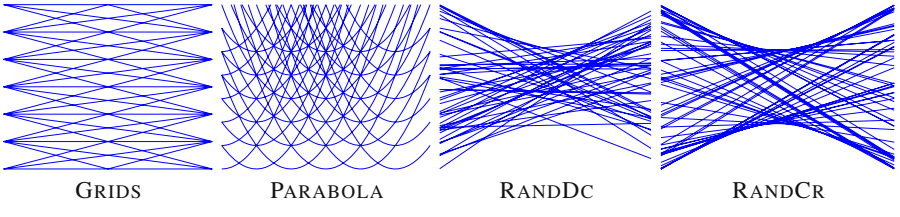
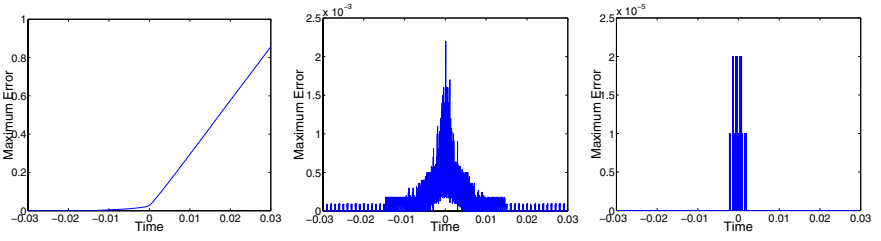


Fig. 2. Various datasets. The figures depict trajectories in tx-plane, after an appropriate scaling.

Kinetic sorting. We tested the kinetic sorting algorithms on the first three types of input data. All experiments were run on inputs of size 900. We first observed the behavior of the traditional kinetic sorting algorithm, which uses floating point arithmetic. In a few instances, the algorithm went into an infinite loop because of simultaneous events. As for the correctness of the maintained structures, the traditional KDS was very fragile: it quickly ran into noticeable errors and was unable to recover from these errors (see Figures 3 (1), 4 (1), and 5 (1)). The reason is that some events that should have been scheduled into the global queue were discarded by the KDS because their computed event times happened to lie in the past.

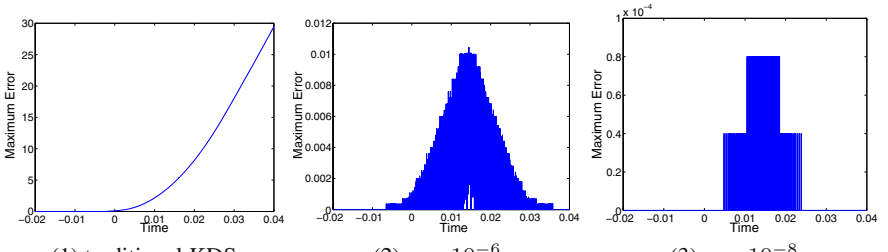
We now turn our attention to the geometric error of the structures maintained by our robust kinetic sorting algorithm. We measure the error of the sorting KDS at time t by $\mathrm{err}(t) = \max_i |y_i(t) - z_i(t)|$, where $\langle y_1, \cdots, y_n \rangle$ and $\langle z_1, \cdots, z_n \rangle$ are the sequence maintained by the KDS and the correctly sorted sequence at time t. In Figures 3- 5



(1) traditional KDS (2) $\varepsilon = 10^{-6}$ (3) $\varepsilon = 10^{-8}$

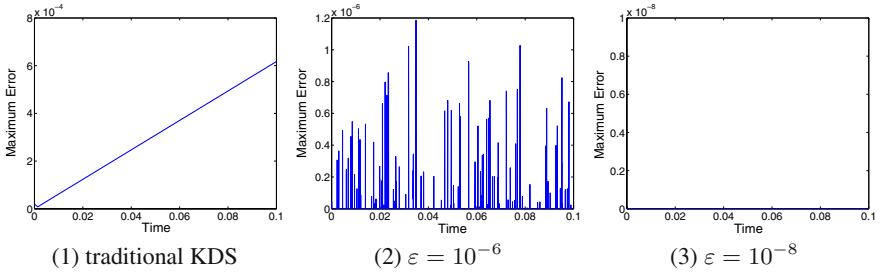
Fig. 3. Error for kinetic sorting on GRID input; y-axis scales are different

we plot $\mathrm{err}(t)$ as t varies, by measuring $\mathrm{err}(t)$ every other 10^{-7} seconds. Note the different scales on the error axis in these figures. As can be seen, while the traditional KDS quickly ran into serious errors and was never able to recover, our robust KDS maintained a rather small error all the time. Observe that the error of the robust KDS reduces as the precision of the CROP procedure increases. We also tested the algorithm on a number of larger inputs, and the error remained roughly the same.



(1) traditional KDS (2) $\varepsilon = 10^{-6}$ (3) $\varepsilon = 10^{-8}$

Fig. 4. Error for kinetic sorting on PARABOLA input; y-axis scales are different



(1) traditional KDS (2) $\varepsilon = 10^{-6}$ (3) $\varepsilon = 10^{-8}$

Fig. 5. Error for kinetic sorting on RANDDC input; y-axis scales are different.

We also studied how long a kinetic event could be delayed before it is eventually processed in the kinetic sorting algorithm—see Table 1. Some events were actually processed earlier instead of being delayed; we regard delays of these events as zero.

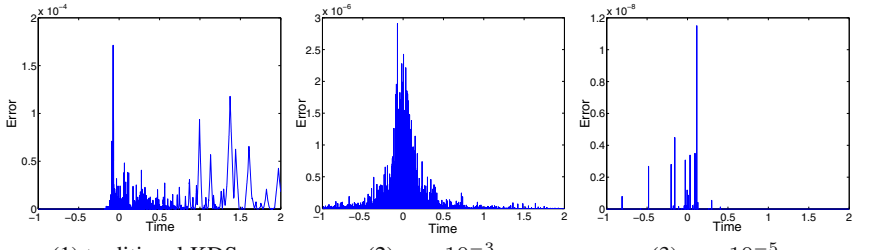
It can be seen that the RMS of the delays are always very small for all inputs. As to the maximum delay, we only observed one instance in the first two types of inputs in which some events are delayed by about 2ε; in all other cases, the maximum delay never exceeds ε, which is far below the rather contrived worst-case bound in Theorem 2.

Table 1. Delay of events in kinetic sorting

Precision of CROP	GRIDS		PARABOLA		RANDDC	
	RMS	Max	RMS	Max	RMS	Max
$\varepsilon = 10^{-6}$	0.48×10^{-6}	2.00×10^{-6}	0.37×10^{-6}	1.00×10^{-6}	0.42×10^{-6}	1.00×10^{-6}
$\varepsilon = 10^{-7}$	0.43×10^{-7}	1.00×10^{-7}	0.37×10^{-7}	1.00×10^{-7}	0.42×10^{-7}	1.00×10^{-7}
$\varepsilon = 10^{-8}$	0.42×10^{-8}	1.00×10^{-8}	0.39×10^{-8}	1.00×10^{-8}	0.41×10^{-8}	1.00×10^{-8}

Kinetic tournament. We tested the kinetic tournament algorithms on the RANDCR data as they tend to have a large number of external events. Since the kinetic tournament algorithms are less sensitive to simultaneous events than kinetic sorting, we artificially increased the error in computing the event times so as to cause noticeable geometric errors in the tested algorithms. Specifically, in the traditional KDS we round the event times to the precision of 10^{-5}, and in the robust KDS we vary the error from 10^{-3} to 10^{-5}.

We first noticed that the traditional kinetic tournament algorithm did not go into an infinite loop; this is because the kinetic events are always "pushed" up in the tree. However, as for the geometric error, one can see from Figure 6 (1) that the KDS maintains a rather inaccurate maximum during the motion. In contrast, the geometric errors in our robust KDS are smaller by orders of magnitudes, even though the event time computation is less precise than in the traditional KDS.



(1) traditional KDS (2) $\varepsilon = 10^{-3}$ (3) $\varepsilon = 10^{-5}$

Fig. 6. Geometric error of the kinetic tournament on a RANDCR input of size 10000

References

1. P. K. Agarwal, L. Arge, and J. Erickson. Indexing moving points. *J. Comput. Syst. Sci.*, 66:207–243, 2003.
2. P. K. Agarwal, J. Gao, and L. J. Guibas. Kinetic medians and kd-trees. In *Proc. 10th European Sympos. on Algorithms*, pages 5–16, 2002.

3. P. K. Agarwal, L. J. Guibas, J. Hershberger, and E. Veach. Maintaining the extent of a moving point set. *Discrete Comput. Geom.*, 26(3):353–374, 2001.
4. P. K. Agarwal, S. Har-Peled, and K. R. Varadarajan. Approximating extent measures of points. *J. ACM*, 51(4):606–635, 2004.
5. G. Alexandron, H. Kaplan, and M. Sharir. Kinetic and dynamic data structures for convex hulls and upper envelopes. In *Proc. 9th Intl. Workshop on Data Structures*, pages 269–281, 2005.
6. J. Basch, L. J. Guibas, and J. Hershberger. Data structures for mobile data. *J. Algorithms*, 31:1–28, 1999.
7. J. Basch, L. J. Guibas, and L. Zhang. Proximity problems on moving points. In *Proc. 13th ACM Sympos. Comput. Geom.*, pages 344–351, 1997.
8. The CGAL Library. `http://www.cgal.org/`.
9. G. Collins and A. Akritas. Polynomial real root isolation using Descarte's rule of signs. In *Proc. 3rd ACM Sympos. Symbol. Algebra. Comput.*, pages 272–275, 1976.
10. The Core Library. `http://www.cs.nyu.edu/exact/`.
11. S. Fortune. Progress in computational geometry. In R. Martin, editor, *Directions in Geometric Computing*, pages 81–128. Information Geometers Ltd., 1993.
12. S. Funke, C. Klein, K. Mehlhorn, and S. Schmitt. Controlled perturbation for Delaunay triangulations. In *Proc. 16th ACM-SIAM Sympos. Discrete Algorithms*, pages 1047–1056, 2005.
13. L. J. Guibas. Algorithms for tracking moving objects. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*. CRC Press, 2nd edition, 2004.
14. L. J. Guibas and M. Karavelas. Interval methods for kinetic simulation. In *Proc. 15th Annu. ACM Sympos. Comput. Geom.*, pages 255–264, 1999.
15. L. J. Guibas, M. Karavelas, and D. Russel. A computational framework for handling motion. In *Proc. 6th Workshop on Algorithm Engineering and Experiments*, pages 129–141, 2004.
16. L. J. Guibas and D. Russel. An empirical comparision of techniques for updating Delaunay triangulations. In *Proc. 20th Annu. Sympos. Comput. Geom.*, pages 170–179, 2004.
17. D. Halperin and C. Shelton. A perturbation scheme for spherical arrangements with application to molecular modeling. *Comput. Geom. Theory Appl.*, 10:273–287, 1998.
18. N. Jacobson. *Basic Algebra I*. W. H. Freeman, New York, 2nd edition, 1985.
19. V. Milenkovic and E. Sacks. An approximate arrangement algorithm for semi-algebraic curves. In *Proc. 22nd Annu. Sympos. Comput. Geom.*, pages 237–246, 2006.
20. S. Schirra. Robustness and precision issues in geometric computation. In J. R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 597–632. Elsevier Science, 2000.
21. C. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*. CRC Press, 2nd edition, 2004.

Kinetic Algorithms Via Self-adjusting Computation

Umut A. Acar[1], Guy E. Blelloch[2], Kanat Tangwongsan[2], and Jorge L. Vittes[3]

[1] Toyota Technological Institute
[2] Carnegie Mellon University
[3] Stanford University

Abstract. Define a *static algorithm* as an algorithm that computes some combinatorial property of its input consisting of static, i.e., non-moving, objects. In this paper, we describe a technique for syntactically transforming static algorithms into *kinetic algorithms*, which compute properties of moving objects. The technique offers capabilities for composing kinetic algorithms, for integrating dynamic and kinetic changes, and for ensuring robustness even with fixed-precision floating-point arithmetic. To evaluate the effectiveness of the approach, we implement a library for performing the transformation, transform a number of algorithms, and give an experimental evaluation. The results show that the technique performs well in practice.

1 Introduction

Since first proposed by Basch, Guibas, and Hershberger [13], many *kinetic data structures* for computing properties of moving objects have been designed and analyzed (e.g., [8, 12, 9]). Some kinetic data structures have also been implemented [14, 12, 17]. A kinetic data structure for computing a property can be viewed as maintaining the proof obtained by running a static algorithm for computing that property. Based on this connection between static algorithms and kinetic data structures, previous work developed kinetic data structures by *kinetizing* static algorithms. In all previous approaches, the kinetization process is performed manually.

This paper proposes techniques for kinetizing static algorithms semi-automatically by applying a syntactic transformation. We call such kinetized algorithms as *kinetic algorithms*. The transformation (Section 2) relies on self-adjusting computation [1], where programs can respond to any change to their data (e.g., insertions/deletions into/from the input, changes to the outcomes of comparisons) by running a general-purpose *change-propagation algorithm*. We evaluate the effectiveness of the approach by kinetizing a number of algorithms (Sections 3 and 4), including the merge-sort and the quick-sort algorithms, the Graham-Scan [16], merge-hull, quick-hull [11], ultimate [15] algorithms for computing convex hulls, and Shamos's algorithm for computing diameters [21] and performing an experimental evaluation. Our experiments (Section 5) show that kinetized algorithms are efficient in practice.

For the transformation to yield an efficient kinetic algorithm, the static algorithms being transformed need to be *stable*. Informally, an algorithm is stable if a small change to the input causes a small change in the execution of the algorithm. In previous work [1, 6] we have formalized the notion of stability and described approaches to analyzing it. In practice many algorithms seem stable with respect to small input changes, or can be

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 636–647, 2006.

made stable with minor modifications. From a theoretical point of view, our approach can be viewed as a reduction from dynamic/kinetic problems to stable, static problems. Given that the algorithm designer needs to analyze the stability, one may wonder what advantages the approach has over direct design of kinetic data structures, which often also start by considering static algorithms.

We briefly describe here several advantages of the approach over traditional approaches. In addition to guaranteeing the correctness of kinetized algorithms, the approach enables some capabilities that can dramatically simplify the design and implementation of algorithms for motion simulation. These capabilities are inherent to the approach (require no changes to the implementation) and include composibility, integration of dynamic and kinetic changes, and the ability to advance the simulation time to any time in the future. *Composibility* refers to the ability to send (or pipe) the output of one kinetic algorithm to another: e.g., if `f(·)` and `g(·)` are kinetic algorithms, then `f(g(·))` is a kinetic algorithm. Composibility is important for building large software systems from smaller modules. *Integration of dynamic and kinetic changes* refers to the ability of kinetic algorithms to respond to both dynamic changes (e.g., insertions/deletions into/from the input), and kinetic changes due to motion. With previously proposed approaches, integrating dynamic and kinetic changes can involve major changes to a dynamic or kinetic data structure. For example, Basch et al.'s kinetic convex-hull data structure [13], which does not handle dynamic changes, is very different from Alexandron et al.'s data structure [10], which supports integrated changes. *Advancing-time* capability refers to the ability to advance the simulation time to any time in the future. In addition to combining time-stepping and kinetic simulation approaches, this capability also helps ensure robustness in the presence of certain numerical inaccuracies (discussed in more detail below). Since the approach makes it possible to transform static algorithms into kinetic semi-automatically and guarantees correctness, it also has some software engineering benefits: only the static code needs to be maintained, documented, and debugged.

An important problem in motion simulation is ensuring robustness in the presence of numerical errors in computing the roots of certain polynomials, called *certificate polynomials*. These roots give the *failure times (events)* at which the computed property may change. Based on the advancing-time capability of kinetic algorithm, we describe a scheduling algorithm that guarantees robustness even with finite-precision floating-point arithmetic (Section 2.2). The idea behind our approach is to process the events closer than the smallest computable precision together as a batch. In all previous work, events are processed one by one by computing their order exactly—this requires expensive numerical techniques based on exact and/or interval arithmetic [19, 18, 17]. The reason for processing events one by one is that events may be interdependent: processing one may invalidate another. Our approach is made possible by the ability of the change-propagation algorithm to process interdependent events correctly. It is not known if previously proposed approaches based on kinetic data structures can be extended to support interdependent events (efficiently).

To illustrate the differences between our proposal and traditional approaches based on kinetic data structures, consider the example of computing the convex hull of a set of points above some dividing line (e.g., Figure 1). In the static setting, where points

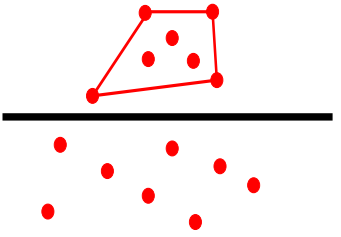


Fig. 1. Computing the convex hull of a set of points that are above some line

do not move, this can be performed by composing a function `filter_s` that finds the points above a line with the function `hull_s` that computes the convex hull, i.e., the algorithm can be expressed as `hull_s(filter_s(P))`, where `P` is a set of points. Our approach enables giving the kinetic algorithm `hull_k(filter_k(P_k))`, where `hull_k` and `filter_k` are the kinetic versions of `filter_s` and `hull_s` obtained by applying our syntactic transformation, and `P_k` is a set of moving points. This algorithm supports the aforementioned capabilities without further modifications to the implementation.

Suppose we want to solve the same problem by composing the kinetic data structures `filter_kds` and `hull_kds` for the filtering and the convex hull problems. Note first that `hull_kds` must respond to integrated dynamic and kinetic changes, because the output of `filter_kds` will change over time as points cross the dividing line. To compose the two data structures, it is also necessary to convert the changes in the output of `filter_kds` into appropriate insert/delete operations for `hull_kds`. This requires 1) computing the "edit" (changes) between successive outputs of `filter_kds`, 2) implementing a data structure for communicating the changes to `hull_kds` (chapter 9 in Basch's thesis [12]). We don't know of any previously proposed general-purpose approaches to computing "edits" between arbitrary data structures efficiently. Finally, kinetic data structures rely on processing events one by one. This requires sequentializing simultaneous events (e.g., when multiple points cross the dividing line at the same time) and using costly numerical techniques to determine the exact order of failing certificates.

2 From Static to Kinetic Programs

We describe the transformation from static to kinetic algorithms, and present an algorithm for robust motion simulation by exploiting certain properties of the transformation (Section 2.2). The asymptotic complexity of kinetic algorithms can be determined by analyzing the *stability* of the program; we describe stability briefly in Section 2.3.

2.1 The Transformation

The transformation of a static program (algorithm) into a kinetic program requires two steps. First, the static program is transformed into a self-adjusting program. Second, the self-adjusting program is kinetized by linking it with a kinetic scheduler.

Transforming a static program into a self-adjusting program requires annotating the program with primitives for creating, reading, and writing modifiable references, and for memoization. A *modifiable (reference)* is a reference, whose contents is a *changeable* or *time dependent* value. In particular, once a self-adjusting program executes, the contents of modifiables can be changed, and the computation can be updated by running a *change-propagation algorithm*. For the purposes of this paper, changeable data consists of all comparisons that involve moving points, and the "next pointers" in the input list. Placing the outcomes of comparisons into modifiables enables changing them as points move; placing the links into modifiables enables inserting/deleting elements into/from the input. After the programmer determines what data is changeable, s/he can transform the program by annotating it with the aforementioned primitives. This transformation is aided by language techniques that ensures correctness [1, 2, 4]. Example transformations can be found elsewhere [7, 1].

Kinetizing a self-adjusting program requires replacing the comparisons in the program with certificate-generating comparisons. This is achieved by linking the program with a library that provides the certificate-generating comparisons. When executed, a certificate-generating comparison creates a *certificate* consisting of a boolean value and a *certificate function* that represents the value of the certificate over time. Creating a certificate requires computing its *failure time* by finding the roots of its certificate function, and inserting the certificate into a *certificate (priority) queue*. An *event scheduler* simulates motion by repeatedly removing the earliest certificate to fail from the certificate queue, changing its outcome, and running the change propagation.

The key difference between our approach and the previously proposed approaches to motion simulation is the use of the change-propagation algorithm for updating computation. Instead of requiring the design of a kinetic data structure, the change-propagation algorithm takes advantage of the computation structure expressed by the static algorithm to update the output. To achieve efficiency, the change-propagation algorithm [3, 1] relies on an integral combination of memoization [5] and dynamic-dependence graphs [6, 4]. Since change-propagation is general purpose and can handle any change to the computation, kinetic (self-adjusting) algorithm have the following capabilities:

- **Integrated Changes:** They can respond to any change to their data including any combination of changes to the input (a.k.a., dynamic changes), and changes to the outcomes of comparisons (a.k.a., kinetic changes).
- **Composibility:** They are composable: if `f(·)` and `g(·)` are kinetic algorithms, then so is `f(g(·))`.
- **Advancing Time:** In a kinetic simulation with a kinetic (self-adjusting) algorithm, the simulation time can be advanced from the current time to any time t in the future. This requires first changing the outcome of certificates that fail between the current time and t, and then running change propagation.

2.2 Robust Motion Simulation

Traditional approaches to motion-simulation based on kinetic data structures rely on computing the exact order in which certificates fail. The reason for this is correctness: since comparisons can be interdependent, changing the outcome of one certificate can invalidate (delete) another certificate. Thus, if the failure order of comparisons is not

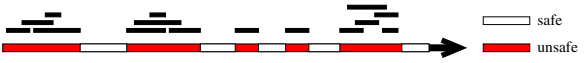


Fig. 2. The simulation time, the certificate failure intervals, and safe and unsafe time intervals

determined exactly, then the event scheduler can prematurely process an event e_1, before the event e_2 that invalidates e_1. This can easily lead to an error by violating critical invariants. Previous work on robust motion simulation focused on techniques for determining the exact order of failure times by using numerical approaches [19, 18, 17].

We propose an algorithm for robust motion simulation that only requires fixed-precision floating-point arithmetic. The algorithm takes advantage of the advancing-time property of kinetic algorithms to perform change-propagation only at "safe" points in time at which the outcomes of certificates can be computed precisely. Given a kinetic simulation, where each certificate is associated with an interval that contains its exact failure time, we say that a time t is *safe* if t is not contained in the interval of any certificate. Figure 2 shows a hypothetical example and some safe time intervals.

If the scheduler could determine the safe time points, then it would perform a robust simulation by repeatedly advancing the time to the earliest next safe time, i.e., *target*. Since the outcomes of all comparisons can be determined correctly at safe targets, such a simulation is guaranteed to be correct. It is not possible, however, to know what targets are safe online, because this requires knowing all the future certificates. Our algorithm therefore selects a safe target t based on existing certificates and aborts when it finds that t becomes unsafe, which happens if, during the change propagation, a certificate whose interval contains t is created. To abort, the algorithm restarts the simulation at the next safe time greater than t (this ensures progress).

As discussed in Section 5, this approach seems very effective in practice. To ensure robustness, the scheduler needs to process less than two certificates per event (on average), and requires very few restarts.

2.3 Stability

The asymptotic complexity of change propagation with a kinetic algorithm can be determined by analyzing the *stability* of the kinetic algorithm. Since this paper concerns experimental issues, we give a brief overview of stability here and refer the reader to the first author's thesis for further details [1]. The stability of an algorithm is measured by computing the "edit distance" between the executions of the algorithm on different data as the symmetric set difference of the executed instructions. For example, the stability of the merge sort algorithm under a change to the outcome of one of the comparisons can be determined by computing the symmetric set difference of the set of comparisons performed before and after this change. Elsewhere [1], we prove that, under certain conditions, change-propagation takes time proportional to the edit distance between the traces of the algorithm on the inputs before and after the change.

3 Implementation

We implemented a library for transforming static algorithms into kinetic. The library consists of primitives for creating certificates, event scheduling, and is based on a

library for self-adjusting-computation. The self-adjusting-computation library is described elsewhere [3, 2]. The implementation of the kinetic event scheduler follows the description in Section 2.2; as a priority queue, a binary heap is used. For solving the roots of the polynomials, the library relies on a degree-two solver, which uses the standard floating-point arithmetic and makes no further accuracy guarantees. The solver can be extended to solve higher-degree polynomials. The full code for the implementation is available at `http://ttic.uchicago.edu/~umut/sting`

4 Applications

Using our library for kinetizing static algorithms, we implemented a number of algorithms and kinetized them. The algorithms include an algorithm for finding the minimum key in a list (`minimum`), the `quick-sort` and the `merge-sort` algorithms, several convex hull algorithms including `graham-scan` [16], `quick-hull` [11], `merge-hull`, the (improved) `ultimate` convex-hull algorithm [15], and an algorithm, called `diameter`, for finding the diameter of a set of points [20]. The input to all our algorithms is a list of one or two dimensional points. Each component of a point is a univariate polynomial of time with floating-point coefficients. In the static versions of the algorithms, the polynomials have degree zero; in the kinetic versions, the polynomials can have an arbitrary degree depending on the particular motion represented. For our experiments, we only consider linear motion plans; the polynomials therefore have no more than degree 2.

To obtain an efficient kinetic algorithm for an application, we first implement a stable, static algorithm for that application and then transform the algorithm into a kinetic algorithm using the techniques described in Section 2.1. The transformation increases the number of lines by about 20% on average. Our implementations rely on the capability to compose kinetic algorithms: the `quick-hull` and `ultimate` algorithms use `minimum` to find the point furthest away from a line; `graham-scan` uses `merge-sort` to sort its input points; `diameter` uses `quick-hull` to compute the convex hull of the points and `minimum` to find the furthest antipodal pair, etc.

Not every algorithm is stable. For example, the straightforward list-traversal algorithm for computing the minimum of a list of keys is not stable. Our algorithm for computing the minimum relies on random-sampling (details can be found elsewhere [7, 1]). The other algorithms require small changes to ensure stability: the `merge-sort` and `merge-hull` algorithms require randomizing the split phase so that the input list is randomly divided into two sets (instead of dividing in the middle); the `ultimate` convex-hull algorithm requires randomizing the elimination step; the `quick-sort`, `quick-hull`, `graham-scan`, and `diameter` algorithms require no changes.

5 Experimental Results

We present an experimental evaluation of the approach. We give detailed experimental results for the `diameter` application, and summarize the results for other applications. We compare our kinetic `minimum` algorithm to a (hand-designed) kinetic data struc-

ture for maintaining the minimum [13, 12].[1] We finish by comparing the convex-hull algorithms and discussing the effectiveness of our robust scheduling algorithm.

Experimental Setup. We ran our experiments on a 2.7GHz Power Mac G5 with 4 gigabytes of memory. We compiled the applications with the MLton compiler using "`-runtime ram-slop 1`" option that directs the run-time system to use all the available memory on the system—MLton, however, can allocate a maximum of about two gigabytes. Since MLton uses garbage collection, the total time depends on the particulars of the garbage-collection system. We therefore report the *application time*, measured as the total time minus the time spent for garbage collection (garbage collection is discussed elsewhere [3]). For the experiments, we use a standard floating-point solver with the robust kinetic scheduler (Section 2.2). We assume that certificate failure times are computed within an error of $\pm 10^{-10}$.

Input Generation. We generate the inputs for our experiments randomly. For one-dimensional applications, we generate points uniformly at random between 0.0 and 1.0 and assign them velocities uniformly at random between -0.5 and 0.5. For two-dimensional applications, we pick points from within the unit square uniformly at random and assigning a constant velocity vector to each point where each component is selected from the interval $[-0.5, 0.5]$ uniformly at random.

Measurements. In addition to measuring various quantities such as the number of events in a kinetic simulation, we run some specific experiments. These experiments are described below; throughout, n denotes the input size (e.g., number of points).

- **Average time for an insertion/deletion:** This is measured by applying a delete-propagate-insert-propagate step to each point in the input. Each step deletes an element, runs change propagation, inserts the element back, and runs change propagation. The average is taken over all propagations.[2]
- **Average time for a kinetic event:** This is measured by running a kinetic simulation and averaging over all events. For all applications except for `graham-scan` and sorting applications, we run the simulations to completion. For sorting and `graham-scan` applications, we run the simulations for the duration of $10 \times n$ events.
- **Average time for an integrated dynamic change & kinetic event:** This is measured by running a kinetic simulation while performing one dynamic change at every kinetic event. Each dynamic change scales the coordinates of a point by 0.8. We run the simulation for the duration of $2 \times n$ events such that all points are scaled twice. The average is taken over all events and changes.

Diameter. The `diameter` application first computes (using `quick-hull`) the convex hull of its input, performs a scan of the convex hull to compute the antipodal pairs, and finds (using `minimum`) the pair that is furthest apart. We note that Agarwal et al. give a

[1] We also tried to compare our implementation to the implementation of kinetic convex-hulls by Basch et al. [14]. Unfortunately, we could not compile their implementation, because it relies on depreciated libraries.

[2] When measuring these operations, the kinetic event queue operations are turned off.

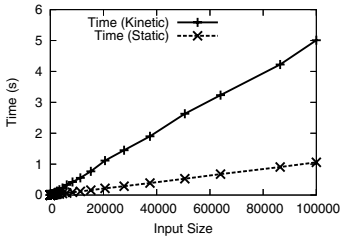


Fig. 3. Time (seconds) for initial run (`diameter`)

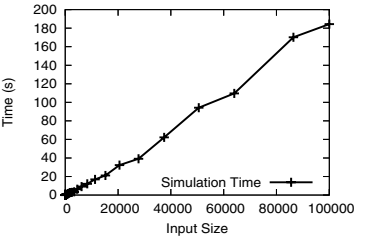


Fig. 4. The time (seconds) for a complete kinetic simulation (`diameter`)

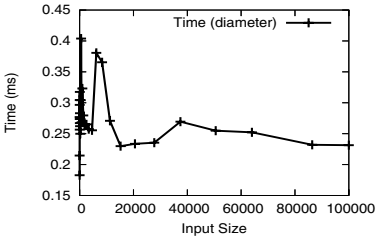


Fig. 5. Average time (milliseconds) for an insertion/deletion (`diameter`)

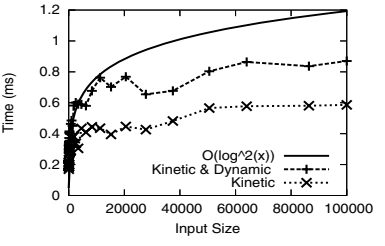


Fig. 6. Average time (milliseconds) per kinetic event and integrated changes (`diameter`)

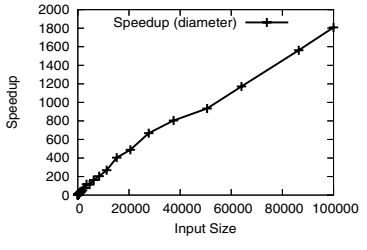


Fig. 7. Average speedup for kinetic events (`diameter`)

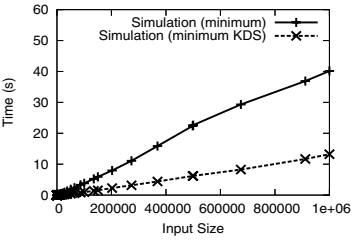


Fig. 8. Simulation time with `minimum` and a tournament-based kinetic data structure

similar algorithm for computing diameters, but they provide no implementation [8]. We expect a similar technique can be used to compute the width of a point set.

Figure 3 shows the total time for a from-scratch run of the kinetic `diameter` algorithm for varying input sizes. The figure shows that the kinetic algorithm is at most 5 times slower than the static algorithm for the considered inputs—due to the event queue, asymptotic overhead of a kinetic algorithm is $O(\log n)$. Figure 4 shows the total time for complete kinetic simulations of varying input sizes—the curve seems slightly super-linear. Figure 5 shows the average time for change propagation after an insertion/deletion for varying inputs. Note that the time for change propagation decreases slightly as the input size increases. We believe that this is because the running time for `diameter` (and thus change propagation) is sensitive to the size of the convex-hull of the points. In particular, 1) deleting/inserting a point from/into the inside of a

Table 1. From-scratch runs and dynamic changes

Application	n	Static Run	Kinetic Run	Overhead	Insert Delete	Speedup
minimum	10^6	0.8	6.2	7.8	1.6×10^{-5}	> 50000
merge-sort	10^5	1.3	9.7	7.4	3.6×10^{-4}	> 4000
quick-sort	10^5	0.3	9.8	31.6	3.7×10^{-4}	> 800
graham-scan	10^5	2.3	12.5	5.4	8.0×10^{-4}	> 3000
merge-hull	10^5	2.2	10.0	4.7	6.0×10^{-3}	> 300
quick-hull	10^5	1.1	5.0	4.7	2.1×10^{-4}	> 5000
ultimate	10^5	1.8	7.8	4.2	1.0×10^{-3}	> 1500
diameter	10^5	1.1	5.0	4.7	2.3×10^{-4}	> 5000

Table 2. Kinetic simulations (also with integrated changes)

Application	n	Static Run	Simulation	# Events	# Ext. Events	Per Event	Per Int. Event	Speedup
minimum	10^6	0.8	40.2	5.3×10^5	9	9.3×10^{-5}	9.3×10^{-5}	> 8000
merge-sort	10^5	1.3	239.1	10^6	10^6	2.4×10^{-4}	9.8×10^{-4}	> 6000
quick-sort	10^5	0.3	430.9	10^6	10^6	4.3×10^{-4}	2.9×10^{-2}	> 700
graham-scan	10^5	2.3	710.3	10^6	38	7.1×10^{-4}	1.4×10^{-3}	> 3000
merge-hull	10^5	2.2	1703.6	6.8×10^5	293	2.5×10^{-3}	7.4×10^{-3}	> 800
quick-hull	10^5	1.1	171.9	3.1×10^5	293	5.6×10^{-4}	8.9×10^{-4}	> 2000
ultimate	10^5	1.8	1757.8	4.1×10^5	293	4.3×10^{-3}	7.3×10^{-3}	> 400
diameter	10^5	1.1	184.4	3.1×10^5	11	5.9×10^{-4}	8.7×10^{-4}	> 2000

convex hull of the input points is cheap and 2) with uniformly randomly distributed points, many of the points are expected to be inside the hull. Figure 6 shows the average time per kinetic event and the average time for an integrated dynamic change and kinetic event. Both curves fit $O(\log^2 n)$. These experimental results match best known asymptotic bounds for the kinetic diameter problem [8]. To measure of how fast change propagation is, we compute the average speedup (Figure 7) as the ratio of the average time for one kinetic event to the time for a from-scratch execution of the static version. As can be seen, the speedup increases nearly linearly with the input size to exceed three orders of magnitude.

Other benchmarks. We report a summary of our results for other benchmarks at fixed input sizes. Table 1 shows, for input sizes ("n"), the timings for from-scratch executions of the static version ("Static Run") and the kinetic version ("Kinetic Run"), the overhead, the average time for change propagation after an insertion/deletion ("Insert/Delete"), and the speedup of change propagation computed as the average time for an insertion/deletion divided by the time for recomputing from scratch using the static algorithm. The overhead, defined as the ratio of the time for a kinetic run to the time for a static run, is $O(\log n)$ asymptotically because of the certificate-queue operations. The experiments show that the overhead is about 9 on average for the considered inputs, but varies significantly depending on the applications. As can be expected, the more

sophisticated the algorithm, the smaller the overhead, because the time taken by the library operations (operations on certificates, event queue, modifiables, etc.) compared to the amount of "real" work performed by the static algorithm is small for more sophisticated algorithms. The "speedup" column shows that change propagation can be orders of magnitude faster than recomputing from scratch.

Table 2 shows the timings for kinetic simulations. The "n" column shows the input size, "Simulation" column shows the time for a kinetic simulation, the "# Events" and "# Ext. Events" columns show the number of events and external events respectively, the "per Event" column shows the average time per kinetic event. The "per int. ev." column shows the average time for an integrated dynamic and kinetic event. The "Speedup" column shows the average speedup computed as the ratio of time for a from-scratch execution of the static version to the average time for an event. The speedup column shows that the change propagation is orders of magnitude faster than re-computing from scratch.

The results show that `merge-sort` is more effective than `quick-sort`: the `merge-sort` algorithm is two times faster for kinetic events and nearly thirty times faster for integrated events. We discuss the convex-hull algorithms below.

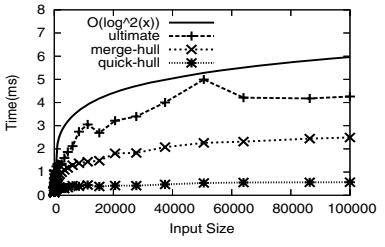


Fig. 9. Average time (milliseconds) per kinetic event for some convex-hull algorithms

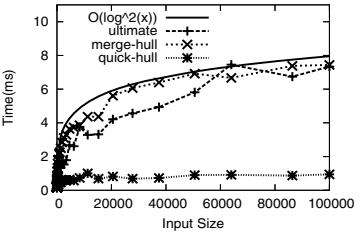


Fig. 10. Average time (milliseconds) per integrated kinetic and dynamic events for some convex-hull algorithms

A comparison of convex hull algorithms. We compare the `quick-hull`, `ultimate`, and `merge-hull` algorithms based on their *responsiveness, efficiency* and *locality*. which measure the effectiveness of kinetic algorithms [13]. Since `graham-scan` algorithm relies on sorting, it is not practical; we therefore do not discuss `graham-scan` in detail.

Figure 9 shows the time per event for the convex hull algorithms. In a kinetic simulation, the time per event measures the *responsiveness* of a kinetic algorithm, and the total number of events processed determines the *efficiency* of an algorithm. The total simulation time measures overall effectiveness of a kinetic algorithm. In all these respects, the algorithms rank from best to worst as `quick-hull`, `merge-hull`, and `ultimate`.

Kinetic algorithms can also be compared based on their *locality* [13], which is defined as the maximum number of certificates that depend on any input point. The time for integrated dynamic and kinetic changes (Figure 10) gives a measure of locality because a change to the coordinates of a point requires recomputing all certificates that depend on that point. In terms of their locality, the algorithms rank from best to worst as `quick-hull`, `ultimate`, and `merge-hull`.

The results show that the `quick-hull` performs best. One disadvantage of `quick-hull` is that it is difficult to prove asymptotic bounds for it. If asymptotic complexity is important, then the experiments indicate that `merge-hull` algorithm performs better than `ultimate`, especially if few dynamic changes are performed.

Comparison to a handcrafted kinetic data structure. One may wonder how the approach performs relative to handcrafted kinetic data structures. We compare our `minimum` algorithm to the tournament-tree based kinetic data structure for maintaining minimum by Basch, Guibas, and Hershberger [13, 12]. Figure 8 shows the total time of a kinetic simulation with our semi-automatically generated algorithm and the Basch-Guibas-Hershberger kinetic data structure. Our algorithm is a factor of 3 slower than the handcrafted data structure.

Robustness. Our experiments rely on the robust scheduling algorithm (Section 2.2). To determine the effectiveness of the approach, we performed additional testing by running kinetic simulations and probabilistically verifying the output after each kinetic event. These tests verified that the approach ensures correctness for all inputs that we considered: up to 100,000 points with all applications.[3] With computational geometry algorithms, the scheduler performed no cold restarts. With sorting (and `graham-scan`) algorithms, there were ten restarts with 100,000 points—no restarts took place for smaller inputs. Since sorting algorithms can process up to $O(n^2)$, this is not surprising.

The robust scheduling algorithm can process multiple certificates simultaneously. We measured the number of certificates processed simultaneously to be less than 1.75 averaged over all our applications. Note that both the number of restarts and the number of certificates can be further decreased by using higher (but still fixed) precision floating-point numbers.

6 Conclusion

This paper describes the first technique for kinetizing static algorithms by applying a syntactic transformation based on self-adjusting computation. The technique ensures that kinetized algorithms are correct, and enables 1) integrating dynamic and kinetic changes, 2) composing kinetic algorithms, and 3) robust motion simulations. The effectiveness of the technique is evaluated by considering a number of algorithms and performing a broad range of experiments. The experimental results show that the approach performs well in practice.

References

1. Umut A. Acar. *Self-Adjusting Computation*. PhD thesis, Department of Computer Science, Carnegie Mellon University, May 2005.
2. Umut A. Acar, Guy E. Blelloch, Matthias Blume, Robert Harper, and Kanat Tangwongsan. A library for self-adjusting computation. In *ACM SIGPLAN Workshop on ML*, 2005.

[3] These limits are due to memory limitations of the MLton compiler. We could run some applications with more than 300,000 points.

3. Umut A. Acar, Guy E. Blelloch, Matthias Blume, and Kanat Tangwongsan. An experimental analysis of self-adjusting computation, 2006. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.
4. Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages*, pages 247–259, 2002.
5. Umut A. Acar, Guy E. Blelloch, and Robert Harper. Selective memoization. In *Proceedings of the 30th Annual ACM Symposium on Principles of Programming Languages*, 2003.
6. Umut A. Acar, Guy E. Blelloch, Robert Harper, Jorge L. Vittes, and Maverick Woo. Dynamizing static algorithms with applications to dynamic trees and history independence. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2004.
7. Umut A. Acar, Guy E. Blelloch, Kanat Tangwongsan, and Jorge L. Vittes. Kinetic algorithms via self-adjusting computation. Technical Report CMU-CS-06-115, Department of Computer Science, Carnegie Mellon University, March 2006.
8. Pankaj K. Agarwal, David Eppstein, Leonidas J. Guibas, and Monika Rauch Henzinger. Parametric and kinetic minimum spanning trees. In *Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science*, pages 596–605, 1998.
9. Pankaj K. Agarwal, Leonidas J. Guibas, John Hershberger, and Eric Veach. Maintaining the extent of a moving set of points. *Discrete and Computational Geometry*, 26(3):353–374, 2001.
10. Giora Alexandron, Haim Kaplan, and Micha Sharir. Kinetic and dynamic data structures for convex hulls and upper envelopes. In *9th Workshop on Algorithms and Data Structures (WADS). Lecture Notes in Computer Science*, volume 3608, pages 269—281, aug 2005.
11. C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, 1996.
12. Julien Basch. *Kinetic Data Structures*. PhD thesis, Department of Computer Science, Stanford University, June 1999.
13. Julien Basch, Leonidas J. Guibas, and John Hershberger. Data structures for mobile data. *Journal of Algorithms*, 31(1):1–28, 1999.
14. Julien Basch, Leonidas J. Guibas, Craig D. Silverstein, and Li Zhang. A practical evaluation of kinetic data structures. In *SCG '97: Proceedings of the thirteenth annual symposium on Computational geometry*, pages 388–390, New York, NY, USA, 1997. ACM Press.
15. Timothy M. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete and Computational Geometry*, 16:361–368, 1996.
16. R. L. Graham. An efficient algorithm for determining the convex hull of a finete planar set. *Information Processing Letters*, 1:132–133, 1972.
17. Leonidas Guibas, Menelaos Karaveles, and Daniel Russel. A computational framework for handling motion. In *Proceedings of teh Sixth Workshop on Algorithm Engineering and Experiments*, pages 129–141, 2004.
18. Leonidas Guibas and Daniel Russel. An empirical comparison of techniques for updating delaunay triangulations. In *SCG '04: Proceedings of the twentieth annual symposium on Computational geometry*, pages 170–179, New York, NY, USA, 2004. ACM Press.
19. Leonidas J. Guibas and Menelaos I. Karavelas. Interval methods for kinetic simulations. In *SCG '99: Proceedings of the fifteenth annual symposium on Computational geometry*, pages 255–264. ACM Press, 1999.
20. F. P. Preparata and M. I. Shamos. *Computational Geometry*. Springer-Verlag Inc., 1985.
21. Michael I. Shamos. *Computational Geometry*. PhD thesis, Department of Computer Science, Yale University, 1978.

Parallel Machine Scheduling Through Column Generation: Minimax Objective Functions (Extended Abstract)

J.M. van den Akker, J.A. Hoogeveen, and J.W. van Kempen

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80089, 3508 TB Utrecht, The Netherlands
marjan@cs.uu.nl, slam@cs.uu.nl, jwkempen@cs.uu.nl

Abstract. In this paper we consider one of the basic problems in scheduling and project management: scheduling on parallel identical machines. We present a *solution framework for a number of scheduling problems* in which the goal is to find a feasible schedule that minimizes some objective function of the minimax type on a set of parallel, identical machines, subject to release dates, deadlines, and/or generalized precedence constraints. Our framework is based on column generation. Although column generation has been successfully applied to many parallel machine scheduling problems with objective functions of the minsum type, the number of applications for minimax objective functions is still small.

We determine a lower bound on the objective function in the following way. We first turn the minimization problem into a decision problem by bounding the outcome value. We then ask ourselves 'Are m machines enough to feasibly accommodate all jobs?'. We formulate this as an integer linear programming problem and we determine a high quality lower bound by applying column generation to the LP-relaxation; if this lower bound is more than m, then we can conclude infeasibility. To speed up the process, we compute an intermediate lower bound based on the outcome of the pricing problem. As the pricing problem is intractable for many variants of the original scheduling problem, we mostly solve it approximately by applying local search, but once in every 50 iterations or when local search fails, we use a time-indexed integer linear programming formulation to solve the pricing problem.

After having derived the lower bound on the objective function of the original scheduling problem, we try to find a matching upper bound by identifying a feasible schedule with objective function value equal to this lower bound. Our computational results show that our lower bound is so strong that this almost always succeeds. We are able to solve problems with up to 160 jobs and 10 machines in 10 minutes on average.

1980 Mathematics Subject Classification (Revision 1991): 90B35.
Keywords and Phrases: parallel machine scheduling, set covering formulation, linear programming, column generation, maximum lateness, release dates, precedence constraints, intermediate lower bounds, time-indexed formulation.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 648–659, 2006.

1 Introduction

In this paper we consider one of the basic problems in scheduling and project management; we refer to the book by Pinedo (2002) for an introduction to scheduling theory. We are given m parallel, identical machines, which are continuously available from time zero onwards and can process no more than one job at a time; these machines have to process n jobs, which are denoted by $J_1, \ldots, J_n$. Processing J_j requires one, arbitrary processor during an uninterrupted period of length p_j, which must start at or after the given release date r_j and must be completed by the given deadline $\bar{d}_j$. Given a schedule σ, we denote the completion time of job J_j by $C_j(\sigma)$, and hence, we need for all jobs J_j that $r_j + p_j \leq C_j(\sigma) \leq \bar{d}_j$ for σ to be feasible. Moreover, the jobs may be subject to generalized precedence constraints, which prescribe that for a pair of jobs J_i and J_j the difference in completion time $C_j(\sigma) - C_i(\sigma)$ should be at least (at most, or exactly) equal to some given value q_{ij}. The quality of the schedule is measured by some objective function of minimax type, which is assumed to be nondecreasing in the completion times, like maximum lateness or maximum cost. Here, the maximum lateness is defined as $\max_j L_j(\sigma)$, where $L_j(\sigma) = C_j(\sigma) - d_j$; d_j signals the due date, by which the job preferably should be completed. A special case occurs when all due dates are equal to zero; in this case, the objective function becomes equal to minimizing the maximum completion time, that is, the makespan of the schedule.

We solve these problems by applying the technique of column generation. This approach has been shown to work very well for the problem of minimizing total weighted completion time on a set of identical parallel machines (see [2] and [7]), and since the appearance of these papers, the method of applying column generation has been applied to many parallel machine problems with a sum type criterion in which the jobs are known to follow a specific order on the individual machines; we refer to [3] for an overview. One notable exception is due to Brucker and Knust ([4], [5], [6]), who apply column generation to a number of resource constrained project scheduling problems in which the goal is to minimize the makespan. Here they first formulate the problem as a decision problem and then use linear programming to check whether it is possible to execute all jobs in a feasible preemptive schedule; here the decision variables refer to the length of a time slice during which a given set of jobs is executed simultaneously.

We use the three-field notation scheme introduced by Graham et al. [9] to denote scheduling problems. The remainder of this paper is organized as follows. In Sect. 2, we describe the basic approach for the relatively simple problem of minimizing $L_{\max}$ without release dates and generalized precedence constraints. We explain the column generation approach and the derivation of an intermediate lower bound in Sect. 3 and 4. In Sect. 5, we add release dates and generalized precedence constraints. In Sect. 6, we describe our local search algorithm to solve the pricing problem approximately, and in Sect. 7 we formulate the pricing problem as a time-indexed integer linear programming problem that can be used to find (an upper bound on) the solution of the pricing algorithm. Finally, in Sect. 8 we draw some conclusions.

Our contribution. We give the first algorithm for solving this kind of problems using column generation. Our approach is a bit complementary to the approach by Brucker and Knust, since they check the existence of a preemptive schedule for a given set of resources, whereas we let the number of machines vary. Moreover, we describe an efficient way to use an intermediate lower bound to be able to make a decision without having to solve the LP-relaxation to optimality.

2 The Basic Approach

In this section, we sketch the basic approach, which we illustrate on the $P||L_{\max}$ problem, that is, there are m parallel, identical machines to execute n jobs, where the objective is to minimize maximum lateness; there are no release dates and precedence constraints, but there can be deadlines, which are not related to the due dates.

It is well-known that this optimization problem can be solved by solving a set of decision problems, which are obtained by putting an upper bound L on the value of the objective function. Since the restriction $L_{\max} \leq L$ is equivalent to the constraint that $L_j = C_j - d_j \leq L$ for each job J_j, we find that $C_j \leq d_j + L \equiv \bar{d}_j$; the decision problem is then to determine whether there exists a feasible schedule meeting all deadlines, where we take the minimum of the original deadline and the deadline induced by the constraint $L_{\max} \leq L$. Hence, we can solve the optimization problem by determining the smallest value L that allows a feasible schedule.

Since the machines are identical, the decision problem can be reformulated as: *is it possible to partition the jobs in at most m subsets such that for each subset we can find a feasible single-machine schedule that meets all deadlines*? Checking the feasibility of a subset is easy by executing the jobs in order of earliest deadline order and see whether these are all met [10]; hoping not to confuse the reader, we call this the ED-order. Note that it is identical to the earliest due date (EDD) order if the original deadlines are not restrictive. We solve this decision problem by answering the question: what is the minimum number of machines that we need to get a feasible schedule?

We call a subset of jobs that allows a feasible single-machine schedule a *machine schedule*. The above question is then to find the minimum number of mutually distinct machine schedules that contain all jobs. We can formulate the above problem as an integer linear programming problem as follows. Let S be the set containing all machine schedules. We introduce binary variables x_s $(s = 1, \ldots, |S|)$ that take value 1 if machine schedule s is selected and 0 otherwise. Each machine schedule s is encoded by a vector $a_s = (a_{1s}, \ldots, a_{ns})$, where $a_{js} = 1$ if machine schedule s contains job J_j and $a_{js} = 0$, otherwise. We have to minimize the number of machine schedules that we select, such that each job is contained in one machine schedule. Hence, we have to determine values x_s that solve the problem

$$(\mathbf{P_1}) \min \quad \sum_{s \in S} x_s$$

subject to

$$\sum_{s \in S} a_{js} x_s = 1, \text{ for each } j = 1, \ldots, n, \tag{1}$$

$$x_s \in \{0, 1\}, \text{ for each } s \in S. \tag{2}$$

We obtain the linear programming relaxation by replacing (2) with $x_s \geq 0$ for all $s \in S$; we do not need to enforce the upper bound of 1 for x_s, since this follows immediately from (1). We solve the LP-relaxation using column generation.

3 Column Generation

We first solve the linear programming relaxation for a small initial subset of the columns. Given the solution to the linear programming problem with the current set of variables, it is well-known from the theory of linear programming that the reduced cost of a variable x_s is given by

$$c'_s = c_s - \sum_{j=1}^{n} \lambda_j a_{js} = 1 - \sum_{j=1}^{n} \lambda_j a_{js},$$

where $\lambda_1, \ldots, \lambda_n$ are the dual multipliers corresponding to the constraints (1) of the solution of the current LP. If for each variable x_s we have that $c'_s \geq 0$, then the solution with the current set of variables solves the linear programming problem with the complete set of variables as well. To check whether this condition is fulfilled, we minimize the reduced cost over all machine schedules. Therefore, we must pick the subset of the jobs with maximum total dual multiplier value among all subsets of jobs that lead to a feasible single-machine schedule, that is, we must solve the problem

$$\max_{s \in S} \sum_{j=1}^{n} \lambda_j a_{js} \tag{3}$$

We use $\hat{c}$ to denote the outcome value of this problem; hence, we have that the minimum reduced cost, which we denote by c^*, is equal to

$$c^* = 1 - \hat{c}$$

This maximization problem is equivalent to the problem of minimizing the total weight of the jobs that are not selected, which is known as the problem of minimizing the weighted number of tardy jobs, where the weight of a job is equal to the dual multiplier λ_j and the due date for each job is equal to the deadline $\bar{d}_j$. Note here that the constraint that each weight is nonnegative in this scheduling problem is not restrictive, since a job with negative weight will never be selected in the maximization problem. This problem, which is denoted as $1||\sum w_j U_j$ in the three-field notation scheme, is solvable in $O(n \sum p_j)$ time by the dynamic programming algorithm of Lawler and Moore [11]. Hence, in this situation we solve the pricing problem to optimality. If $c^* \geq 0$, then we have solved the linear programming relaxation; otherwise, we add the variable with

minimal reduced cost value to the LP and solve it again. In this way, we solve the linear programming relaxation to optimality. If the outcome value is more than m, then we know that the answer to the decision problem is 'no'; if the outcome value is no more than m, and we have not identified a feasible solution yet that uses m (or fewer) machines, then we solve the integer linear programming problem to optimality using the branch-and-bound algorithm developed by Van den Akker et al. [2] for the problem $P||\sum w_jC_j$.

4 An Intermediate Lower Bound

In the above implementation we have to apply column generation to the bitter end, that is, until we have concluded that the linear programming relaxation has been solved to optimality, before we have found a valid lower bound. Since we only need to know whether the outcome value is more than m or no more than m, we are not interested in the exact outcome value, as long as it allows us to decide the decision problem. Fortunately, it is possible to compute an *intermediate lower bound* on basis of the reduced cost.

Theorem 1. *An intermediate lower bound for the optimal value of LP-relaxation of P_1 is given by $\sum_{j=1}^{n} \lambda_j/\hat{c}$, where $\hat{c}$ is the solution value of the maximization problem (3).*

We can use this lower bound to decide whether m machines are sufficient. If it has value smaller than or equal to m, then we continue with solving the LP-relaxation.

Solving the problem $P||L_{\max}$ was relatively simple, since each machine schedule can be represented by just listing the indices of the jobs that it contains, and since the pricing problem can be solved by applying dynamic programming. We can use the same methodology to solve any problem for which putting an upper bound on the objective function results in a set of deadlines. Hence, we can solve the more general $P||f_{\max}$ problem in the same fashion, where $f_{\max}$ denotes maximum cost, which is defined as $\max_j f_j(C_j)$, where $f_j(t)$ is the cost function of job J_j, which is assumed to be nondecreasing in t.

5 Release Dates and Precedence Constraints

In this section, we assume that next to the deadlines there are release dates and generalized precedence constraints. We again translate the problem into one of minimizing the number of machine schedules that are needed. Since two jobs that are connected through a precedence constraint do not have to be executed by the same machine, we assume that the machine schedules obey the release dates and deadlines, and we include a constraint in the integer linear programming formulation for each of the generalized precedence constraints. We define A^1 as the arc set containing all pairs (i, j) such there exists a precedence constraint of the form $C_j - C_i \geq q_{ij}$; similarly, we define A^2 and A^3 as the arc sets that

contain an arc for each pair (i,j), for which $C_j - C_i \leq q_{ij}$ and $C_j - C_i = q_{ij}$, respectively. Note that the intersection of A^1 and A^2 does not have to be empty. To be mathematically correct, we should replace q_{ij} by q_{ij}^1 and q_{ij}^2, if the arc (i,j) occurs in both A^1 and A^2, but to ease notation, we do not make this distinction. We denote the union of A^1, A^2, and A^3 by the multi set A. This leads to the following integer linear programming formulation

$$(\mathbf{P_2}) \min \quad \sum_{s \in S} x_s$$

subject to

$$\begin{aligned}
\sum_{s \in S} a_{js} x_s &= 1, \text{ for each } j = 1, \ldots, n, \\
\sum_{s \in S} C_{js} x_s - \sum_{s \in S} C_{is} x_s &\geq q_{ij} \text{ for each } (i,j) \in A^1; \\
\sum_{s \in S} C_{js} x_s - \sum_{s \in S} C_{is} x_s &\leq q_{ij} \text{ for each } (i,j) \in A^2; \\
\sum_{s \in S} C_{js} x_s - \sum_{s \in S} C_{is} x_s &= q_{ij} \text{ for each } (i,j) \in A^3; \\
x_s &\in \{0,1\}, \text{ for each } s \in S.
\end{aligned}$$

Here C_{js} denotes the completion time of job J_j in column s, which we define to be equal to 0 if J_j is not contained in s. If we want to solve the LP-relaxation by applying column generation, then we find that the reduced cost of a machine schedule s is equal to

$$c'_s = c_s - \sum_{j=1}^{n} a_{js} \lambda_j - \sum_{j=1}^{n} \left[\sum_{h \in P_j} \delta_{hj} C_{js} - \sum_{k \in S_j} \delta_{jk} C_{js} \right].$$

Here the sets P_j and S_j are defined as the sets containing all predecessors and successors of job J_j in A, respectively. Hence, we must solve the maximization problem

$$\sum_{j=1}^{n} a_{js} \lambda_j + \sum_{j=1}^{n} \left[\sum_{h \in P_j} \delta_{hj} C_{js} - \sum_{k \in S_j} \delta_{jk} C_{js} \right]. \tag{4}$$

over all machine schedules $s \in S$. We solve this problem approximately using local search (see Sect. 6). Again, we can compute an intermediate lower bound.

Theorem 2. *An intermediate lower bound for the optimal value of LP-relaxation of P_2 is given by $\left\lceil \sum_{j=1}^{n} \lambda_j + \sum_{(j,k) \in A} \delta_{jk} q_{jk} \right\rceil / \hat{c}$, where $\hat{c}$ is the outcome value of the maximization problem (4).*

If we get stuck, that is, the outcome of the LP-relaxation does not lead to 'no' on the decision problem, then we assume for the time-being that the decision problem is feasible, and we decrease the upper bound L on $L_{\max}$ that we want to test. If we end up with a value L for which we know that $L-1$ yields an infeasible

decision problem and for which the LP-relaxation cannot decide whether the decision problem obtained by putting the upper bound on $L_{\max}$ equal to L is feasible, then we can apply branch-and-bound with a branching strategy based on splitting the execution intervals. It turned out in our experiments, however, that it is better to solve an integer linear programming formulation in which we request that $L_{\max} = L$ by using CPLEX (see Sect. 8).

6 Generating New Columns by Local Search

In this section, we describe the local search algorithm that we have implemented to solve the pricing problem.
Recall from (4) that solving the original pricing problem is equivalent to finding the single-machine schedule that obeys the release dates and deadlines and maximizes

$$\sum_{j=1}^{n} \lambda_j a_{js} + \sum_{j=1}^{n} Q_j C_{js}, \tag{5}$$

where $Q_j = \sum_{h \in P_j} \delta_{hj} - \sum_{k \in S_j} \delta_{jk}$. Looking at this formula we see that, if job J_j gets selected, then this Q_j value determines whether it is more profitable to execute the job as late as possible ($Q_j > 0$) or as early as possible ($Q_j < 0$). In a preprocessing step, we can even determine the time interval during which we must complete job J_j, if selected, since its total contribution to the objective function would be negative otherwise, in which case it would have been better not to select J_j.

In our local search we use a two-phase procedure. In the first phase, we determine the jobs that are selected and the order in which they are executed. In the second phase, we then determine the optimal set of completion times, which can be done in linear time using a shifting procedure, which resembles the procedure for a similar problem given by Garey et al. [8].

We now describe the first step in the local search procedure. We define a solution in our local search as a selection of the jobs and the order in which they should be processed, after which we find the value of this solution by solving the second step. Our algorithm uses the following methods to exploit the solution space:

(i) Remove a random job from the set of selected jobs;
(ii) Add a random, yet unselected job at a random place in the order of selected jobs;
(iii) Replace a random job from the current selection by a random, yet unselected job;
(iv) Swap the positions of two random jobs in the set of selected jobs.

We choose to apply simulated annealing. In our computational experiments, we added up to 50 columns with negative reduced cost per iteration.

7 Time Indexed Formulation

Finally we describe how to find an upper bound for $\hat{c}$, which is defined as the outcome of the maximization problem (5), such that we can compute the intermediate lower bound. To this end, we formulate the problem as an integer linear programming problem using a time-indexed formulation (see for instance [12] and [1]) and solve the LP-relaxation, which gives an upper bound. We checked the intermediate lower bound by computing this upper bound on $\hat{c}$ every 50 iterations, or when our local search algorithm could not find any column with negative reduced cost. If infeasibility could not be decided and if we could not find a column with negative reduced cost, we turn to the time-indexed ILP formulation of the pricing problem. We compute for which value of the objective function of the pricing problem we find an intermediate lower bound equal to m. We then ask our ILP solver CPLEX whether there exists a solution to the pricing problem with this value or larger. If the answer to this decision problem is 'no', then we can conclude that m machines are not enough; if the answer is 'yes', then we add the corresponding column and continue with solving the LP-relaxation by column generation.

8 Computational Results

Compared Methods

Since we could not find other results of reports trying to solve the problem $P|r_j, prec|L_{max}$, we have compared our method to the rather straightforward and direct approach using a time-indexed ILP formulation of this problem like the one stated in Section 7.

When we compared the column generation approach that we had originally in mind, that is, with the branch-and-bound based on splitting the execution intervals, to the method of solving this time-indexed ILP formulation through CPLEX, we noticed that these methods have difficulty with exactly opposite problems. We noticed that for all our testing instances the lower bound on L_{max} found by the LP-relaxation coincided with the optimal value of L_{max}. The problem with our method is that it is often not able to generate a set of columns in the LP-relaxation that form a solution to the original ILP formulation. CPLEX has exactly the opposite problem when solving the time-indexed formulation; it has a very hard time to conclude that a solution is optimal. Therefore, we tried to exploit the best of both worlds in defining a *hybrid method*, using the very strong lower bound LB on L_{max} found by our column generation method and then let CPLEX find a solution with this value (thus adding the constraint $L = LB$ and ignoring a big part of the variables). Hence, the hybrid method first spends some time to find the lower bound and passes this information to the time-indexed formulation.

Results

In our experiments we compare our hybrid algorithm to the direct ILP solved by CPLEX (that is, without knowing the value of the lower bound). We have applied both algorithms on 13 scenarios; for each scenario we ran five test instances. The scenarios are described in Table 1; n denotes the number of jobs, m the number of machines, and # prec denotes the number of precedence constraints. The first 8 scenarios are used to compare our hybrid algorithm to solving the time-indexed formulation directly, without knowing the lower bound. Hence, we are testing whether spending time on determining the lower bound is worthwhile. The scenarios 8-10 are used to find out the influence of the number of machines, whereas in the last two the influence of a doubling of the times value is measured. The results of the experiments are summarized in Table 2. The results of the hybrid algorithm are denoted in the row starting with Hi, where i denotes the number of the scenario; the results of applying CPLEX (Version 9.0) to the ILP formulation appear in the same row between brackets, starting with Ci. The algorithms were encoded in Java (Version 1.4.2_05) and the experiments were run on a Dell Optiplex GX270 P4 2,8 Ghz computer. For each instance we let each algorithm run for at most 30 minutes. We keep track of the number of times out of 5 that an optimum was found ('# success') and also the average and maximum amount of time in seconds needed for the successful runs ('Avg t' and 'Max t'). Next, we measured the average and maximum time needed to find the lower bound for the successful runs ('Avg t LB' and 'Max t LB'). Finally, by ('Avg #ILP' and 'Max #ILP'), we denote the number of times that we solved the ILP formulation of the pricing problem; this was conducted after each series of 50 runs of the local search algorithm, since we wanted to

Table 1. Test scenarios

Number	p_j	r_j	d_j	n	m	# prec
0	U[1,20]	U[0,60]	U[50,80]	40	4	20
1	U[1,20]	U[0,40]	U[30,60]	70	5	35
2	U[1,20]	U[0,80]	U[80,150]	80	7	30
3	U[1,20]	U[0,40]	U[60,80]	100	9	40
4	U[1,20]	U[0,60]	U[80,110]	120	9	50
5	U[1,20]	U[0,60]	U[80,110]	140	10	50
6	U[1,20]	U[0,60]	U[80,110]	160	10	50
7	U[1,20]	U[0,60]	U[80,110]	180	10	60
8	U[1,20]	U[0,60]	U[40,80]	60	3	30
9	U[1,20]	U[0,60]	U[40,80]	60	5	30
10	U[1,20]	U[0,60]	U[40,80]	60	7	30
11	U[1,20]	U[0,60]	U[50,80]	30	3	15
12	U[1,40]	U[0,120]	U[100,160]	30	3	15

Table 2. Results of comparing the direct time-indexed approach and the hybrid algorithm

	# success	Avg t	Max t	Avg t LB	Max t LB	Avg #ILP	Max #ILP
H0(C0)	5(2)	66(27)	194(53)	38	92	33	77
H1(C1)	4(3)	53(30)	170(37)	14	26	4	16
H2(C2)	5(3)	153(231)	396(645)	83	180	47	139
H3(C3)	5(0)	342(-)	1109(-)	110	174	14	33
H4(C4)	5(0)	342(-)	393(-)	183	302	38	57
H5(C5)	5(0)	452(-)	689(-)	228	269	25	41
H6(C6)	5(0)	636(-)	1045(-)	354	415	29	37
H7(C7)	1(0)	553(-)	553(-)	470	470	27	27
H8(C8)	3(0)	199(-)	296(-)	158	266	40	98
H9(C9)	5(2)	88(80)	197(351)	53	85	17	42
H10(C10)	5(5)	6(2)	9(3)	5	8	0	0
H11(C11)	5(5)	31(92)	55(180)	24	35	15	50
H12(C12)	4(0)	101(-)	150(-)	73	147	24	54

find out whether the intermediate lower bound could decide the problem already, and whenever the local search algorithm could not find an improving column.

We also tested the performance of our local search algorithm on the pricing problem by comparing it to the method of only generating columns found by the ILP formulation of the pricing problem. For the scenarios 0, 5, 11 and 12 we ran another set of 5 instances each. We determined the lower bound on these instances by using our local search algorithm and by using the optimal solutions to the ILP only. The results are depicted in Table 3. Here Hi denotes the hybrid algorithm run on scenario i, and Ii denotes the results obtained on scenario i by the algorithm in which the pricing algorithm is solved by the ILP. Scenario 0 is used to show the difference for *easy* instances, where scenario 5 is used to investigate *difficult* instances. Finally scenarios 11 and 12 are used to investigate the influence of the doubling of time values on the results.

Table 3. Results of comparing LS to only ILP solving

	# success	Average time	Maximum time
H0	5	36	89
I0	5	108	230
H5	5	190	298
I5	0	-	-
H11	5	28	48
I11	5	88	127
H12	5	51	91
I12	4	307	506

Evaluation of Our Experiments

Our results clearly show our hybrid algorithm outperforms the method of letting CPLEX solve the full ILP by far. CPLEX is not able to solve the ignorant time-indexed ILP in less than 30 minutes for most of the tested instances, where our hybrid algorithm easily solves nearly all instances. Looking at scenario 3 we already see that CPLEX fails to solve any of the 5 instances with 100 jobs and 9 machines within 30 minutes, where our hybrid algorithm solves all instances we tested up to 160 jobs and 10 machines (scenarios 3-6).

Our results also show that for all instances we managed to solve, the derived lower bound was equal to the optimal value. There are some instances for which we could not check whether optimum and lower bound coincided, for we could not solve them within 30 minutes. It seems reasonable that in at least some of these cases this is due to the fact that the lower bound was not strict. However, we never were able to show that the lower bound differed from the optimum for any instance. Altogether we may draw the conclusion that our lower bound is extremely strong.

If we compare the algorithm of solving the time-indexed formulation without specifying the lower bound with the second part of the hybrid algorithm, then we see that specifying the optimum makes a lot of difference. If we for instance stopped an instance of C5, then the best found upper bound so far in general was way off the optimum. This may be explained partly by the reduction in size of the model, but it is most certainly also due to the preprocessing steps performed by CPLEX. Therefore, we may expect the technique of constraint satisfaction to work very well to find a solution of value L' if such solution exists.

The hardness of the problem seems to depend mostly on the number of jobs per machine: if we look at scenarios 8-10 we can see that 20 jobs per machine gets really difficult. However, doubling the time values (scenarios 11 and 12) adds a relatively little increase to the average time needed to solve an instance, but one problem becomes unsolvable for our hybrid algorithm. But also here our hybrid algorithm shows its merit in comparison to the ignorant time-indexed method, for doubling the times makes CPLEX incapable to solve any of the instances: it does not even find any solution for these instances, which is of course due to the large increase in variables in the ILP model.

Table 2 already shows the quality of our local search algorithm since the number of (costly) ILP solves of the pricing problem is limited and does not seem to depend much on the size or difficulty of the problem. Table 3 further validates that our local search algorithm performs very well, for without the local search algorithm *easy* instances already take 3 times as much time to compute the lower bound. And *difficult* instances even become unsolvable within 30 minutes, while our local search algorithm only needs a little more than 3 minutes on average to compute the lower bound for these instances. Next, doubling the time values also doubles the time needed to compute the lower bound, where using only ILP to solve the pricing problem quadruples the average time for 4 instances and is not able to compute a lower bound for one instance within 30 minutes.

9 Conclusion and Future Research

We have described a column generation approach for the parallel machine scheduling problem with minimax objective subject to release dates and precedence constraints. We have tested the algorithm for maximum lateness subject to release dates, but with greater than or equal precedence constraints only. We suspect that problems with maximum cost are harder, since changing the upper bound on the cost with a small amount does not necessarily change the deadlines of all jobs. We expect that the nature of the precedence constraints does not change the effectiveness of the algorithm. It is an interesting, nontrivial step to extend this algorithm to the case with *uniform*, or even *unrelated* machines.

The next step in the research will be to investigate the natural connection with constraint satisfaction, which for instance can be used to tighten the release dates and deadlines. This looks a very promising direction to improve the effectiveness of the algorithm, as already witnessed by the success of the preprocessing phase of the CPLEX algorithm.

References

1. J.M. van den Akker (1994). *LP-based solution methods for single-machine scheduling problems*, PhD Thesis, Eindhoven University of Technology.
2. J.M. van den Akker, J.A. Hoogeveen, and S.L. Van de Velde (1999). Parallel machine scheduling by column generation. *Operations Research 47*, 862–872.
3. J.M. van den Akker, J.A. Hoogeveen, and S.L. Van de Velde (2005). Applying column generation to machine scheduling. G. Desaulniers, J. Desrosiers, and M.M. Solomon (eds.). *Column Generation*, Springer, 303–330.
4. P. Brucker and S. Knust (2000). A linear programming and constraint propagation-based lower bound for the RCPSP. *European Journal of Operational Research 127*, 355–362.
5. P. Brucker and S. Knust (2002). Lower bounds for scheduling a single robot in a job-shop environment. *Annals of Operations Research 115*, 147–172.
6. P. Brucker and S. Knust (2003). Lower bounds for resource-constrained project scheduling problems. *European Journal of Operational Research 149*, 302–313.
7. Z.L. Chen and W.B. Powell (1999). Solving parallel machine scheduling problems by column generation. *INFORMS Journal on Computing 11*, 78–94.
8. M.R. Garey, R.E. Tarjan, G.T. Wilfong (1988). One-processor scheduling with symmetric earliness and tardiness penalties. *Mathematics of Operations Research 13*, 330–348.
9. R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G.Rinnooy Kan (1979). Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics 5*, 287–326.
10. J.R. Jackson (1955). *Scheduling a production line to minimize maximum tardiness*, Research Report 43, Management Sciences Research Project, UCLA.
11. E.L. Lawler and J.M. Moore (1969). A functional equation and its application to resource allocation and sequencing problems. *Management Science 16*, 77–84.
12. J.P. de Sousa and L.A. Wolsey (1992). A time-indexed formulation of non-preemptive single-machine scheduling problems. *Mathematical Programming 54*, 353–367.

Reporting Flock Patterns

Marc Benkert[1,⋆], Joachim Gudmundsson[2],
Florian Hübner[1], and Thomas Wolle[2]

[1] Department of Computer Science, Karlsruhe University, P.O. Box 6980,
D-76128 Karlsruhe, Germany
{mbenkert, fhuebner}@ira.uka.de
[2] National ICT Australia Ltd⋆⋆, Locked Bag 9013, Alexandria NSW 1435, Australia
{joachim.gudmundsson, thomas.wolle}@nicta.com.au

Abstract. Data representing moving objects is rapidly getting more available, especially in the area of wildlife GPS tracking. It is a central belief that information is hidden in large data sets in the form of interesting patterns. One of the most common spatio-temporal patterns sought after is flocks. A flock is a large enough subset of objects moving along paths close to each other for a certain pre-defined time. We give a new definition that we argue is more realistic than the previous ones, and we present fast approximation algorithms to report flocks. The algorithms are analysed both theoretically and experimentally.

1 Introduction

Data related to the movement of objects is becoming increasingly available because of substantial technological advances in position-aware devices such as GPS receivers, navigation systems and mobile phones. The increasing number of such devices will lead to huge spatio-temporal data volumes documenting the movement of animals, vehicles or people. One of the objectives of spatio-temporal data mining [12, 14] is to analyse such data sets for interesting patterns. For example, a group of 25 elks in Sweden was equipped with GPS-GSM collars. The GPS collar acquires a position every half hour and then sends the information to a GSM-modem where the positions are extracted and stored. Analysing this data gives insight into entity behaviour, in particular, migration patterns. There are many other examples where spatio-temporal data is collected [1, 13]. The analysis of moving objects also has applications in sports (e.g. soccer players [8]), in socio-economic geography [5] and in defence and surveillance areas.

The input is a set P of n moving point objects $p_1, \ldots, p_n$ whose locations are known at τ consecutive time steps $t_1, \ldots, t_\tau$, i.e. the trajectory of each object is a polygonal line, see Fig. 1a. We will call moving point objects *entities* from now on, and assume their velocity between two consecutive time steps is constant.

⋆ Supported by grant WO 758/4-2 of the German Science Foundation (DFG). Parts of this work were done while M.Benkert visited NICTA on DAAD grant D/05/08276.
⋆⋆ Funded by the Australian Government's Backing Australia's Ability initiative, in part through the Australian Research Council.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 660–671, 2006.

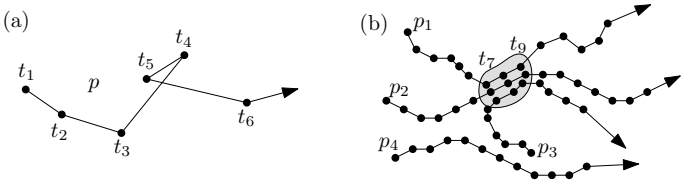


Fig. 1. (a) A polygonal line describing the movement of an entity p in the time interval $[t_1, t_6]$. (b) A flock for p_1, p_2, p_3 in the time interval $[t_7, t_9]$.

There is some research on data mining of moving objects (e.g. [15, 16, 18]) in particular, on the discovery of similar directions or clusters. Kalnis et al. detect moving clusters over many time steps [9]. However, their definition of clusters does not lead to an approach that in general finds flocks. Verhein and Chawla [18] used associated data mining to detect patterns in spatio-temporal sets. They partitioned space into cells and then defined a cell to be a *source*, *sink* or *thoroughfare* depending on the number of objects entering, exiting or passing through the cell.

Laube and Imfeld [10] proposed a different approach in 2002 - the REMO framework (RElative MOtion) which defines similar behaviour in groups of entities. They define a collection of spatio-temporal patterns based on similar direction of motion or change of direction. Laube et al. [11] extended the framework by not only including direction of motion, but also location itself. They defined several spatio-temporal patterns, including *flock*, *leadership*, *convergence* and *encounter*, and gave algorithms to compute them efficiently. In [11] they developed an algorithm for finding the largest flock pattern (maximum number of entities) using the higher-order Voronoi diagram with running time $\mathcal{O}(\tau(nm^2 + n \log n))$, they also proved that the detection problem can be answered in $\mathcal{O}(\tau(nm + n \log n))$ time. Applying the paper by Aronov and Har-Peled [2] to the problem gives a $(1 + \varepsilon)$-approximation with expected running time $\mathcal{O}(\tau n/\varepsilon^2 \log^2 n)$. Gudmundsson et al. [7] showed that if the region that should contain the entities (disk) is $(1 + \varepsilon)$-approximated then the detection problem can be solved in $\mathcal{O}(\tau(n/\varepsilon^2 \log 1/\varepsilon + n \log n))$ time.

However, the above algorithms only consider each time step separately, that is, given $m \in \mathbb{N}$ and $r > 0$ a flock is defined by at least m entities within a circular region of radius r and moving in the same direction at some point in time. We argue that this is not enough for most practical applications, e.g. a group of animals may need to stay together for days or even weeks before they define a flock. Therefore we propose the following definition of a flock:

Definition 1. *(m, k, r)-flock$_A$ - Given a set of n trajectories where each trajectory consists of $\tau - 1$ line segments, a flock in a time interval $I = [t_i, t_j]$, where $j - i + 1 \geqslant k$, consists of at least m entities such that for every point in time within I there is a disk of radius r that contains all the m entities. Note that $m, k \in \mathbb{N}$ and $r > 0$ are given constants.*

Gudmundsson and van Kreveld [6] recently showed that (in the discrete model, see Definition 2) computing the longest duration flock and the largest subset

flock is NP-hard to approximate within a factor of $\tau^{1-\varepsilon}$ and $n^{1-\varepsilon}$ respectively. They also give a 2-radius approximation algorithm for the longest duration flock with running time $\mathcal{O}(n^2\tau \log n)$.

We describe efficient approximation algorithms for reporting and detecting flocks, where we let the size of the region deviate slightly from what is specified. Approximating the size of the circular region with a factor of $\Delta > 1$ means that a disk with radius between r and Δr that contains at least m objects may or may not be reported as a flock while a region with a radius of at most r that contains at least m entities will always be reported. We present several approximation algorithms, for example, a $(2+\varepsilon)$-approximation with running time $T(n) = \mathcal{O}(\tau n k^2(\log n + 1/\varepsilon^{2k-1}))$ and a $(1+\varepsilon)$-approximation algorithm with running time $\mathcal{O}(1/(m\varepsilon^{2k}) \cdot T(n))$.

Our aim is to present algorithms that are efficient not only with respect to the size of the input (which is τn) but also try to keep the dependency on k and m as small as possible. For most of the practical applications we have seen; m will be between a couple of entities to a few hundreds or even thousands, and k is expected to be between 5 and 30 for most applications.

The paper is organised as follows. In Section 2 we show a discrete version of the definition of a flock and prove that it is equivalent to the original definition. Three approximation algorithms (all derived from a general approach) are presented in Section 3. In Section 4 we evaluate these algorithms. Note that due to space constraints proofs are omitted in this paper, they can be found in [3].

1.1 The Skip Quadtree and the Computational Model

One of the main tools used in this paper is the skip-quadtree by Eppstein et al. [4]. A small modification to the skip-quadtree results in the following lemma:

Lemma 1. *Insertion, deletion and search in the modified d-dimensional skip quadtree using a total of $\mathcal{O}(dn)$ space can be done in $\mathcal{O}(d \log n)$ time. An $(1+\delta)$-approximate range counting query for a fat convex region of complexity $\mathcal{O}(d)$ can be answered in time $T(n) = \mathcal{O}(d^2(\log n + \delta^{1-d}))$, where $\delta > 0$ is a given constant.*

The standard practice [4] in computational geometry using quadtrees is that certain operations can be done in constant time. In arithmetic terms, the computations needed to perform point location, range queries or nearest neighbour queries in a quadtree, involve finding the most significant binary digit at which the coordinates of two points differ. This can be done using a constant number of machine instructions if we have a most-significant-bit instruction, or by using floating point or extended precision normalisation.

2 Approximate Flocks

The input is a set P of n trajectories $p_1, \ldots, p_n$, where each trajectory p_i is a sequence of τ coordinates in the plane $(x_1^i, y_1^i), (x_2^i, y_2^i), \ldots, (x_\tau^i, y_\tau^i)$, and (x_j^i, y_j^i) is the position of entity p_i at time t_j. We will assume that the movement of an

entity from its position at time t_j to its position at time t_{j+1} is described by the straight-line segment between the two coordinates, and that the entity moves along the segment with constant velocity.

2.1 An Equivalent Definition of Flock

We will give an alternative and algorithmically simpler definition of a flock.

Definition 2. *(m, k, r)-flock$_B$ - Given a set of n trajectories where each trajectory consists of $\tau - 1$ line segments a flock in a time interval $[t_i, t_j]$, where $j - i + 1 \geqslant k$ consists of at least m entities such that for every discrete time step t_ℓ, $i \leqslant \ell \leqslant j$, there is a disk of radius r that contains all the m entities.*

Lemma 2. *If the entities move with constant velocity along the straight line segment between two consecutive time steps then flock$_A$ and flock$_B$ are equivalent.*

Note that the centre of a disk does not have to coincide with one of the positions of the entities. In the remainder of this paper we refer to Definition 2 whenever we talk about flocks. Definition 2 immediately suggests a new approach; for each time interval $[t_i, t_{i+k-1}]$ check whether there is a set of m entities $F = \{p_1, \ldots, p_m\}$ that can be covered by a disk of radius r at each discrete time step in $[t_i, t_{i+k-1}]$. We will show how this observation will allow us to develop an approximation algorithm.

2.2 The General Approach

When developing an algorithm for this problem one of the main hurdles that we encountered was to detect flocks without having to keep track of all the objects in a potential flock. That is, when we consider a specific time step, the number of potential flocks can be very large and the number of objects that one needs to keep track of for each potential flock might be $\Omega(n)$. In general this problem occurs whenever one attempts to develop a method that processes the input time step by time step. In this paper we avoid this problem by transforming the trajectories into higher dimensional space. Note that the gain is that we only need to count the number of points in a region, instead of keeping track of the actual objects. This might seem like overkill but both the theoretical and the experimental bounds supports this approach, at least as long as k is fairly small.

The basic idea builds upon the fact that a polygonal line with d vertices in the plane can be modelled as a single point in $2d$ dimensions. The trajectory of an entity p in the time interval $[t_i, t_j]$ is described by the polygonal line $p(i, j) = \langle (x_i, y_i), (x_{i+1}, y_{i+1}), \ldots, (x_j, y_j) \rangle$, which corresponds to a point $p'(i, j) = (x_i, y_i, x_{i+1}, y_{i+1}, \ldots, x_j, y_j)$ in $2(j - i + 1)$-dimensional space.

The first step when checking whether there is a flock in the time interval $[t_i, t_{i+k-1}]$ is to map the polygonal lines of all entities to $\mathbb{R}^{2k}$. Equivalence 1 gives the key characterisation of flocks. First, we define an (x, y, i, r)-*pipe* which is an unbounded region in $\mathbb{R}^{2k}$. Such a pipe contains all the points that are only restricted in two of the $2k$ dimensions (namely in dimensions i and $i + 1$) and

when projected on those two dimensions lie in a circle of radius r around the point (x, y). Formally, a (x, y, i, r)-pipe is the following region:

$$\{(x_1, \ldots, x_{2k}) \in \mathbb{R}^{2k} \mid (x_i - x)^2 + (x_{i+1} - y)^2 \leqslant r^2\}.$$

Equivalence 1. *Let $F = \{p_1, \ldots, p_m\}$ be a set of entities and $I = [t_1, t_k]$ a time interval. Let $\{p'_1, \ldots, p'_m\}$ be the mapping of F to $\mathbb{R}^{2k}$ w.r.t. I. It holds that: F is a (m, k, r)-flock $\iff \exists x_1, y_1, \ldots, x_k, y_k : \forall p \in F : p' \in \bigcap_{i=1}^{k} (x_i, y_i, 2i-1, r)$-pipe.*

To see that this equivalence holds we observe the following: for each time step $t_i \in I$ the disk with radius r and centre (x_i, y_i) contains the entity positions $p_1^i, \ldots, p_m^i$. We will show that approximation algorithms can be obtained by performing a set of range counting queries in higher dimensional space.

3 Approximation Algorithms

We now give approximation algorithms where the radius r is approximated.

Method 1: A $(\sqrt{8}+\varepsilon)$-Approximation Algorithm (Box). Using the general idea discussed in Section 2.2 we will develop a $(\sqrt{8} + \varepsilon)$-approximation algorithm. For each time interval $I = [t_i, t_{i+k-1}]$, where $1 \leqslant i \leqslant \tau - k + 1$, we will do the following computations. For simplicity our first method uses a $2k$-dimensional box to approximate the region of a potential flock.

For each entity p let p' denote the mapping of p to $\mathbb{R}^{2k}$ with respect to I. Construct a skip quadtree T for the point set $P' = \{p'_1, \ldots, p'_n\}$. Then, for each point $p' \in P'$ and an appropriately chosen $\delta > 0$ perform a $(1 + \delta)$-approximate range counting query in T where the query range $Q(p')$ is a $2k$-dimensional cube. As we do not know the centre of a potential flock we choose to query around p' and any flock that contains p is within distance $2r$ from p. Hence, our query box has side length $4r$ and centre at p'. We $(1 + \delta)$-approximate the $2k$-dimensional cube which is itself a $\sqrt{8}$-approximation for the query region (see Fig. 2a) in two dimensions. Thus, every counting query containing at least m entities corresponds to an $(m, k, (\sqrt{8} + \varepsilon)r)$-flock as stated in Lemma 3. Note that the same flock may be reported several times.

Lemma 3. *Method 1 is a $(\sqrt{8}+\varepsilon)$-approximation algorithm and requires $\mathcal{O}(\tau n)$ space and $\mathcal{O}(\tau n k^2 (\log n + 1/\varepsilon^{2k-1}))$ time.*

Method 2: A $(2+\varepsilon)$-Approximation Algorithm (Pipes). The algorithm is similar to the above algorithm. The main difference is that we will use the intersection of k pipes as the query regions instead of the $2k$-dimensional box. For each time interval $I = [t_i, t_{i+k-1}]$, where $1 \leqslant i \leqslant \tau - k + 1$, we will do the following computations.

Construct a skip quadtree T for the point set $P' = \{p'_1, \ldots, p'_n\} \in \mathbb{R}^{2k}$ as in Method 1. Then, for each point $p' \in \mathcal{P}'$ perform a $(1 + \varepsilon)$-approximate range counting query in T where the query range $Q(p')$ is the intersection of the k pipes $(x_i, y_i, 2i - 1, 2r)$ and (x_i, y_i) is the position of p at time step t_i (see Fig. 2b).

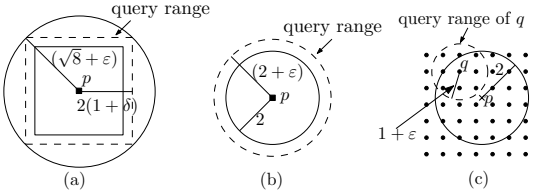


Fig. 2. Illustration of the approximative range of methods 1,2 and 3 for $r = 1$. The dashed region is the query region.

Recall that since the query region is convex and fat we can apply Lemma 1. The definition of fatness we use was introduced by van der Stappen [17].

Definition 3 ([17]). *Let $\alpha > 1$ be a real value. An object s is α-fat if for any d-dimensional ball D whose centre lies in s and whose boundary intersects s, we have* $\text{volume}(D) \leqslant \alpha \cdot \text{volume}(s \cap D)$.

Lemma 4. *The intersection of d pipes $(x_i, y_i, 2i-1, 2r)$, $1 \leqslant i \leqslant k$, in $2d$-dimensional space is a bounded convex 4^d-fat region whose boundary consists of $\mathcal{O}(d)$ surfaces of quadratic complexity.*

Lemma 5. *Method 2 is a $(2+\varepsilon)$-approximation algorithm and requires $\mathcal{O}(\tau n)$ space and $\mathcal{O}(\tau n k^2(\log n + 1/\varepsilon^{2k-1}))$ time.*

Remark 1. A comparison between Lemmas 3 and 5 shows that even though the approximation factor of the second method is smaller the running time is identical. However, this is a theoretical bound, in practice we chose to implement the second method using a compressed quadtree for which we only have to decide whether the intersection of a d-dimensional cell and the k pipes is non-empty, while for the skip-quadtree we have to compute the volume of this intersection which is possible in theory but hard in practice. Consequently, the experiments performed with methods 1 and 2 use different data structures.

Method 3: A (1+ε)-Approximation Algorithm. We use the same approach as above but instead of querying only the input points in $\mathbb{R}^{2k}$ we will now query $\mathcal{O}(1/\varepsilon^{2k})$ sample points for each entity point. For each time interval $I = [t_i, t_{i+k-1}]$, where $1 \leqslant i \leqslant \tau - k + 1$, we will do the following computations.

Construct a skip quadtree T for the point set $P' = \{p'_1, \ldots, p'_n\} \in \mathbb{R}^{2k}$ as in Method 1 and 2. Let Γ be the vertices of a regular grid in $\mathbb{R}^{2k}$ of spacing $\varepsilon \cdot r/2$. Each input point p'_i generates the sample set $\Gamma \cap D(p'_i)$ where $D(p'_i)$ is the $2k$-dimensional ball of radius $2r$ centred at p'_i. Clearly, this gives rise to $\mathcal{O}(1/\varepsilon^{2k})$ sample points for each entity (see Fig. 2c).

Next, we perform a $(1 + \frac{\varepsilon}{2+\varepsilon})$-approximate range counting query in T for each sample point $(x_1, y_1, \ldots, x_k, y_k)$ where the query range is the intersection of the k pipes $(x_i, y_i, 2i-1, (1+\varepsilon/2)r)$, $1 \leqslant i \leqslant k$. However, a necessary condition for a sample point q to induce an (m, k, r)-flock is that there are at least m entities in the disk $D(q)$ of radius $2r$ centred at q. During the processing of the sample points we can count how many entities indeed lie in $D(q)$ for each sample point

q. As we generate at most $\mathcal{O}(n/\varepsilon^{2k})$ sample points, this means that we have to check at most $\mathcal{O}(n/(m\varepsilon^{2k}))$ candidate sample points for inducing a flock. Next we prove the approximation bound.

Lemma 6. *Method 3 is a* $(1+\varepsilon)$*-approximation algorithm and requires* $\mathcal{O}(\tau n)$ *space and* $\mathcal{O}(\frac{\tau n k^2}{m\varepsilon^{2k}}(\log n + 1/\varepsilon^{2k-1}))$ *time.*

4 Experiments

We used a Linux operated off-the-shelf PC with an Intel Pentium-4 3.6 GHz processor and 2 GB of main memory. The data structures and algorithms were implemented and compiled with the Gnu C++ compiler. Our point sets used in the experiments were created artificially. Each point coordinate of an input point is an integer from the interval $[0, ..., 2^{13}]$ or $[0, ..., 2^{16}]$. The point sets differ in size (10,000 - 160,000 points; one algorithm was run with more than 1 million points), in length of the time interval (4 - 16 time steps) and also in the distribution of the points (uniformly random or clustered). This was done to see the impact of the different characteristics on the algorithms behaviour. In all point sets, 10% of the points were placed in such a way that they form (randomly positioned) flocks of $m = 50$ entities in a circle of radius $r = 50$. The distribution and density of the clusters were chosen not to considerably increase the number of flocks found by the algorithms. This makes a comparison between the results for clustered and uniformly randomly distributed point sets easier. Note that each generated data instance contains the coordinates of points for a certain number of time steps τ, and in the experiments on that instance, we always looked for (m, k, r)-flocks with $m = r = 50$ and $k = \tau$.

4.1 Methods

We compare the results of four methods called 'box', 'pipes', 'no-tree' and 'pruning'. The box and pipes method are explained in Section 3 and use a skip-quadtree or a compressed quadtree, respectively.

The no-tree method (which was implemented for comparison) is a 2-approximation and does not use a tree structure. It has two nested loops (each running over all input points), the outer one specifying a potential flock centre and the inner one computing the distance between a point and the potential flock centre. If there are enough points within a ball (around the potential flock centre) of double flock-radius then we found a flock.

The pruning method takes advantage of the fact that all points not involved in flocks of length $k^* < k$ cannot be involved in flocks of length k. It works as follows: At first, we compute flocks of length $k^* = 4$ using the box method. Then we build a new tree containing only those points that were contained in flocks during the first step. This drastically reduces the number of points. We then again perform the box method on the new tree for the entire length $k = \tau$.

A set of entities can have many flocks and even one single entity can be involved in several flocks, e.g. a flock involving $m+1$ entities implies the existence

of $m+1$ flocks of cardinality m that pairwise differ by one entity. We must specify what we want to find and report in a given data set, see [7] for a discussion. The general approach described in Section 3 has the following disadvantage: As every entity is tested, a flock consisting of exactly m elements can be reported up to m times. We chose to use an approach in the experiments which guarantees a high level of correctness while bounding the number of flocks that an entity may simultaneously belong to. The idea is that when a flock is found every query point within the query region will be marked, so that no query will be performed with those marked points as centres. Using a simple packing argument it follows that the maximal number of flocks an entity can be part of during a time step is bounded by $\mathcal{O}(2^{2k})$. The additional time that we have to spend updating the tree is $\mathcal{O}(nk \log n)$ per time step, thus $\mathcal{O}(\tau nk \log n)$ in total.

4.2 Results

We run the experiments with a series of generated point-sets for each combination of point-set characteristics. The results were very similar for fixed characteristics and hence the tables below show the numbers for only one collection of point-sets with the specified characteristics. The results of the algorithms for $\varepsilon = 0.05$ are depicted in Table 1, where the coordinates of the points are chosen from the interval $[0, ..., 2^{16}]$. The columns below 'input' specify the number of points and the number of time steps, and the columns below 'uniformly' and 'clustered' show the number of flocks found (our algorithms also output the size and the centre of those flocks) and the running times (in seconds) needed when performing the box-, pipes- and no-tree-algorithm on the corresponding input. We also performed the same experiments on point-sets where the coordinates where chosen from $[0, ..., 2^{13}]$. Table 2 shows those results. The results for the method with pruning are given in Table 3. Because of the similarity of the results for a different number of time steps, we only report the results for 16 time steps in that table. We also report the number of flocks found (indicated in *italics* if it deviates from the number of artificially inserted flocks) to ensure and verify that our methods indeed find them. The dependencies of the running times are a more important result than the number of flocks.

4.3 Discussion

Flat trees in high dimensions. One obvious observation is that the running times of our algorithms are increasing with the number of time steps (i.e. with the number of dimensions d). Recall that an internal node of a quadtree has 2^d children. Using 16 time steps means 32 dimensions which translates to more than 4 billion quadrants, i.e. children of an internal node (in our approach we only store non-empty children in a list, which reduces storage space but increases time complexity). In an experiment with 160K points in 32 dimensions it is not likely that many of the random points (not in flocks) fall into the same quadrant. Therefore the tree is very flat (i.e. have only a very small depth, but large width), which results in high running times.

Table 1. Results for $\varepsilon = 0.05$ and point-sets with coordinates from $[0, ..., 2^{16}]$

input		uniformly						clustered					
		box		pipes		no-tree		box		pipes		no-tree	
n	τ	flocks	time	flocks	time	flocks	time	flocks	time	flocks	time	flocks	time
10K	4	20	0	20	0	20	5	20	0	20	1	20	5
10K	8	20	2	20	1	20	5	20	1	20	0	20	5
10K	16	20	2	20	1	20	6	20	1	20	0	20	5
20K	4	40	1	*41*	0	40	21	40	0	40	1	40	20
20K	8	40	7	40	5	40	21	40	1	40	0	40	22
20K	16	40	13	40	10	40	25	40	3	40	2	40	25
40K	4	80	0	80	1	80	83	80	1	80	0	80	83
40K	8	80	32	80	22	80	87	80	2	80	2	80	87
40K	16	80	62	80	44	80	99	80	8	80	7	80	101
80K	4	160	3	*163*	3	160	332	160	3	160	2	160	332
80K	8	160	129	160	88	160	347	160	6	160	4	160	346
80K	16	160	244	160	182	160	392	160	30	160	29	160	392
160K	4	320	8	*321*	10	320	1326	320	8	320	5	320	1327
160K	8	320	441	320	316	320	1391	320	20	320	15	320	1384
160K	16	320	986	320	768	320	1576	320	102	320	93	320	1564

Error value ε. When performing a range query, ε influences the approximate region to be queried. One could expect that a larger value of ε leads to shorter running times and more flocks, because the descent in the tree can be stopped earlier. However, apart from marginal fluctuations, this behaviour could not be observed in our experiments. Our trees in the experiments are rather sparsely filled. Hence, the squares corresponding to most of the leaves in the tree (which correspond to single points in a point set) are still quite large compared to the approximated flock radius $(1 + \varepsilon)r$. Furthermore, it often seems that the point sets are too sparse to find any random flocks. Therefore we refrained from reporting results for different ε and only used $\varepsilon = 0.05$.

Number of flocks. Most of the times the algorithms found exactly as many flocks as were artificially inserted. A few times more flocks were found, i.e. some of the randomly positioned points created flocks by chance. This happened only in instances with a small number of time steps, which is reasonable as it is more likely for random points to form flocks only for a small number of time steps. In one case more than 1300 flocks were found which indicates that for that instance and for that characteristic the distribution of the points and clusters reached a limit were the clusters are dense enough to often create random flocks. In some of our experiments we observed that the algorithms found less flocks than were inserted. This can happen if two flocks are close to each other and fall into one query region and hence will be counted as one flock by the algorithm.

Coordinate space $[0, ..., 2^{13}]$ vs. $[0, ..., 2^{16}]$. The experiments with coordinate space $[0, ..., 2^{16}]$, i.e. where each coordinate of a point is in $[0, ..., 2^{16}]$, were

Table 2. Results for $\varepsilon = 0.05$ and point-sets with coordinates from $[0, ..., 2^{13}]$

input		uniformly						clustered					
		box		pipes		no-tree		box		pipes		no-tree	
n	τ	flocks	time	flocks	time	flocks	time	flocks	time	flocks	time	flocks	time
10K	4	20	1	20	0	20	5	20	2	20	1	20	4
10K	8	20	8	20	6	20	6	20	2	20	2	20	6
10K	16	20	14	20	11	20	6	20	5	20	11	20	6
20K	4	40	1	40	4	40	20	40	2	40	1	40	20
20K	8	40	52	40	35	40	22	40	6	40	4	40	22
20K	16	40	83	40	58	40	25	40	17	40	44	40	25
40K	4	80	4	80	15	80	83	*81*	6	80	2	80	83
40K	8	80	237	80	166	80	87	80	16	80	21	80	87
40K	16	80	347	80	244	80	99	80	55	80	177	80	99
80K	4	160	10	160	57	160	333	*206*	16	160	8	160	332
80K	8	160	932	160	696	160	348	160	45	160	77	160	348
80K	16	160	1411	160	1124	160	394	160	164	160	594	160	395
160K	4	320	29	320	201	320	1326	*1317*	42	320	27	320	1331
160K	8	320	3179	320	2658	320	1393	320	124	320	238	320	1392
160K	16	320	6015	320	4226	320	1575	320	692	320	2306	320	1576

comparatively much faster than those with coordinate space $[0, ..., 2^{13}]$. One explanation is that the query region is relatively larger in the coordinate space $[0, ..., 2^{13}]$. Also, in a bigger underlying coordinate space it is more likely that the query region falls into a single quadrant of a quadtree. Due to the sparseness of the point-sets the algorithms are likely to find just a single point in that quadrant. On the other hand in a smaller underlying space the query region might intersect more quadrants, which results in more subsequent queries.

Uniformly vs. clustered. We can observe that our tree-based algorithms almost always perform better on the clustered point-sets. This behaviour could be expected because, as we have seen from the experiments in general, uniformly distributed points result in quadtrees that are rather flat (especially for higher dimensions). But it is a 'good balance' between height and width of a tree that allows fast query times. Clustered data sets are more likely to create trees that are faster descended by the algorithms. The no-tree method (which is not using a tree) is not affected by the two different types of data.

No-tree vs. box vs. pipe. The no-tree method's running times are quadratic in the number of points and not influenced by the number of time steps, as expected. On the other hand the box and pipes algorithms are strongly influenced by the number of time steps and the number of points. A large query region in combination with a small coordinate space causes their behaviour to become similar (although with a big overhead) to the no-tree method. The difference between the box and pipes method is caused by the different data structure they use. That is why the pipe method is almost always faster than the box method.

Table 3. Results for pruning method, $\varepsilon = 0.05$

input		coordinates from $[0, ..., 2^{13}]$				coordinates from $[0, ..., 2^{16}]$			
		uniformly		clustered		uniformly		clustered	
		pruning		pruning		pruning		pruning	
n	τ	flocks	time	flocks	time	flocks	time	flocks	time
10K	16	20	0	20	1	20	1	20	0
20K	16	40	1	40	2	40	1	40	0
40K	16	80	3	80	6	80	2	80	2
80K	16	160	11	160	15	160	3	160	3
160K	16	320	30	320	45	320	9	320	9
320K	16	*639*	82	*633*	303	640	26	640	25
640K	16	*1271*	194	*1268*	1796	1280	75	1280	75
1280K	16	*2501*	533	*2507*	9213	2560	249	2560	246

Pruning. Table 3 shows the impressive impact of the pruning step. Depending on the density and distribution, even some point-sets with more than 1 million points can be dealt with in a couple of minutes. Furthermore, we observed that the number of time steps has almost always no effect on the running times. An exception to this are the clustered point sets with many points and with coordinates in $[0, ..., 2^{13}]$, where we experienced much longer running times. (Because of space restrictions, we only give the numbers for 16 time steps.) This can be explained by noting that after the pruning step it is likely that the remaining points form a flock also for more time steps (as intended). Therefore, almost every query to the datastructure leads to finding a flock and hence, the number of queries is drastically decreased. For the clustered point sets with coordinates in $[0, ..., 2^{13}]$, however, the probability of random flocks is higher, because the query region is comparatively large. The fact that the pruning method sometimes finds less flocks than the box method can be explained by noting that the pruning method performs two runs of the box method each of which can handle the points in a different order. Therefore the second run of the box method can encounter points which will not belong to any flock.

5 Conclusions and Open Problems

This paper is a first step towards practical algorithms for finding spatio-temporal patterns, such as flocks, encounters and convergences. Future research does not only include more efficient approaches to compute these patterns but also more complicated patterns, e.g. hierarchical patterns or repetitive patterns. In this paper we have presented different algorithms for finding flock patterns and analysed them theoretically as well as experimentally. From the experiments we have seen that our algorithms can perform very well, especially for a small number of time steps. However, their running times depend very much on the characteristics of the input point-sets, which motivates more research and experiments, preferably on real-world data.

References

1. Wildlife tracking projects with GPS GSM collars, 2006. `http://www.environmental-studies.de/projects/projects.html`.
2. B.Aronov and S. Har-Peled. On approximating the depth and related problems. In *Proc. of 16th ACM-SIAM Symp. on Discrete Algorithms*, pages 886–894, 2005.
3. M. Benkert, J. Gudmundsson, F. Hübner, and T. Wolle. Reporting flock patterns. Technical Report 2006-14, Fakultät für Informatik, Universität Karlsruhe, 2006. `http://www.ubka.uni-karlsruhe.de/indexer-vvv/ira/2006/14`.
4. D. Eppstein, M. T. Goodrich, and J. Z. Sun. The skip quadtree: a simple dynamic data structure for multidimensional data. In *Proc. 21st ACM Symposium on Computational Geometry*, pages 296–305, 2005.
5. A.U. Frank, J.F. Raper, and J.-P. Cheylan, editors. *Life and motion of spatial socio-economic units*. Taylor & Francis, London, 2001.
6. J. Gudmundsson and M. van Kreveld. Computing longest duration flocks in spatio-temporal data. Manuscript, April 2006.
7. J. Gudmundsson, M. van Kreveld, and B. Speckmann. Efficient detection of motion patterns in spatio-temporal sets. In *Proceedings of the 13th International Symposium of ACM Geographic Information Systems*, 2004.
8. S. Iwase and H. Saito. Tracking soccer player using multiple views. In *Proc. of the IAPR Workshop on Machine Vision Applications (MVA02)*, pages 102–105, 2002.
9. P. Kalnis, N. Mamoulis, and Spiridon Bakiras. On discovering moving clusters in spatio-temporal data. In *Proc. of the 9th Int. Symp. on Spatial and Temporal Databases (SSTD05)*, volume 3633 of *LNCS*, pages 364–381. Springer-Verlag, 2005.
10. P. Laube and S. Imfeld. Analyzing relative motion within groups of trackable moving point objects. In *GIScience 2002*, number 2478 in Lecture Notes in Computer Science, pages 132–144. Springer, Berlin, 2002.
11. P. Laube, M. van Kreveld, and S. Imfeld. Finding REMO – detecting relative motion patterns in geospatial lifelines. In *Developments in Spatial Data Handling: Proceedings of the 11th Int. Symp. on Spatial Data Handling*, pages 201–214, 2004.
12. H.J. Miller and J. Han, editors. *Geographic Data Mining and Knowledge Discovery*. Taylor & Francis, London, 2001.
13. Porcupine caribou herd satellite collar project. `http://www.taiga.net/satellite/`.
14. J. Roddick, K. Hornsby, and M. Spiliopoulou. An updated bibliography of temporal, spatial, and spatio-temporal data mining research. In *TSDM 2000*, Lecture Notes in Artificial Intelligence, vol.2007, pages 147–163. Springer, Berlin, 2001.
15. C.-B. Shim and J.-W.Chang. A new similar trajectory retrieval scheme using k-warping distance algorithm for moving objects. In *Proc. of the 4th Int. Conf. on Advances in Web-Age Information Management, (WAIM 2003)*, number 2762 in Lecture Notes in Computer Science, pages 433–444. Springer, Berlin, 2003.
16. N. Sumpter and A. J. Bulpitt. Learning spatio-temporal patterns for predicting object behaviour. *Image Vision and Computing*, 18(9):697–704, 2000.
17. A. F. van der Stappen. *Motion Planning amidst Fat Obstacles*. PhD thesis, Department of Computer Science, Utrecht University, The Netherlands, 1994.
18. F. Verhein and S. Chawla. Mining spatio-temporal association rules, sources, sinks, stationary regions and thoroughfares in object mobility databases. In *Proc. of the Workshop on Temporal Data Mining: Algorithms, Theory and Applications*, 2005.

On Exact Algorithms for Treewidth*

Hans L. Bodlaender[1], Fedor V. Fomin[2], Arie M.C.A. Koster[3], Dieter Kratsch[4], and Dimitrios M. Thilikos[5]

[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 Informatics, University of Bergen, N-5020 Bergen, Norway
fomin@ii.uib.no
[3] Konrad-Zuse-Zentrum für Informationstechnik Berlin, Takustraße 7, D-14195 Berlin, Germany
koster@zib.de
[4] LITA, Université Paul Verlaine - Metz, 57045 Metz Cedex 01, France
kratsch@univ-metz.fr
[5] Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, Edifici Ω, c/ Jordi Girona 1-3, E-08034 Barcelona, Spain
sedthilk@lsi.upc.edu

Abstract. We give experimental and theoretical results on the problem of computing the treewidth of a graph by exact exponential time algorithms using exponential space or using only polynomial space. We first report on an implementation of a dynamic programming algorithm for computing the treewidth of a graph with running time $O^*(2^n)$. This algorithm is based on the old dynamic programming method introduced by Held and Karp for the Traveling Salesman problem. We use some optimizations that do not affect the worst case running time but improve on the running time on actual instances and can be seen to be practical for small instances. However, our experiments show that the space used by the algorithm is an important factor to what input sizes the algorithm is effective. For this purpose, we settle the problem of computing treewidth under the restriction that the space used is only polynomial. In this direction we give a simple $O^*(4^n)$ algorithm that requires *polynomial* space. We also prove that using more refined techniques with balanced separators, Treewidth can be computed in $O^*(2.9512^n)$ time and polynomial space.

1 Introduction

The use of treewidth in several application areas requires efficient algorithms for computing the treewidth and optimal width tree decompositions of given graphs.

* This research was partially supported by the project *Treewidth and Combinatorial Optimization* with a grant from the Netherlands Organization for Scientific Research NWO and by the Research Council of Norway and by the DFG research group "Algorithms, Structure, Randomness" (Grant number GR 883/9-3, GR 883/9-4). The research of the last author was supported by the Spanish CICYT project TIN-2004-07925 (GRAMMARS).

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 672–683, 2006.

In the past years, a large number of papers appeared studying the problem to determine the treewidth of a graph, including both theoretical and experimental results, see e.g., [4] for an overview. Since the problem is NP complete [1], there is a little hope in finding an algorithm which can determine the treewidth of a graph in polynomial time. There are several exponential time (exact) algorithms known in the literature for the treewidth problem. (See the surveys [10,21] for an introduction to the area of exponential algorithms.) Arnborg et al. [1] gave an algorithm that tests in $O(n^{k+2})$ time if a given graph has treewidth at most k. It is not hard to observe that the algorithm runs for variable k in $O^*(2^n)$ time[1]. See also [18]. In 2004, Fomin et al. [11] presented an $O(1.9601^n)$ algorithm to compute the treewidth based on minimal separators and potential maximal cliques of graphs, using the paradigms introduced by Bouchitté and Todinca [7,6]. The analysis of the algorithm of Fomin et al. from [11] was improved by Villanger [20], who showed that the treewidth of a graph can be computed in $O(1.8899^n)$ time. While the algorithms from [11,20] provide the best known running time, they are based on computations of potential maximal cliques and are difficult to implement.

In this paper we try another approach to compute the treewidth, which seems to be much more suitable for implementations. While Treewidth is usually formulated as the problem to find a tree decomposition of minimum width, it is possible to formulate it as a problem to find a linear ordering of (the vertices of) the graph such that a specific cost measure of the ordering is as small as possible. Several existing algorithms and heuristics for treewidth are based on this linear ordering characterization of treewidth, see e.g., [2,8,12]. In this paper, we exploit this characterization again, and a lesser known property of the characterization. Thus, we can show that an old dynamic programming method, introduced by Held and Karp for the Traveling Salesman problem [15] in 1962 can be adapted and used to compute the treewidth of given graphs. Suppressing polynomial factors, time and space bounds of the algorithm for treewidth is the same as that of the algorithm of Held and Karp for Traveling Salesman: $O^*(2^n)$ running time and $O^*(2^n)$ space. The Held-Karp algorithm tabulates some information for pairs (S, v), where S is a subset of the vertices, and v is a vertex (from S); a small variation of the scheme allows us to save a factor $O(n)$ on the space for the problems considered in this paper: we tabulate information for all subsets $S \subseteq V$ of vertices.

We have carried out experiments that show that the method works well to compute the treewidth of graphs of size up to around forty to fifty. For larger graphs, the space requirements of the algorithm appear to be the bottleneck. Thus, this raises the question: are there polynomially space algorithms to compute the treewidth having running time of the form $O^*(c^n)$ for some constant c? In this paper we answer this question in the affirmative. We show that there is an algorithm to compute the treewidth that uses $O^*(4^n)$ time and only polynomial space. It uses a simple recursive divide-and-conquer technique and is similar to the polynomial space algorithm of Gurevich and Shelah [14] for TSP.

[1] We sometimes use O^*-notation which is a modified O-notation introduced by Woeginger [21] suppressing all polynomially bounded factors.

Finally, we further provide theoretical results improving upon the running time for the polynomial space algorithm for Treewidth. Using balanced separators, we obtain an algorithm for Treewidth that uses $O^*(2.9512^n)$ time and polynomial space. As we expect that in practical cases, the algorithm will use too much time, we do not provide an experimental evaluation of the algorithm.

2 Algorithms for Treewidth

2.1 Definitions

We assume the reader to be familiar with standard notions from graph theory. Throughout this paper, $n = |V|$ denotes the number of vertices of graph $G = (V, E)$. A graph $G = (V, E)$ is *chordal*, if every cycle in G of length at least four has a chord, i.e., there is an edge connecting two non-consecutive vertices in the cycle. A *triangulation* of a graph $G = (V, E)$ is a graph $H = (V, F)$ that contains G as subgraph ($F \subseteq E$) and is chordal. $H = (V, F)$ is a *minimal triangulation* of $G = (V, E)$ if H is a triangulation of G and there does not exist a triangulation $H' = (V, F')$ of G with H' a proper subgraph of H.

Definition 1. *A* tree decomposition *of a graph $G = (V, E)$ is a pair $(\{X_i \mid i \in I\}, T = (I, F))$ with $\{X_i \mid i \in I\}$ a collection of subsets of V, called* bags*, and $T = (I, F)$ a tree, such that*

- *for all $v \in V$, there exists an $i \in I$ with $v \in X_i$,*
- *for all $\{v, w\} \in E$, there exists an $i \in I$ with $v, w \in X_i$,*
- *for all $v \in V$, the set $I_v = \{i \in I \mid v \in X_i\}$ forms a connected subgraph (subtree) of T.*

The width *of tree decomposition $(\{X_i \mid i \in I\}, T = (I, F))$ equals $\max_{i \in I} |X_i| - 1$. The* treewidth *of a graph G, $\mathbf{tw}(G)$, is the minimum width of a tree decomposition of G.*

2.2 Treewidth as a Linear Ordering Problem

It is well known that treewidth can be formulated as a linear ordering problem, and this is exploited in several algorithms for treewidth, see e.g., [2,12,8,9].

A *linear ordering* of a graph $G = (V, E)$ is a bijection $\pi : V \to \{1, 2, \ldots, |V|\}$. For a linear ordering π and $v \in V$, we denote by $\pi_{<,v}$ the set of vertices that appear before v in the ordering: $\pi_{<,v} = \{w \in V \mid \pi(w) < \pi(v)\}$. Likewise, we define $\pi_{\leq,v}$, $\pi_{>,v}$, and $\pi_{\geq,v}$. A linear ordering π of G is a *perfect elimination scheme*, if for each vertex, its higher numbered neighbors form a clique, i.e., for each $i \in \{1, 2, \ldots, |V|\}$, the set $\{\pi^{-1}(j) \ : \ \{\pi^{-1}(i), \pi^{-1}(j)\} \in E \wedge j > i\}$ is a clique. A graph has a perfect elimination scheme, if and only if it is chordal, see [13, Chapter 4].

For arbitrary graphs G, a linear ordering π defines a triangulation H of G that has π as perfect elimination scheme. The *triangulation with respect to* π of

G is built as follows: first, set $G_0 = G$, and then for $i = 1$ to n, G_i is obtained from G_{i-1} by adding an edge between each pair of non adjacent higher numbered neighbors of $\pi^{-1}(i)$. One can observe that the resulting graph $H = G_n$ is chordal, has π as perfect elimination scheme, and contains G as subgraph.

For our algorithms, we want to avoid working with the triangulation explicitly. The following predicate allows us to 'hide' the triangulation. For a linear ordering π, and two vertices $v, w \in V$, we say $P_\pi(v, w)$ holds, if and only if there is a path $v, x_1, x_2, \ldots, x_r, w$ from v to w in G, such that for each i, $1 \leq i \leq r$, $\pi(x_i) < \pi(v)$, and $\pi(x_i) < \pi(w)$. In other words, $P_\pi(v, w)$ is true, if and only if there is a path from v to w such that all internal vertices are before v and w in the ordering π. Note that the definition implies that $P_\pi(v, w)$ is always true when $v = w$ or when $\{v, w\} \in E$.

With $R_\pi(v)$, we denote the number of higher numbered vertices $w \in V$ for which $P_\pi(v, w)$ holds, i.e., $R_\pi(v) = |\{w \in V \mid \pi(w) > \pi(v) \wedge P_\pi(v, w)\}|$. The proof of the following proposition is an immediate consequence of a lemma of Rose et al. [17]. (See also [3,8,9].)

Proposition 1. *Let $G = (V, E)$ be a graph, and k a non-negative integer. The treewidth of G is at most k iff there is a linear ordering π of G, such that for each $v \in V$, $R_\pi(v) \leq k$.*

2.3 A Dynamic Programming Algorithm for Treewidth

The results of this section are based on the observation that the value $R_\pi(v)$ only depends on v, G, and the set of vertices left of v in π.

Let for sets S, Q, $Q \subseteq S$, $\Pi(S)$ be the set of all permutations of S, and $\Pi(S, Q)$ be the set of all permutations of S that end with the vertices in Q. For a set of vertices $S \subseteq V$ and a vertex $v \in V - S$, we define

$$Q(S, v) = |\{w \in V - S - \{v\} \mid \text{there is a path from } v \text{ to } w \text{ in } G[S \cup \{v, w\}]|\}$$

$$TW(S) = \min_{\pi \in \Pi(S)} \max_{v \in S} Q(\pi_{<,v}, v).$$

Let us note that $Q(S, v)$ can be computed in time $O(n + m)$ by checking for each $w \in V - S - \{v\}$ whether w has a neighbor in the component of $G[S \cup \{v\}]$ containing v, and that

$$\mathbf{tw}(G) = \min_{\pi \in \Pi(V)} \max_{v \in V} R_\pi(v) = \min_{\pi \in \Pi(V)} \max_{v \in V} Q(\pi_{<,v}, v) = TW(V)$$

Lemma 1. *For any graph $G = (V, E)$, and any set of vertices $S \subseteq V$, $S \neq \emptyset$,*

$$TW(S) = \min_{v \in S} \max\{TW(S - \{v\}), Q(S - \{v\}, v)\}.$$

Theorem 1. *The treewidth of a graph on n vertices can be determined in $O^*(2^n)$ time and $O^*(2^n)$ space.*

Proof. By Lemma 1, we almost directly obtain a Held-Karp-like dynamic programming algorithm for the problem. In order of increasing sizes, we compute for each $S \subseteq V$, $TW(S)$ using Lemma 1. The algorithm uses $O^*(2^n)$ time, as we do polynomially many steps per subset of V. The algorithm also keeps all subsets of V and thus uses $O^*(2^n)$ space. □

2.4 A Recursive Algorithm for Treewidth

For vertex subsets $L, S \subseteq V, S \cap L = \emptyset$ of a graph $G = (V, E)$ we define

$$\mathbf{t}(L, S) = \min_{\pi \in \Pi(S)} \max_{v \in S} Q(L \cup \pi_{<,v}, v).$$

The intuition behind $\mathbf{t}(L, S)$ is as follows: we investigate the resulting cost of the 'best' ordering of the vertices in S, assuming that all vertices in L are left of all vertices in S, and all vertices in $V - (L \cup S)$ are right of all vertices in S. We observe that if $S = \{v\}$, then $\mathbf{t}(L, S) = Q(L, v)$. Also, by definition, $\mathbf{t}(\emptyset, S) = TW(S)$ and therefore $\mathbf{tw}(G) = \mathbf{t}(\emptyset, V)$.

Lemma 2. *Let $G = (V, E)$ be a graph, let $S \subseteq V$, $|S| \geq 2$, $L \subseteq V$, $L \cap S = \emptyset$, $k = \lfloor |S|/2 \rfloor$. Then*

$$\mathbf{t}(L, S) = \min_{S' \subseteq S, |S'|=k} \max \{\mathbf{t}(L, S'), \mathbf{t}(L \cup S', S - S')\}$$

Theorem 2. *The treewidth of a graph on n vertices can be determined in $O^*(4^n)$ time and polynomial space.*

Proof. Lemma 2 is used to obtain Algorithm 1. This algorithm computes $\mathbf{t}(L, S)$ recursively. Algorithm 1 computes the treewidth of the graph G when calling Recursive-Treewidth(G,$\emptyset$,V). It is not hard to show that the algorithm uses polynomial space and $O^*(4^n)$ time. □

Algorithm 1. Recursive-Treewidth(Graph G, Vertex Set L, Vertex Set S)

```
if |S|=1 then
  Suppose S = {v}.
  return Q(L, v)
end if
Set Opt = ∞.
for all sets S' ⊆ S, |S'| = ⌊|S|/2⌋ do
  Compute v1 = Recursive-Treewidth(G, L, S');
  Compute v2 = Recursive-Treewidth(G, L ∪ S', S − S');
  Set Opt = min {Opt, max {v1, v2}};
end for
return Opt
```

3 Experimental Results

In this section, we comment on the experiments we have carried out for the dynamic programming algorithm for computing the treewidth of a given graph.

For practical considerations, we use a scheme that is slightly different than that of Theorem 1. We can note that it is not useful to perform computations with sets S for which $TW(S)$ is larger or equal than a known upper bound up on the treewidth of G: these cannot lead to a smaller bound on the treewidth of G. Thus, in order to save time and space in practice, we avoid handling some of such S. We compute collections $TW_1, TW_2, \ldots, TW_n$. Each collection TW_i $(1 \leq i \leq n)$ contains pairs $(S, TW(S))$ with $|S| = i$. The collection for sets of size $i > 1$ is built as follows: for each pair $(S, r) \in TW_{i-1}$ and each $x \in V - S$, we compute $r' = \max(r, Q(S, x))$. If $r' < up$, then we check if there is a pair $(S \cup \{x\}, t)$ in TW_i for some t, and if so, replace it by $(S \cup \{x\}, \min(t, r'))$. If no such pair exists in TW_i, we insert $(S \cup \{x\}, r')$ in TW_i.

In our implementation, we use two additional optimizations that appeared to give significant savings in time and memory consumption. The following simple lemma gives the first idea.

Lemma 3. *Let $G = (V, E)$ be a graph, and let $S \subseteq V$. The treewidth of G is at most* $\max\{TW(S), n - |S| - 1\}$.

Lemma 3 shows correctness of the following rule that was used in the implementation: we keep an upper bound up for the treewidth of G, initially set by the user or set to $n - 1$. Each time, we get a pair (S, r) in a collection TW_i, either by insertion, or by replacement of an existing pair, we set the upper bound up to the minimum of up and $n - |S| - 1 = n - i - 1$. Moreover, when handling a pair (S, r) from TW_{i-1}, it is first checked if r is smaller than up; if not, then this pair cannot contribute to an improvement of the upper bound, and hence is skipped. Our second optimization is stated in Lemma 4.

Lemma 4. *Let $G = (V, E)$ be a graph of treewidth k. Given a subset $Q \subset V$ inducing a clique in G, there exists a linear ordering π with $R_\pi(v) \leq k$ and $\pi(v) \geq |V| - |Q| + 1$ for all $v \in Q$.*

By Lemma 4, we can restrict the sets S to elements from $V \setminus Q$ for some clique Q; in particular for the maximum clique. Although it is NP-hard to compute the maximum clique in a graph, it can be computed extremely fast for the graphs considered. In our program, we use a simple combinatorial branch-and-bound is used to compute all maximum cliques. It recursively extends a clique by all candidate vertices once.

The algorithm was implemented in C++, using the Boost graph library, as part of the Treewidth Optimization Library TOL, a package of algorithms for the treewidth of graphs. The package includes preprocessing, upper bound, and lower bound algorithms for treewidth. Experiments were carried out on a number of graphs taken from applications; several were used in other experiments. See [19] for the used graphs, information on the graphs, and other results of experiments

to compute the treewidth. The experiments were carried out on a Sun computer with 4 AMD Dualcore Opteron 875, 2.2 GHz processor and at most 20 GB of internal memory available. The program did not use parallelism.

In Table 1 the results of our experiments on a number of graphs are reported. Besides instance name, number of vertices, number of edges, and the computed treewidth, we report on the CPU time in seconds and the maximum number of sets (S, r), considered at once, $\max |TW| = \max_{i=0,\ldots,n} |TW_i|$ in a number of cases. First, we report on the CPU time and maximum number of sets for the case that no initial upper bound *up* is exploited. Next, we report on the case where we use an initial upper bound, displayed in the column *up*. The last two columns report on the experiments in which the algorithm is advanced by both an initial upper bound *up* and a maximum clique Q.

In several instances reported in [19], the best bound obtained from a few upper bound heuristics, and the lower bound obtained by the LBP+(MMD+) heuristic match, and then we have obtained in a relatively fast way an exact bound on the treewidth of the instance graph. In other cases, these bounds do not match. Then, when the graph is not too large, the dynamic programming algorithm can be of good use.

A nice example is the celar03 graph. This graph has 200 vertices and 721 edges. A combination of different preprocessing techniques yield an equivalent instance celar03-pp-001 which has 38 vertices and 238 edges. Existing upper bound heuristics gave a best upper bound of 15, while the lower bound of the LBP+(MMD+) heuristic was 13. With the dynamic programming algorithm with 15 as input for an upper bound, we obtained the exact treewidth of 14 for this graph, and hence also for celar03.

The algorithm can also be used as a lower bound heuristic: give the algorithm as 'upper bound' a conjectured lower bound ℓ: when it terminates, it either has found the exact treewidth, or we know that ℓ is indeed a lower bound for the treewidth of the input graph. In a few cases, we could thus increase the lower bound for the treewidth of considered instances, e.g., for the treewidth of the queen8-8 graph (the graph modeling the possible moves of a queen on an 8 by 8 chessboard) the lower bound could be improved from 27 to 35.

For larger graphs, the above idea can be combined by an idea exploited earlier in various papers. Given a graph G and a minor G' of G, $\mathbf{tw}(G') \leq \mathbf{tw}(G)$. In [5,12,16], a lower bound on $\mathbf{tw}(G')$ is computed to obtain a lower bound for G. With the dynamic programming algorithm, we can compute $\mathbf{tw}(G')$ exactly to obtain a lower bound for $\mathbf{tw}(G)$. For the 1024 vertices graph pignet2-pp, we have generated a sequence of minors by repeatedly contracting a minimum degree vertex with a neighbor with least number of common neighbors (see [5]). Figure 1 shows the treewidth (right y-scale) for the minors with 70 to 79 vertices. Moreover, the maximum number of sets for three different upper bounds is reported (left y-scale, logarithmic). If the used upper bound is less than or equal to the treewidth, no feasible solution is found in the end. The best known lower bound for pignet2-pp is increased from 48 to 59 by the treewidth of the

Table 1. Experimental results for some DIMACS vertex coloring graphs, some probabilistic networks and celar03-pp-001

				no up, no Q			with up, no Q			with up, w Q	
instance	$\|V\|$	$\|E\|$	**tw**	CPU	max $\|TW\|$	up	CPU	max $\|TW\|$		CPU	max $\|TW\|$
myciel3	11	20	5	0.00	240	5	0.00	35		0.00	21
myciel4	23	71	10	7.64	296835	10	0.14	4422		0.12	4064
queen5-5	25	160	18	0.15	18220	18	0.02	944		0.02	392
queen6-6	36	290	25	36.43	2031716	26	1.16	18872		0.36	6994
queen7-7	49	476	35	-	-	37	1012.12	96517095		248.03	24410915
pathfinder-pp	12	43	6	0.00	107	6	0.00	1		0.00	1
oesoca+-pp	14	75	11	0.00	48	11	0.00	5		0.00	5
fungiuk	15	36	4	0.07	4713	4	0.00	4		0.00	4
weeduk	15	49	7	0.02	2906	7	0.00	35		0.00	35
munin-kgo-pp	16	41	5	0.11	6892	5	0.00	2		0.00	2
wilson	21	27	3	14.44	350573	3	0.08	2412		0.06	2342
water-pp	22	96	9	1.60	77286	10	0.04	816		0.01	475
oow-trad-pp	23	54	6	42.91	1065120	6	0.09	2953		0.05	1895
barley-pp	26	78	7	349.31	6110572	7	0.61	13597		0.29	7971
oow-bas	27	54	4	1579.38	19937301	4	0.01	303		0.00	111
oow-solo-pp	27	63	6	1059.50	17048070	6	0.91	22484		0.30	9426
ship-ship-pp	30	77	8	-	-	9	291.20	3062863		50.75	820910
water	32	123	9	-	-	10	12.59	127545		1.53	25874
oow-trad	33	72	6	-	-	6	129.55	1162650		14.55	178846
mildew	35	80	4	-	-	4	2.98	33045		0.35	5431
mainuk	48	198	7	-	-	8	-	-		2251.97	11748147
celar03-pp-001	38	238	14	-	-	15	121.29	911918		4.36	55504

79 vertex-minor. Figure 1 shows one more time the impact of the upper bound on the memory consumption (and time consumption) of the algorithm.

4 Improved Polynomial Space Algorithms for Treewidth

In this section, we give a faster exponential time algorithm with polynomial space for Treewidth. The algorithm is based on results of earlier sections combined with techniques based upon balanced separators.

Lemma 5. *Let $S \subseteq V$ be a set of vertices, such that the treewidth of G is equal to the treewidth of the graph $G' = (V, E \cup \{\{v, w\} \mid v, w \in S, v \neq w\})$ obtained from G by turning S into a clique. Then there is a linear ordering $\pi \in \Pi(V)$, that ends with a permutation of S, with $\mathbf{tw}(G) = \max_{v \in V} R_\pi(v)$, Moreover, $\mathbf{tw}(G) = \max\{TW(V - S), TWR(V - S, S)\}$.*

Lemma 6. *Let $G = (V, E)$ be a graph with treewidth at most k. There is a set $S \subseteq V$ with $|S| = k+1$, such that each connected component of $G[V-S]$ contains at most $(|V|-k)/2$ vertices and the graph $G' = (V, E \cup \{\{v, w\} \mid v, w \in S, v \neq w\})$ obtained from G by turning S into a clique has treewidth at most k.*

Lemma 6 is a variant on a folklore result on balanced separators in graphs of small treewidth. Lemmas 5 and 6 are used to show Lemma 7.

Lemma 7. *Let $G = (V, E)$ be a graph with treewidth at most k. Let $k + 1 \leq r \leq n$. There is a set $W \subseteq V$, with $|W| = r$, such that each connected component of $G[V - W]$ contains at most $(|V| - r + 1)/2$ vertices, and $\mathbf{tw}(G) = \max\{TWR(\emptyset, V - W), TWR(V - W, W)\}$.*

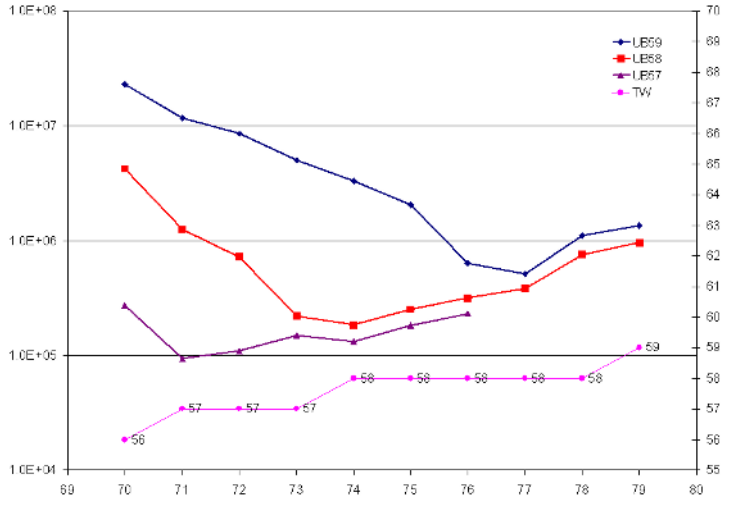


Fig. 1. Maximum number of subsets S during algorithm for different upper bounds

For a graph $G = (V, E)$, and a set W, let $G^+[W]$ be the fill-in graph, obtained by eliminating the vertices in $V - W$, i.e., $G^+[W] = (W, F)$, with for all $v, w \in W$, $v \neq w$, we have that $\{v, w\} \in F$, if and only if there is a path from v to w that uses only vertices in $V - W$ as internal vertices. The next lemma formalizes the intuition behind $TWR(V - W, W)$: when computing $TWR(V - W, W)$, we look for the best ordering of the vertices in W, after all vertices in $V - W$ are eliminated — i.e., in the graph $G^+[W]$.

Lemma 8. *Let $G = (V, E)$ be a graph, and $W \subseteq V$ a set of vertices. Then* $\mathbf{tw}(G^+[W]) = TWR(V - W, W)$.

Lemma 9. *Let $G = (V, E)$ be a graph, and let $S = S_1 \cup S_2 \subseteq V$. Suppose $S_1 \cap S_2 = \emptyset$, and that there is no edge between a vertex in S_1 and a vertex in S_2. Then $TW(S) = \max\{TW(S_1), TW(S_2)\}$.*

The lemmas above are summarized in the following result, which gives a main idea of the improved recursive algorithm.

Corollary 1. *Let $G = (V, E)$ be a graph, and let k, r be integers, $0 \leq k < r \leq |V|$. The treewidth of G is at most k, if and only if there is a set of vertices $S \subseteq V$, with $|S| = r$, such that each connected component of $G[V - S]$ contains at most $(|V| - r + 1)/2$ vertices, for each connected component W of $G[V - S]$, $TWR(\emptyset, W) \leq k$, and the treewidth of $G^+[S]$ is at most k.*

We now present the main result of this section.

Theorem 3. *The treewidth of a graph G on n vertices can be computed in polynomial space and time $O^*(2.9512^n)$.*

Proof. We describe a decision algorithm for treewidth: given a graph G, and an integer k, it decides whether the treewidth of G is at most k. Of course, an algorithm that, given a graph G, computes $\mathbf{tw}(G)$ can be constructed at the cost of an additional multiplicative factor $O(\log n)$. Correctness of the algorithm follows from Corollary 1. Let $\gamma = 0.4203$.

The algorithm works as follows. If $|V| \leq k+1$, then the treewidth of G is at most $|V|-1 \leq k$, so the algorithm returns true.

Otherwise, the algorithm checks if $k \leq 0.25 \cdot |V|$ or $k \geq \gamma \cdot |V|$. If this is the case, then we search for a set S, as implied by Corollary 1 when we take $r = k+1$. I.e., we enumerate all sets S of size $k+1$. For each such S, we check if all connected components of $G = (V, E)$ have size at most $(|V| - |S| + 1)/2$. If so, we use the algorithm of Theorem 2 (Algorithm 1 Recursive-Treewidth) to compute for each connected component W the value $TWR(W, \emptyset)$. If for each such component W, $TWR(\emptyset, W)$, then the algorithm returns true: as $G^+(W)$ has $k+1$ vertices, its treewidth is trivially at most k, and hence all conditions of Corollary 1 are fulfilled, so G has treewidth at most k. If no set S of size $k+1$ yields true, then the algorithm returns false.

The remaining case is that $0.25 \cdot |V| < k < \gamma \cdot |V|$. Now, we search for a set S as implied by Corollary 1 when taking $r = \lceil \gamma \cdot |V| \rceil$. I.e., we enumerate all sets S of size $r = \lceil \gamma \cdot |V| \rceil$. For each we check if all connected components W of $G[V-S]$ have size at most $(|V| - r + 1)/2$. If so, we use Algorithm 1 Recursive-Treewidth for deciding if all connected components W of $G[V-S]$ fulfill $TWR(\emptyset, W) \leq k$. We also recursively call the algorithm on $G^+(W)$ to decide if this graph has treewidth at most k. If all these checks succeed, the algorithm returns true. If no S of size r made the algorithm return true, the algorithm returns false.

We now analyze the running time of the algorithm. Write $\alpha = k/|V|$.

We start with analyzing the case where $k \leq 0.25 \cdot |V|$ or $\gamma \cdot |V| \leq k$. We have $\alpha \leq 0.25$ or $\alpha \geq \gamma$. The number of subsets of size $\alpha \cdot n$ of a set of size n is known to be of size $O^*((\alpha^{-\alpha} \cdot (1-\alpha)^{\alpha-1})^n)$. Write $f(\alpha) = \alpha^{-\alpha} \cdot (1-\alpha)^{\alpha-1} \cdot 2^{1-\alpha}$. Each connected component W of $G[V-S]$ for which the algorithm calls Recursive-Treewidth has size at most $(|V| - \alpha \cdot |V| + 1)/2$, thus we use at most $O^*(4^{(|V|-\alpha\cdot|V|+1)/2}) = O^*(2^{(1-\alpha)|V|}$ time for one such component. Thus, the total time in this case is bounded by $O^*(f(\alpha)^n))$. f monotonically increases in the interval $(0, \frac{1}{3})$, and monotonically decreases in the interval $(\frac{1}{3}, 1)$. As $f(0.25) < 2.9512$, and $f(\gamma) < 2.9512$, we have for all α with $0 < \alpha \leq 0.25$ or $\gamma \leq \alpha < 1$, that $f(\alpha) < 2.9512$, and hence that the algorithm uses $O^*(2.9512^n)$ time.

We now look at the case where $0.25 \cdot |V| < k < \gamma \cdot |V|$, i.e., where $0.25 < \alpha < \gamma$. As in the previous case, the time for all computations of $TWR(\emptyset, W)$ for all connected components of $G[V-S]$ over all sets $S \subseteq V$ of size r is bounded by $O^*((\gamma^{-\gamma} \cdot (1-\gamma)^{\gamma-1} \cdot 2^{1-\gamma})^n = O^*(f(\gamma)^n) = O^*(f(0.4203)^n) < O^*(2.9512^n)$.

We have to add to this time the total time over all recursive calls to the algorithm with graphs of the form $G^+(S)$. Note that the recursion depth is at most 1: in the recursion, the value of k is unchanged, while we now have a graph with $|S| = \lceil \gamma \cdot |V| \rceil$ vertices. So, in the recursive call, $k > 0.25 \cdot |V| > \gamma \cdot |S|$, and the algorithm executes the first case. From the analysis above, it follows

that each recursive call of Improved-Recursive-Treewidth on a graph $G^+[S]$ costs $O^*(\left(\beta^{-\beta} \cdot (1-\beta)^{\beta-1} \cdot 2^{1-\beta}\right)^{\gamma \cdot n})$ time, with $\beta = \alpha/\gamma$. Write

$$g(\alpha) = \gamma^\gamma \cdot (1-\gamma)^{\gamma-1} \cdot \left((\alpha/\gamma)^{-\alpha/\gamma} \cdot (1-\alpha/\gamma)^{\alpha/\gamma-1} \cdot 2^{1-\alpha/\gamma}\right)^\gamma$$

As there are $O^*((\gamma^\gamma \cdot (1-\gamma)^{\gamma-1})^n)$ vertex sets $S \subseteq V$ of size γn, the total time of all calls of Improved-Recursive-Treewidth with graphs of the form $G^+[S]$ is bounded by $O^*(g(\alpha)^n)$. On the interval $[0.25, \gamma]$, the function g is monotonically decreasing, with $g(0.25) < 2.9511$. Thus, the total time over all calls of Improved-Recursive-Treewidth for graphs $G^+[S]$ is bounded by $O^*(2.9511^n)$, and the total time of the algorithm is bounded by $O^*(2.9512^n)$. □

We conjecture that with a more detailed analysis and more levels of recursion, small improvements to the running time are possible.

5 Conclusions

In this paper, we have given dynamic programming and recursive algorithms to compute the treewidth. Similar results can be obtained for related graph parameters, like minimum fill-in. The dynamic programming algorithm for the treewidth problem has been implemented; for small instances (slightly below 50 vertices), the algorithm appears to be practical. On a more theoretical side, we gave the first exponential time algorithms for Treewidth with a running time of the type $O^*(c^n)$ for some constant c that use polynomial space and we reduced the running time of the algorithm with polynomial space to $O^*(2.9512.)$.

A comparison of the dynamic programming algorithm for Treewidth with other algorithms, (e.g., a branch and bound algorithm as in [12] or the algorithm of Shoikhet and Geiger [18]) would be very interesting.

References

1. 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.
2. E. H. Bachoore and H. L. Bodlaender. New upper bound heuristics for treewidth. In S. E. Nikoletseas, editor, *Proceedings of the 4th International Workshop on Experimental and Efficient Algorithms WEA 2005*, pages 217–227. Springer-Verlag, Lecture Notes in Computer Science, vol. 3503, 2005.
3. H. L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theor. Comp. Sc.*, 209:1–45, 1998.
4. H. L. Bodlaender. Discovering treewidth. In P. Vojtáš, M. Bieliková, and B. Charron-Bost, editors, *SOFSEM 2005: Theory and Practive of Computer Science: 31st Conference on Current Trends in Theory and Practive of Computer Science*, pages 1–16. Springer-Verlag, Lecture Notes in Computer Science 3381, 2005.
5. 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.

6. V. Bouchitté and I. Todinca. Treewidth and minimum fill-in: Grouping the minimal separators. *SIAM J. Comput.*, 31:212–232, 2001.
7. V. Bouchitté and I. Todinca. Listing all potential maximal cliques of a graph. *Theor. Comp. Sc.*, 276:17–32, 2002.
8. F. Clautiaux, A. Moukrim, S. Négre, and J. Carlier. Heuristic and meta-heuristic methods for computing graph treewidth. *RAIRO Operations Research*, 38:13–26, 2004.
9. N. D. Dendris, L. M. Kirousis, and D. M. Thilikos. Fugitive-search games on graphs and related parameters. *Theor. Comp. Sc.*, 172:233–254, 1997.
10. F. V. Fomin, F. Grandoni, and D. Kratsch. Some new techniques in design and analysis of exact (exponential) algorithms. *Bulletin of the EATCS*, 87:47–77, 2005.
11. 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, ICALP 2004*, pages 568–580, 2004.
12. 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.
13. M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York, 1980.
14. Y. Gurevich and S. Shelah. Expected computation time for Hamiltonian path problem. *SIAM J. Comput.*, 16:486–502, 1987.
15. M. Held and R. Karp. A dynamic programming approach to sequencing problems. *J. SIAM*, 10:196–210, 1962.
16. A. M. C. A. Koster, T. Wolle, and H. L. Bodlaender. Degree-based treewidth lower bounds. In S. E. Nikoletseas, 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.
17. D. J. Rose, R. E. Tarjan, and G. S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM J. Comput.*, 5:266–283, 1976.
18. K. Shoikhet and D. Geiger. A practical algorithm for finding optimal triangulations. In *Proc. National Conference on Artificial Intelligence (AAAI '97)*, pages 185–190. Morgan Kaufmann, 1997.
19. Treewidthlib. http://www.cs.uu.nl/people/hansb/treewidthlib, 2004.
20. Y. Villanger. Improved exponential-time algorithms for treewidth and minimum fill-in. In *Proceedings of the 7th Latin American Theoretical Informatics Symposium (LATIN 2006)*, volume 3887 of *LNCS*, pages 800–811. Springer-Verlag, Berlin, 2006.
21. G. J. Woeginger. Exact algorithms for NP-hard problems: A survey. In *Combinatorial Optimization: "Eureka, you shrink"*, pages 185–207, Berlin, 2003. Springer Lecture Notes in Computer Science, vol. 2570.

An Improved Construction for Counting Bloom Filters

Flavio Bonomi[1], Michael Mitzenmacher[2,*], Rina Panigrahy[3,**], Sushil Singh[1], and George Varghese[1,***]

[1] Cisco Systems Inc.
{bonomi, sushilks, gevarghe}@cisco.com
[2] Harvard University
michaelm@eecs.harvard.edu
[3] Stanford University
rinap@cs.stanford.edu

Abstract. A counting Bloom filter (CBF) generalizes a Bloom filter data structure so as to allow membership queries on a set that can be changing dynamically via insertions and deletions. As with a Bloom filter, a CBF obtains space savings by allowing false positives. We provide a simple hashing-based alternative based on d-left hashing called a d-left CBF (dlCBF). The dlCBF offers the same functionality as a CBF, but uses less space, generally saving a factor of two or more. We describe the construction of dlCBFs, provide an analysis, and demonstrate their effectiveness experimentally.

1 Introduction

A Bloom filter is an inexact representation of a set that allows for false positives when queried; that is, it can sometimes say that an element is in the set when it is not. In return, a Bloom filter offers very compact storage: less than 10 bits per element are required for a 1% false positive probability, independent of the size or number of elements in the set. There has recently been a surge in the popularity of Bloom filters and variants, especially in networking [6]. One variant, a counting Bloom filter [10], allows the set to change dynamically via insertions and deletions of elements. Counting Bloom filters have been explicitly used in several papers, including for example [7, 8, 9, 10, 12, 18, 19].

In this paper, we present a new construction with the same functionality as the counting Bloom filter, based on d-left hashing. We call the resulting structure a d-left counting Bloom filter, or dlCBF. For the same fraction of false positives, the dlCBF generally offers a factor of two or more savings in space over the standard solution, depending on the parameters. Moreover, the construction is very simple and practical, much like the original Bloom filter construction. As counting Bloom filters are often used in settings where space and computation

* Supported in part by NSF grant CCR-0121154 and a research grant from Cisco.
** Part of this work was done while working at Cisco Systems, Inc.
*** Now back at U.C. San Diego.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 684–695, 2006.

are limited, including for example routers, we expect that this construction will prove quite useful in practice.

2 Background

2.1 Bloom Filters and Counting Bloom Filters

We briefly review Bloom filters; for further details, see [6]. A Bloom filter represents a set S of m elements from a universe U using an array of n bits, denoted by $B[1], \ldots, B[n]$, initially all set to 0. The filter uses a group H of k independent hash functions $h_1, \ldots, h_k$ with range $\{1, \ldots, n\}$ that independently map each element in the universe to a random number uniformly over the range. (This optimistic assumption is standard and convenient for Bloom filter analyses.) For each element $x \in S$, the bits $B[h_i(x)]$ are set to 1 for $1 \leq i \leq k$. (A bit can be set to 1 multiple times.) To answer a query of the form "Is $y \in S$?", we check whether all $h_i(y)$ are set to 1. If not, y is not a member of S, by the construction. If all $h_i(y)$ are set to 1, it is assumed that y is in S, and hence a Bloom filter may yield a *false positive*.

The probability of a false positive for an element not in the set is easily derived. If p is the fraction of ones in the filter, it is simply p^k. A standard combinatorial argument gives that p is concentrated around its expectation

$$\left(1 - (1 - 1/n)^{mk}\right) \approx \left(1 - \mathrm{e}^{-km/n}\right).$$

These expressions are minimized when $k = \ln 2 \cdot (n/m)$, giving a false positive probability f of $f \approx (1/2)^k \approx (0.6185)^{n/m}$. In practice, k must be an integer, and both n/m (the number of bits per set element) and k should be thought of as constants. For example, when $n/m = 10$ and $k = 7$ the false positive probability is just over 0.008.

Deleting elements from a Bloom filter cannot be done simply by changing ones back to zeros, as a single bit may correspond to multiple elements. To allow for deletions, a *counting Bloom filter* (CBF) uses an array of n counters instead of bits; the counters track the number of elements currently hashed to that location [10]. Deletions can now be safely done by decrementing the relevant counters. A standard Bloom filter can be derived from a counting Bloom filter by setting all non-zero counts to 1. Counters must be chosen large enough to avoid overflow; for most applications, four bits suffice [5, 10]. We generally use the rule of four bits per counter when comparing results of our data structure with a standard CBF, although we do note that this could be reduced somewhat with some additional complexity.

2.2 Related Work on Counting Bloom Filters

The obvious disadvantage of counting Bloom filters is that they appear quite wasteful of space. Using counters of four bits blows up the required space by a factor of four over a standard Bloom filter, even though most entries are zero.

Some work has been done to improve on this. The spectral Bloom filter was desigend for multi-sets, but also considers schemes to improve the efficiency of storing counters [7]. A paper on "optimal" Bloom filter replacements is another work in this vein [17], introducing a data structure with the same functionality as a counting Bloom filter that is, at least asymptotically, more space-efficient.

While this problem has received some attention, the previous work does not appear to give useful solutions. Spectral Bloom filters are primarily designed for multi-sets and skewed streams; we do not know of experiments or other evidence suggesting they are appropriate as a replacement for a CBF. The alternatives suggested in [17] do not appear to have been subject to experimental evaluation. Moreover, the schemes suggested in both of these papers appear substantially more complex than the standard counting Bloom filter scheme. This simplicity is not just useful in terms of ease of programming; for implementations in hardware, the simplicity of the Bloom filter scheme translates into very straightforward and clean hardware designs.

Our goal is to provide a scheme that maintains the simplicity of the original counting Bloom filter construction, and further is backed by experimental results demonstrating that the scheme is likely to be very useful in practice. We believe our work is novel in these regards. Our motivation for our general approach came about when considering generalizations of Bloom filters for state machines. See [4] for more details.

2.3 Background: *d*-Left Hashing

Our approach makes use of d-left hashing, a variation of the balanced allocations paradigm [1] due to Vöcking [20], which we now recall. Often this setting is described in terms of balls and bins; here, for consistency, we use the terms elements and buckets. We have a hash table consisting of n buckets. Initially they are divided into d disjoint subtables of n/d buckets. (For convenience we assume n/d is an integer.) We think of the subtables as running consecutively from left to right. Each incoming element is hashed to give it a collection of d possible buckets where it can be placed, one in each subtable. We assume in the analysis that these choices are uniform and independent. Each incoming element is placed in the bucket containing the smallest number of elements; in case of a tie, the element is placed in the bucket of the leftmost subtable with the smallest number of elements. To search for an element in the hash table, the contents of d buckets must be checked. Note that, if the bucket size is fixed a priori, there is the possibility of overflow. Various combinatorial bounds on the resulting maximum load have been proven [2, 20].

For our purposes, we are more interested in obtaining precise estimates of d-left hashing under constant average load per bucket. For the case where elements are only inserted, this can be obtained by considering the fluid limit, corresponding to the limiting case where the number of elements and buckets grow to infinity but with the ratio between them remaining fixed. The fluid limit is easily represented by a family of differential equations, as described in [5, 14]. The advantage of using the fluid limits in conjunction with simulation is that

it provides insight into how d-left hashing scales and the probability of overflow when fixed bucket sizes are used. Because of lack of space, we do not review the derivation of the differential equations. (See the full version for more details.)

Analyzing the behavior with deletions is somewhat more problematic in this framework, as one requires a suitable model of deletions. Good insight can be gained by the following approach. Suppose that we begin by inserting m elements, and then repeatedly, at each time step, delete an element chosen uniformly at random and then insert a new element. The corresponding fluid limit equations can be easily derived and are very similar to the insertion-only case. We can run the family of equations until the system appears to reach a steady state distribution. (Again, more details are in the full version.)

3 The d-Left CBF Construction

3.1 The Framework

Our goal is to design a structure that allows membership queries on a set S over a universe U that can change dynamically via insertions and deletions, although there will be an upper bound of m on the size of the set. A query on $x \in S$ should always return that $x \in S$; a query on some $y \notin S$ could give a false positive. The target false positive rate is ϵ. We allow for data structures that may with very small probability reach a failure condition, such as the overflow of a counter, at some point in its lifetime. Preferably, the failure probability is so small that it should not occur in practice. The failure condition should, however, be apparent, so that an appropriate response can be taken if necessary.

A standard counting Bloom filter offers one possible solution to this problem. Our alternative has a different starting point. It is a folklore result (see [6]) that if the set S is static, one can achieve essentially optimal performance by using a perfect hash function and fingerprints. One finds a perfect hash function $P : U \to [|S|]$, and then stores at each location an $f = \lceil \log 1/\epsilon \rceil$ bit fingerprint in an array of size $|S|$, computed according to some (pseudo-)random hash function H. A query on z requires computing $P(z)$ and $H(z)$, and checking whether the fingerprint stored at $P(z)$ matches $H(z)$. When $z \in S$ a correct response is given, and when $z \notin S$ a false positive occurs with probability at most ϵ; this uses $m\lceil \log 1/\epsilon \rceil$ bits.

The problem with this approach is that it does not cope with changes in the set S, and perfect hash functions are generally too expensive to compute for most applications. To deal with this, we make use of the fact, recognized in [5], that using d-left hashing provides a natural way to obtain an "almost perfect" hash function. The resulting hash function is only almost perfect in that instead of having one set element in each bucket, there can be several, and space is not perfectly utilized. A strong advantage, however, is that it can easily handle dynamically changing sets. The resulting construction meets our goals of being a substantial improvement over Bloom filters while maintaining simplicity. (See [12] for an alternative approach for dynamic "almost perfect" hash functions.)

3.2 The Construction of a *d*-Left Counting Bloom Filter

We first present a seemingly natural construction of a dlCBF that has a subtle flaw; we then demonstrate how this flaw can be corrected. To begin, we use a d-left hash table, where each bucket consists of many cells, each cell being a fixed number of bits meant to hold a fingerprint and a counter. As we want to avoid pointers to keep our representation as small as possible, we use a fixed number of cells per bucket, so that our hash table may be viewed as a large bit array.

We store a fingerprint for each element. The fingerprints are essentially compressed by taking advantage of how they are stored. Specifically, each fingerprint will consist of two parts. The first part corresponds to the *bucket index* the element is placed in. We assume the bucket index has range $[B]$, where in this setting we use $[x] = \{0, 1, \ldots, x-1\}$. The second part is the remaining fingerprint, which we refer to as the *remainder*, and is stored explicitly. We assume the remainder has range $[R]$.

For example, if we were just using a single hash function, and a single hash table with B buckets, we would use a hash function $H : U \to [B] \times [R]$. The m elements of S would be stored by computing $H(x)$ for each $x \in S$ and storing the appropriate remainders in a cell for each bucket. In order to handle deletions in the case that two (or more) elements might yield the same bucket and remainder, each cell would also contain a small counter. A false positive would occur if and only if for a query $y \notin S$ there existed $x \in S$ with $H(x) = H(y)$.

Using a single hash function yields the problem that the distribution of the load varies dramatically across buckets, essentially according to a Poisson distribution. Since we use a fixed number of cells per bucket, to avoid overflow requires a small average load as compared to the maximum load, leading to a lot of wasted spaced. Using d-left hashing dramatically reduces this waste.

We now explain the subtle problem with using d-left hashing directly. Let us suppose that our hash table is split into d subtables, each with B buckets. To use d-left hashing, we would naturally use a hash function $H : U \to [B]^d \times [R]$, giving d choices for each element, and store the remainder in the least loaded of the d choices (breaking ties to the left). The problem arises when it comes time to delete an element from the set. The corresponding remainder might be found in more than one of the d choices, as the same remainder might have been placed by another element in another of these d buckets at some later point in time. When this happens, we do not know which copy to delete.

It is worth making this clear by framing a specific example. Suppose that when an element x is inserted into the table, its d choices correspond to the first bucket in each subarray, and its remainder is a. Suppose further that the loads are such that the remainder is stored in the last subarray. Now suppose later than an element y is inserted into the table, its d choices correspond to the ith bucket in the ith subarray for each i, and its remainder is also a. Notice that, because the remainder a was placed in the first bucket of the *last* subarray for x, when y is placed, this remainder a will not be seen in any of y's buckets. Now suppose that, due to y, the remainder a is placed in the first bucket of the first subarray. Finally, consider what happens when we now try to delete x. The

appropriate remainder a now appears in two of x's buckets, the first and the last, and there is no way to tell which to delete. Deleting both would lead to false negatives for queries on the element y; such occurrences happen too frequently to allow this approach. Failing to delete would leave garbage in the table, causing it to fill and leading to increased false positives.

We solve this problem by breaking the hashing operations into two phases. For the first phase, we start with a hash function $H : U \to [B] \times [R]$; this gives us the true fingerprint $f_x = H(x)$ for an element. For the second phase, to obtain the d locations, we make use of additional (pseudo)-random permutations $P_1, \ldots, P_d$. Specifically, let $H(x) = f_x = (b, r)$. Then let

$$P_1(f_x) = (b_1, r_1), P_2(f_x) = (b_2, r_2), \ldots, P_d(f_x) = (b_d, r_d).$$

The values $P_i(f_x)$ correspond to the bucket and remainder corresponding to f_x for the ith subarray. Notice that for a given element, the remainder that can be stored in each subarray can be different; although this is not strictly necessary, it proves convenient for implementation. When storing an element, we first see whether in any bucket b_i the remainder r_i is already being stored. If so, we simply increment the corresponding counter. We point out that these counters can be much smaller than counters used in the standard CBF construction, as here collisions are much rarer; they occur only when H gives the same result for multiple elements. Also, as we show in Claim 3.2, only one remainder associated with f_x is stored in the table at any time, avoiding any problem with deletions. If r_i is not already stored, we store the remainder in the least loaded bucket according to the d-left scheme.

The following simple claims demonstrate the functionality of this dlCBF construction. When considering false positives below, we ignore the negligible probabilities of counter or bucket overflow, which must be considered separately.

Claim. When deleting an element in the set, only one remainder corresponding to the element will exist in the table.

Proof. Suppose not. Then there is some element $x \in S$ whose remainder is stored in subtable j to be deleted and at the same time another element $y \in S$ such that $P_i(f_x) = P_i(f_y)$ for $i \neq j$. Since the P_i are permutations, we must have that $f_x = f_y$, so x and y share the same true fingerprint. Now suppose without loss of generality that x was inserted before y; in this case, when y is inserted, the counter in subtable j associated with the remainder of x would be incremented, contradicting our assumption.

Claim. A false positive for a query z occurs if and only if $H(z) = H(x)$ for some $x \in S$.

Proof. If z gives a false positive, we have $P_i(f_x) = P_i(f_z)$ for some $x \in S$. But then $H(x) = H(z)$.

Claim. The false positive probability is $1 - (1 - 1/BR)^{|S|} \approx m/BR$.

Proof. The probability that there is no false positive for z is the probability that no $x \in S$ has $H(x) = H(z)$, and this expression corresponds to that probability.

We have thereby avoided the problem of finding two possible fingerprints to delete when handling deletions. In return, however, we have introduced another issue. Our process is no longer exactly equivalent to the d-left hashing process we have analyzed, since our d choices are no longer independent and uniform, but instead determined by the choice of the permutations. In any instantiation, there are really only BR collections of d choices available, not a full B^d. Fortunately this problem is much less significant, at least in practice. Intuitively, this is because the dependence is small enough that the behavior is essentially the same. We verify this with simulations below. A formal argument seems possible, for limited numbers of deletions, but is beyond the scope of this paper. More discussion of this point is given in the full version.

3.3 Additional Practical Issues

In practice we recommend using simple linear functions for the permutations; for example, when $H(x)$ can has range $[2^q]$, we suggest using

$$P_i(H(x)) = aH(x) \bmod 2^q$$

for a chosen uniformly at random from the odd numbers in $[2^q]$. The high order bits of $P_i(H(x))$ can be used for the bucket, and the low order bits for the fingerprint. (Because of the dependence of these hash functions, using the low order bits for the buckets is less effective; the same groups of buckets will frequently be chosen. Also, although even $H(x)$ values are then placed only in even buckets, and similarly for odd $H(x)$ values, since $H(x)$ values are (pseudo)-random the effect is negligible.) In this case, it is harder to see why the system behavior should necessarily follow the fluid limit, since the dependence among the bucket choices is quite strong with such limited hash functions. However, our simulations, discussed below, suggest the fluid limit is still remarkably accurate. (Some theoretical backing for this comes from the the recent results of [11]; again, further discussion is in the full version.)

Using simple invertible permutations P_i may allow further advantages. For example, when inserting an element, it may be possible to move other elements in the hash table, as long as each element is properly placed according to one of its d choices. (Allowing such movement of elements was the insight behind cuckoo hashing [15], and subsequent work based on cuckoo hashing, including [16].) Intuitively, such moves allow one to rectify previous placement decisions that may have subsequently turned out poorly. Recent work has shown that even limited ability to move existing elements in the hash table can yield better balance and further reduce loads [15, 16]. In particular, such moves may be useful as an emergency measure for coping with situations where the table becomes overloaded, using the moves to prevent overflow. By using simple invertible permutations P_i, one can compute f_x from a value $P_i(f_x)$, and move the fingerprint to another location given by $P_j(f_x)$. We have not studied this approach extensively, as we believe the costs outweigh the benefits for our target applications, but it may be useful in future work.

4 A Comparison with Standard Counting Bloom Filters

Roughly speaking, our experience has been that for natural parameters, our dlCBF uses half the space or less than standard CBF with the same false positive probability, and it appears as simple or even simpler to put into practice. We now formalize this comparison. Suppose, for convenience, that we are dynamically tracking a set of m elements that changes over time.

For m elements, a standard CBF using cm counters, each with four bits, as well as the theoretically optimal $k = c \ln 2$ hash functions, gives a false positive probability of approximately $(2^{-\ln 2})^c$ using $4cm$ bits. (This is slightly optimistic, because of the rounding for k.) The probability of counter overflow is negligible.

A comparable system using our dlCBF would use four subarrays, each with $m/24$ buckets, giving an average load of six elements per bucket. The method of Section 2.3 shows that providing room for eight elements per bucket should suffice to prevent bucket overflow with very high probability. Each cell counter should allow for up to four elements with the same hash value from the hash function H in the first step to prevent counter overflow with high probability. This can be done effectively with 2 bit counters, as long as one has a sentinel cell value, say 0, that cannot be a fingerprint but represents an empty cell. (We ignore the minimal effect of the sentinel value henceforth.) With an r bit remainder, the false positive probability is upper bounded by $24 \cdot 2^{-r}$, and the total number of bits used is $4m(r + 2)/3$. (For convenience, we think of $r \geq 5$ henceforth, so that our upper bound on the probability is less than 1.) Alternatively, one can think in the following terms: to obtain a false positive rate of $f = 24 \cdot 2^{-r}$, one needs to use $(4 \log_2(1/f) + 20 + 4 \log_2 3)/3$ bits per element. We note that the constant factor of $4/3$ in the leading term $4 \log_2(1/f)/3$ could be reduced arbitrarily close to 1 by using buckets with more items and filling them more tightly; the corresponding disadvantages would be a larger constant term, and more cells would need to be examined on each lookup when trying to find a matching remainder.

Equating $c = (r + 2)/3$, the two approaches use the same amount of space. But comparing the resulting false positive probabilities, we find

$$(2^{-\ln 2})^{(r+2)/3} > 24 \cdot 2^{-r}$$

for all integers $r \geq 7$. Indeed, the more space used, the larger the ratio between the false positive probability of the standard CBF and the dlCBF. For $r = 14$ and $c = 16/3$, for example, the two structures are the same size, but the false positive probability is over a factor of 100 smaller for the dlCBF. Moreover, the dlCBF actually uses less hashing than a standard CBF once the false positive probability is sufficiently small.

Alternatively, we might equalize the false positive probabilities. For example, using 9 4-bit counters (or 36 bits) per element with six hash functions in a standard CBF gives a false positive probability of about 0.01327. Using 11-bit remainders (or 52/3 bits per element) with the dlCBF gives a smaller false positive probability of approximately 0.01172. The dlCBF provides better performance with less than 1/2 the space of the standard CBF for this very natural parameter setting.

5 Simulation Results

5.1 A Full Example and Comparison

We have implemented a simulation of the dlCBF in order to test its performance and compare to a standard CBF. We focus here on a specific example, and extrapolate from it. We chose a table with 4 subarrays, each with 2048 buckets, and each bucket with 8 cells, for a total capacity of 2^{16} elements. Our target load is $3 \cdot 2^{14} = 49152$ elements, corresponding to an average load of six items per bucket. The approach of Section 2.3 suggests that bucket overload is sufficiently rare (on the order of 10^{-27} per set of elements) that it can be ignored.

We must also choose the size of the remainder and the number of counter bits per cell. In our example we have chosen 14 bit fingerprints, which as per our analysis of Section 4 should give us a false positive rate of slightly less than $24 \cdot 2^{-14} \approx 0.001465$. We use 2 bit counters per cell to handle cases where a hash value is repeated. The total size of our structure is therefore exactly 2^{20} bits.

In our construction, we use a "strong" hash function for the first phase (based on drand48), and random linear permutations exactly as described in Section 3.3 for the second phase.

For every experiment we perform, we do 10000 *trials*. In each trial, we initially begin with a set of 49152 elements, which we represent with the dlCBF; this corresponds to an average of six elements per bucket. We then simulate 2^{20} time steps, where in each time step we first delete an element from the current set uniformly at random, and then we insert a new element chosen uniformly at random from the much larger universe. This test is meant to exemplify the case where the dlCBF always remains near capacity, although the underlying set is constantly changing; it also matches the setting of our fluid limit equations. We test to make sure counter and bucket overload do not occur. After the 2^{20} time steps, we consider 10000 elements not in the final set, and see how many give false positives, in order to approximate the false positive rate that would be observed in practice.

We first consider the issue of overflow in the hash table. Over the 10000 trials, overflow never occurred. In fact, the fourth subarray *never*, over all insertions and deletions, had any buckets with eight elements, so overflow was never even an immediate danger. More concretely, we note that the fluid limit provides a very accurate representation of what we see in the simulation (even though we are using simple random linear permutations). Specifically, after all of the random insertions and deletions, we examine the bucket loads, and consider their distribution. As we can see in Table 1, the fluid limit matches the simulations extremely well, providing a very accurate picture of performance.

We now turn consider the cell counter. Recall that this counter is necessary to track when multiple elements have the same first round hash value. Over all 10000 trials, the largest this counter needed to be was 4. That is, on six of the trials, there were at some time four extant elements that shared the same hash value. This requires only two bits per cell counter (again assuming a sentinel value). While more precise calculations can be made, it is easy to

Table 1. Simulation results with $6n$ elements being placed into n buckets using four choices, compared to the differential equations. The simulation results give the fraction of buckets with load at least k for each k up to 9; the results are based on the final distribution of elements after 2^{20} deletions and insertions, averaged over 10000 trials. No bucket obtained a load of 9 at any time over all 10000 trials.

	Simulation Results	Steady State (Fluid limit)
Load ≥ 1	1.0000	1.0000
Load ≥ 2	0.9999	0.9999
Load ≥ 3	0.9990	0.9990
Load ≥ 4	0.9920	0.9920
Load ≥ 5	0.9502	0.9505
Load ≥ 6	0.7655	0.7669
Load ≥ 7	0.2868	0.2894
Load ≥ 8	0.0022	0.0023
Load ≥ 9	0.0000	1.681e-27

bound the probability of counter flow: for any set of m elements, the probability that any five will share the same hash value is at most $\binom{m}{5}\left(\frac{1}{|B||R|}\right)^5$, which for our parameters is approximately $5.62\text{e}-17$. Again, for most practical settings, counter overflow can be ignored, or if necessary a failsafe could be implemented.

Finally, we consider false positives. Over the 10000 trials, the fraction of false positives ranged from 0.00106 to 0.00195, with an average of slightly less than 0.001463. This matches our predicted performance almost exactly.

We emphasize again that this is a specific example, and the various performance metrics could be improved in various ways. False positives could be reduced by simply increasing the fingerprint size; space utilization could be improved by using fewer, larger buckets.

We now compare with a simulation of a standard CBF. We choose to compare by trying to achieve nearly the same false positive rate We use 13.5 counters per element (with 9 hash functions), or 663552 counters for 49152 elements. At four bits per counter, this works out to 2654208 bits, over 2.5 times the size of our dlCBF. Again, we performed 10000 trials, each having 2^{20} deletions and insertions after the initial insertion of the base set. The largest counter value obtained was 13, which appears in just one of the 10000 trials; counter values of 12 were obtained in 16 trials. These results match what one would obtain using a back-of-the-envelope calculation based on the Poisson approximation of the underlying balls and bins problem, as in e.g. Chapter 5 of [13] (or see also [10]). The approximate probability that a counter is hashed to by 16 elements (giving an overflow) in an optimal CBF configuration, where the expected number of hashed elements per counter is $\ln 2$, is approximately $\mathrm{e}^{-\ln 2}(\ln 2)^{16}/(16!) \approx 6.79 \cdot 10^{-17}$, roughly the same as the counter overflow probability for the dlCBF.

Over the 10000 trials, the fraction of false positives ranged from 0.00108 to 0.00205, with an average of slightly less than 0.001529. Again, this matches our

predicted performance almost exactly, and it is just slightly higher then the dlCBF. As in Section 4, our conclusion is that the dlCBF can provide the same functionality as the standard CBF, using much less space without a significant difference in complexity.

5.2 Additional Simulation Results

Suppose that we add more elements to the initial set, so that the average load per bucket is 6.5 instead of 6. (Specifically, in our simulations, we had 8192 buckets and 53428 elements, with each bucket having a capacity of 8 elements.) Using the fluid limit differential equations, we find that (in the steady state limit) the fraction of buckets with load at least 9 is approximately $2.205e-08$. We would therefore expect in our experiments, with just over 1 million deletions and insertions, that some bucket would reach a load of 9 (causing an overflow if buckets have capacity 8) slightly over 2 percent of the time. This indeed matches our simulations; over 10000 trials, we had an overflow condition in 254 trials. This example demonstrates the fact that in general the equations are quite useful for determining the threshold number of elements over which overflow is likely to occur.

These overflows could be mitigated by allowing elements to be moved, as discussed in section 3.3. We have implemented and tested this functionality as well. Specifically, we have focused on a simple improvement: if all the buckets associated with an inserted element are at capacity, we see if any of the items in just the *first* subarray can possibly be moved to another of its choices to resolve the overflow. This additional failsafe allowed us to handle an average load per bucket of 6.75 (or 55296 elements), without overflow in 10000 trials. A potential bucket overflow occured between 40 to 100 times in each trial, but even this limited allowance of movement allowed the potential overflows to be resolved. Greater loads could be handled by allowing more movement, and this is certainly worthy of further experimentation. For many applications, however, including the router-based applications we are considering, we believe that movement should remain a failsafe or a very rare special case. The small loss in space utilization is compensated for by simplicity of design.

6 Conclusion

We have demonstrated via both analysis and simulation that the dlCBF provides the same performance as a standard CBF using much less space. We believe the dlCBF will become a valuable tool in routing hardware and other products where the functionality of the counting Bloom filter is required.

One interesting area we hope to examine in future work is how to make the dlCBF responsive to large variations in load. The ability to move fingerprints offers one approach. Another interesting possibility is to dynamically changing the size of the fingerprint stored according to space needs.

A more general question relates to the use of d-left hashing. Since d-left hashing provides a natural way to obtain an "almost perfect" hash function, where else could it be used effectively to improve on a scheme that calls for perfect hashing?

References

1. Y. Azar, A. Broder, A. Karlin, and E. Upfal. Balanced allocations. *SIAM Journal of Computing* 29(1):180-200, 1999.
2. P. Berenbrink, A. Czumaj, A. Steger, and B. Vöcking. Balanced allocations: the heavily loaded case. In *Proc. of the 32nd Annual ACM STOC*, pp. 745–754, 2000.
3. B. Bloom. Space/time tradeoffs in in hash coding with allowable errors. *Communications of the ACM*, 13(7):422-426, 1970.
4. F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. Beyond Bloom filters: From approximate membership checks to approximate state machines. To appear in Proc. of *SIGCOMM*, 2006.
5. A. Broder and M. Mitzenmacher. Using multiple hash functions to improve IP Lookups. In *Proceedings of IEEE INFOCOM*, pp. 1454-1463, 2001.
6. A. Broder and M. Mitzenmacher. Network applications of Bloom filters: A survey. *Internet Mathematics*, 1(4):485-509, 2004.
7. S. Cohen and Y. Matias. Spectral Bloom Filters. *Proceedings of the 2003 ACM SIGMOD Conference*, pp. 241-252.
8. S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep Packet Inspection using Parallel Bloom Filters. In *IEEE Hot Interconnects 12*, 2003.
9. S. Dharmapurikar, P. Krishnamurthy, and D. Taylor. Longest prefix matching using Bloom filters. *Proceedings of the ACM SIGCOMM 2003*, pp. 201-212.
10. L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area Web cache sharing protocol. *IEEE/ACM Trans. on Networking*, 8(3):281-293, 2000.
11. K. Kenthapadi and R. Panigrahy. Balanced allocation on graphs. In *Proc. of the Seventeenth Annual ACM-SIAM Symp. on Discrete Algorithms*, pp. 434-443, 2006.
12. Y. Lu, B. Prabhakar, and F. Bonomi. Perfect Hashing for Network Applications. To appear in *Proc. of ISIT 2006.*
13. M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis.* Cambridge University Press, 2005.
14. M. Mitzenmacher and B. Vöcking. The asymptotics of selecting the shortest of two, improved. In *Analytic Methods in Applied Probability: In Memory of Fridrikh Karpelevich*, edited by Y. Suhov, American Mathematical Society, 2003.
15. R. Pagh and F. Rodler. Cuckoo Hashing. In *Proc. of the 9th Annual European Symposium on Algorithms*, pp. 121-133, 2001.
16. R. Panigrahy. Efficient hashing with lookups in two memory accesses. In *Proc. of the Sixteenth Annual ACM-SIAM Symp. on Discrete Algorithms*, pp. 830-839, 2005.
17. A. Pagh, R. Pagh, and S. Rao. An Optimal Bloom Filter Replacement. In *Proc. of the Sixteenth Annual ACM-SIAM Symp. on Discrete Algorithms*, pp. 823-829, 2005.
18. R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *Proc. of the 32nd Annual Int'l Symp. on Computer Architecture*, pp. 494–505, 2005.
19. M. Sharma and J. Byers. Scalable Coordination Techniques for Distributed Network Monitoring. *6th International Workshop on Passive and Active Network Measurement (PAM)*, pp. 349-352, 2005.
20. B. Vöcking. How asymmetry helps load balancing. In *Proceedings of the* 40^{th} *IEEE-FOCS*, pp. 131-140, 1999.

An MINLP Solution Method for a Water Network Problem

Cristiana Bragalli[1], Claudia D'Ambrosio[2], Jon Lee[3], Andrea Lodi[2,3], and Paolo Toth[2]

[1] DISTART, University of Bologna, viale Risorgimento 2, 40136 Bologna, Italy
[2] DEIS, University of Bologna, viale Risorgimento 2, 40136 Bologna, Italy
[3] IBM TJ Watson Research Center, PO Box 218, Yorktown Heights, NY 10598, USA

Abstract. We propose a solution method for a water-network optimization problem using a nonconvex continuous NLP relaxation and an MINLP search. We report successful computational experience using available MINLP software on problems from the literature and on difficult real-world instances.

Introduction

The optimal design of a WDN (Water Distribution Network) consists, in its classical formulation, of the choice of a diameter for each pipe, while other design properties are considered to be fixed (e.g., the topology and pipe lengths). From a mathematical viewpoint, we can cast the optimal design problem of a WDN as a MINLP (Mixed Integer NonLinear Programming) problem in which the discrete variables select from a set of commercially-available diameters, water flows must respect the hydraulic constraints, and we seek to minimize the cost function which only depends on the selected diameters.

Recently there has been renewed interest in optimal WDN design, due to emerging issues related to water distribution systems; in particular, the gradual deterioration of network pipes and the need for a more rational use of water resources has lead to very costly renovation activities.

Approaches in the literature use various combinations of linearization and relaxation, which lead to MILP (Mixed Integer Linear Programming), NLP NonLinear Programming) and meta-heuristic algorithms. We survey these approaches in §3. In this paper we are interested in approaches exploiting mathematical-programming formulations, and we consider two cases.

The MILP approach to our problem relies on using piecewise-linear approximations. If tractable, a solution of such a model would provide a global optimum of an approximation to the real system. If accurate models are desired for a large network, we are lead to using a large number of binary variables (to manage the linear pieces). This tends to lead to a very poor relaxation and ultimately an intractable model.

With an MINLP approach, we are lead to a more natural model. Our view is that by accurately modeling the nonlinear phenomena, we will have a model

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 696–707, 2006.

that will provide an MINLP search with a good NLP relaxation. While foregoing any hope of practically verifying MINLP global optimality of the best solution encountered, we are able to find very good solutions to large real-world instances.

Our experiments were carried out using AMPL [1] as an interface to two MINLP codes. We are using Sven Leyffer's code `MINLP_BB` [2], (available from the University of Dundee) as well as the new CMU/IBM open-source MINLP code `BONMIN` [3, 4] (to be available from COIN-OR [5]). Our modeling and solution methods are worked out with the target software in mind (in particular, the branch-and-bound implementation in `BONMIN`).

In Section §1, we formally set the notation for specifying instances of the problem. In §2, we describe the problem more fully, through a preliminary continuous model. In §3, we survey earlier approaches, and we describe an NLP model in which we make a smooth (approximate) relaxation of the preliminary model described in §2, so that we can apply methods of smooth optimization. In §4, so as to decrease the nonlinearity, we describe a reparameterization of pipes by (cross-sectional) area, rather than diameter. In §5, we describe how we incorporate binary variables for the purposes of then applying different MINLP codes. In §6, we describe the results of computational experiments.

1 Notation

The network is oriented for the sake of making a careful formulation, but flow on each pipe is not constrained in sign (i.e., it can be in either direction). The network consists of pipes (arcs) and junctions (nodes). In the optimization, the pipes are to have their diameters sized.

Sets:

E = set of pipes.
N = set of junctions.
ν = source junction (ν is a fixed element of N).
$\delta_+(i)$ = set of pipes with tail at junction i ($i \in N$).
$\delta_-(i)$ = set of pipes with head at junction i ($i \in N$).

Parameters:

$len(e)$ = length of pipe e ($e \in E$).
$k(e)$ = physical constant depending on pipe material ($e \in E$).
$dem(i)$ = demand at junction i ($i \in N$).
$elev(i)$ = physical elevation of junction i ($i \in N$).
$p_{min}(i)$ = minimum gauge pressure at junction i ($i \in N$).
$p_{max}(i)$ = maximum gauge pressure at junction i ($i \in N$).
$d_{min}(e)$ = minimum diameter of pipe e ($e \in E$).
$d_{max}(e)$ = maximum diameter of pipe e ($e \in E$).
$v_{max}(e)$ = maximum speed of pipe e ($e \in E$).

Pipes are only available from a discrete set of r_e diameters. For $e \in E$:

$$d_{min}(e) := \mathfrak{D}(e,1) < \mathfrak{D}(e,2) < \cdots < \mathfrak{D}(e,r_e) =: d_{max}(e).$$

For each pipe $e \in E$, there is a cost function $C_e()$ having a discrete specification as a (typically rapidly) increasing function of diameter. That is, $\mathfrak{C}(e,r) := C_e(\mathfrak{D}(e,r))$, $r = 1,\ldots,r_e$, where

$$\mathfrak{C}(e,1) < \mathfrak{C}(e,2) < \cdots < \mathfrak{C}(e,r_e).$$

2 A Preliminary Continuous Model

In this section, we fully describe the problem, and at the same time we develop a preliminary NLP relaxation.

Variables:

$Q(e)$ = flow in pipe e ($e \in E$).
$D(e)$ = diameter of pipe e ($e \in E$).
$H(i)$ = hydraulic head of junction i ($i \in N$).

Simple bounds [Linear]:

$d_{min}(e) \le D(e) \le d_{max}(e) \quad (\forall e \in E)$.
$p_{min}(i) + elev(i) \le H(i) \le p_{max}(i) + elev(i) \quad (\forall\, i \in N)$.

The hydraulic head of slowly flowing water is its energy per unit weight. It is expressed in terms of a height (since the pressure exerted by a column of water only depends on the height of the column). Furthermore, the hydraulic head is the sum of gauge pressure and pressure related to elevation, and all of these are measured in units of length.

Flow bounds (dependent on cross-sectional area of pipe) [Smooth but nonconvex]:

$$-\tfrac{\pi}{4} v_{max}(e) D^2(e) \le Q(e) \le \tfrac{\pi}{4} v_{max}(e) D^2(e) \quad (\forall\, e \in E).$$

Flow conservation [Linear]:

$$\textstyle\sum_{e \in \delta_-(i)} Q(e) - \sum_{e \in \delta_+(i)} Q(e) = dem(i) \quad (\forall\, i \in N \setminus \{\nu\}).$$

Head loss across links [Nonsmooth and nonconvex]:

$$H(i) - H(j) = \mathrm{sgn}(Q(e))|Q(e)|^{1.852} \cdot 10.7 \cdot len(e) \cdot k(e)^{-1.852} / D(e)^{4.87} \quad (\forall\, e = (i,j) \in E).$$

This constraint models friction loss in water pipes using the empirical Hazen-Williams equation. This is an accepted model for fully turbulent flow in *water* networks (see Walski [6]). Diameter is bounded away from 0, so the only nondifferentiability is when the flow is 0.

Objective to be minimized [Discrete]:

$\sum_{e \in E} C_e(D(e))\ len(e)$

Since we only have discretized cost data, within AMPL we are fitting a polynomial to the input discrete cost data to make a working continuous cost function $C_e()$.

We have experimented with different fits: l_1, l_2 and l_∞; with and without requiring that the fit under or over approximates the discrete points. Requiring an under approximation makes our formulation a true relaxation — in the sense that the global minimum of our relaxation is a lower bound on the discrete optimum. We use and advocate weighted fits to minimize relative error. For example, our least-squares fit for arc e minimizes

$$\sum_{r=1}^{r_e} \frac{1}{\mathfrak{C}(e,r)^2} \left[\mathfrak{C}(e,r) - \left(\sum_{j=0}^{t} \beta(j,e) \left(\frac{\pi}{4} \mathfrak{D}(e,r)^2 \right)^j \right) \right]^2 = \sum_{r=1}^{r_e} \left[1 - \left(\frac{\sum_{j=0}^{t} \beta(j,e) \left(\frac{\pi}{4} \mathfrak{D}(e,r)^2 \right)^j}{\mathfrak{C}(e,r)} \right) \right]^2$$

3 Models and Algorithms

Optimal design of a WDN has already received considerable attention. Artina and Walker [7] linearize and use an MILP approach. Savic and Walters [8] and Cunha and Sousa [9] work within an accurate mathematical model, but they use meta-heuristic approaches for the optimization, and they work with the constraints by numerical simulation. Fujiwara and De Silva [10] employ a "split-pipe model" in which each pipe e is split into r_e stretches with unknown length where r_e is the number of possible choices of the diameter of pipe e and the variables become the length of the stretches. It is not difficult to see that models of this type have the disadvantage of allowing solutions with many changes in the diameter along the length of a pipe. Using this type of model, they employ a meta-heuristic approach for the optimization, working with the constraints by numerical simulation. Eiger et al. [11] also work with a split-pipe model, but they use NLP methods for calculating a solution. Sherali et al. [12] also work with a split-pipe model, and they successfully employ global optimization methods. Xu and Goulter [13] and Lansey and Mays [14] also employ an NLP approach, but they use an approximation of the split-pipe methodology (using just 2 discrete pipe sections).

In what follows, we develop an MINLP approach and compare it to the more standard MILP approach. The MILP approach has the advantage of correctly modeling the choices of discrete diameters with binary indicator variables $x_{e,r}$ representing the assignment of diameter $\mathfrak{D}(e,r)$ to arc e. In this way we can also easily incorporate costs for the chosen diameters. There is still the nonlinearity of the flow terms in the head-loss constraints. Piecewise-linear approximation

of these nonlinear constraints is the standard MILP approach here. Unfortunately, the resulting MILPs are typically very difficult to solve. The difficulty of the MILP models is related to the fact that once the diameters have been fixed, the objective function is set, and a feasibility problem associated with the piecewise-linear approximation must be solved, without any guidance from the objective function. It turns out that linear-programming tools in such a context are not effective at all. Good feasible solutions to the models are not always obtainable for even networks of moderate size. Often one is lead to using very coarse piecewise-linear approximations to get some sort of solution, but these tend to not be accurate enough to be truly feasible. Indeed, especially with few linearization points, the MILP may (i) generate flows that are infeasible, and (ii) cut off some feasible (and potentially optimal) solutions. §6 includes some of these rather negative computational results with the MILP approach.

Instead, our preferred starting point is a fully-continuous nonconvex NLP model as described in §2. The main difficulty, besides giving up on global optimality, is to deal algorithmically with the absolute value term in the head-loss constraints. This term is nondifferentiable (at 0) but not badly. One possibility is to ignore the nondifferentiability issue, and just use a solver that will either get stuck or will handle it in its own way. This has the advantage of straightforward implementation from AMPL and access to many NLP solvers (e.g., via NEOS [15]). But since we ultimately wish to employ available MINLP solvers, and these solvers count on being given smooth NLP subproblems, we look for a more promising approach.

We suggest smoothing away the mild nondifferentiability as follows: Let $f(x) = x^p$ ($p = 1.852$) when x is nonnegative, and $f(x) = -f(-x)$ when x is negative (x is standing in for $Q(e)$). This function misbehaves at 0 (the second derivative does not exist there). Choose a small positive δ and replace f with a function g on $[-\delta, +\delta]$. Outside of the interval, we leave f alone. We will choose g to be of the following form: $g(x) = ax + bx^3 + cx^5$. In this way, we can choose a, b, c (uniquely) so that f and g agree in value, derivative and second derivative, at $x = |\delta|$. So we end up with a nice smooth-enough anti-symmetric function. It agrees in value with f at 0 and outside $[-\delta, +\delta]$. It agrees with f in the first two derivatives outside of $(-\delta, +\delta)$. Some simple calculations yields

$$
\begin{aligned}
g(x) = &\left(\frac{3\delta^{p-5}}{8} + \frac{1}{8}(p-1)p\delta^{p-5} - \frac{3}{8}p\delta^{p-5} \right) x^5 \\
&+ \left(-\frac{5\delta^{p-3}}{4} - \frac{1}{4}(p-1)p\delta^{p-3} + \frac{5}{4}p\delta^{p-3} \right) x^3 \\
&+ \left(\frac{15\delta^{p-1}}{8} + \frac{1}{8}(p-1)p\delta^{p-1} - \frac{7}{8}p\delta^{p-1} \right) x
\end{aligned}
$$

Note that $f'(0) = 0$, while $g'(0)$ is slightly positive.

As can be seen in Figure 1, this seems to work pretty well on a micro level since the function f is not so bad near $x = 0$. In the figure, we have taken $\delta = 0.1$. Indeed the quintic curve fits very well on $(-\delta, +\delta)$, and of course it matches up

to second order with the true function f at $\pm\delta$. This is all no surprise since we are operating in a small interval of 0, and the function that we approximate is not pathological. The NLP solvers that we have tested appear to respond well to this technique.

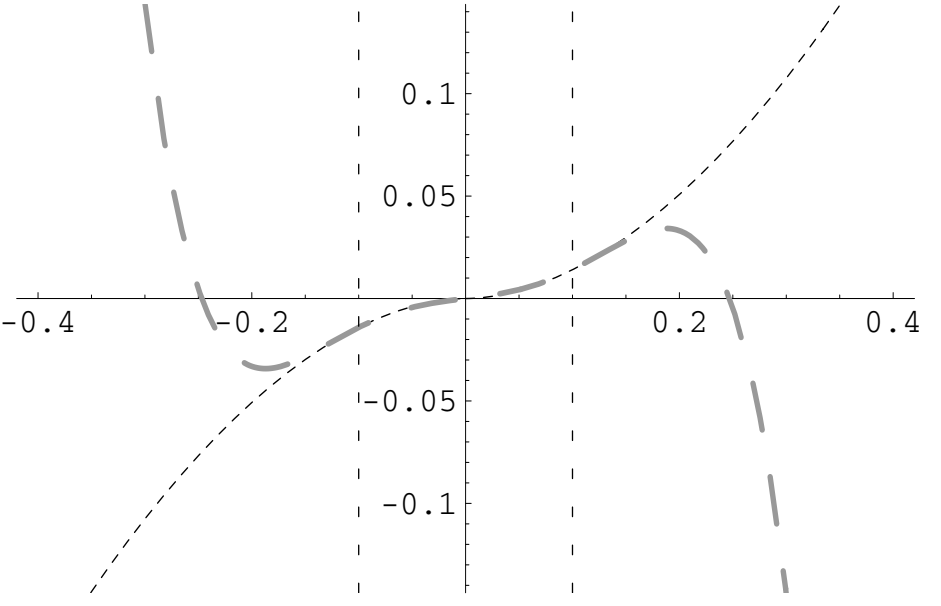


Fig. 1. Smoothing f near $x = 0$

Piecewise constraints can be modeled in AMPL (see §18.3 of [1]), so we have the advantage of being able to use a variety of NLP solvers, as well as a path to using `BONMIN` and `MINLP_BB`, both of which are interfaced with AMPL. Our experience is that the inaccuracy in using this smoothed function is minimal compared to the other inaccuracies (e.g., numerical and modeling inaccuracies).

4 Parameterizing by Area Rather Than Diameter

We can use variables

$A(e)$ = cross-sectional area of pipe e ($e \in E$),

rather than the diameter variables $D(e)$ ($e \in E$). This makes the model 'less nonlinear.' In particular, we have the *now linear* flow bounds:

$$-v_{max}(e)A(e) \le Q(e) \le v_{max}(e)A(e) \quad (\forall\ e \in E),$$

the *still linear* simple bounds:

$$\tfrac{\pi}{4}d^2_{min}(e) \le A(e) \le \tfrac{\pi}{4}d^2_{max}(e) \quad (\forall e \in E),$$

and the *less nonlinear* head loss across links constraints:

$$H(i)-H(j) = \mathrm{sgn}(Q(e))|Q(e)|^{1.852}\cdot 10.7\cdot len(e)\cdot k(e)^{-1.852}\left(\tfrac{\pi}{4}\right)^{2.435}/A(e)^{2.435} \quad (\forall\ e=(i,j)\in E).$$

Finally, there is the possibility that the cost function may be well modeled by a function that is nearly quadratic in diameter — this means *nearly linear* in area, which would be very nice.

We have tried out this area parameterization with different NLP solvers, and it seems to work well, presumably due to the fact that the model is "less nonlinear" and "less nonconvex". We will report on more testing regarding reparameterizations in the full version of this paper.

5 Discretizing the Diameters

With an eye toward using `BONMIN` as well as `MINLP_BB`, we discretized the diameters in a certain way. Specifically, we defined additional binary variables

$$X_{e,r}, \quad r=1,\ldots,r_e-1; \quad \forall e\in E.$$

These variables are used to represent diameter *increments*. That is, we have the linking equations

$$D(e) = \mathfrak{D}(e,1) + \sum_{r=2}^{r_e} \left(\mathfrak{D}(e,r)-\mathfrak{D}(e,r-1)\right) X_{e,r-1}, \quad \forall e\in E.$$

and

$$X_{e,r} \geq X_{e,r+1}, \quad \text{for } r=1,\ldots,r_e-2; \quad \forall e\in E.$$

The advantage of this incremental modeling is that branching $D(e)\leq \mathfrak{D}(e,r)$ vs $D(e)\geq \mathfrak{D}(e,r+1)$ can be realized by ordinary 0/1 branching on the single binary variable $X_{e,r}$, without requiring any special solver handling of so-called SOS (Type 1) constraints (see [16]).

If we wish to work with the area parameterization instead (see §4), we employ precisely the same discretization. That is, we keep the same 0/1 variables, but we employ the *still linear* linking equations:

$$A(e) \;=\; \frac{\pi}{4}\left(\mathfrak{D}^2(e,1) + \sum_{r=2}^{r_e}\left(\mathfrak{D}^2(e,r)-\mathfrak{D}^2(e,r-1)\right) X_{e,r-1}\right), \quad \forall e \;\in\; E.$$

6 Some Computational Results

The area parameterization seems to be better behaved than the diameter one, so we confine our reported experimental results to the area parameterization. For convenience, we define the discrete areas $\mathfrak{A}(e,r) := \frac{\pi}{4}\mathfrak{D}(e,r)^2$, for $r=1,\ldots,r_e$.

For the computational results, for approximating the cost function (see §2), we used rather high-degree polynomials, l_2 approximation, and we required that the fitted curve be a lower approximation of the discrete points.

We created an AMPL model that first fits the cost function, and then solves the continuous problem instances using a variety of NLP solvers (notably, we experimented with the Dundee solver `filterSQP` and the open-source COIN-OR solver `Ipopt`). This seems to give decent local minima without any special starting points needed. On all of our data sets, `filterSQP` and `Ipopt`, using the AMPL interface, have been able to find good local optima rather easily.

We first solve the NLP relaxation to get continuous areas $A(e)$. Then, toward using the MINLP solvers `MINLP_BB` and `BONMIN`, we are setting branching priorities as follows. If $A(e)$ is between say $\mathfrak{A}(e, r')$ and $\mathfrak{A}(e, r' + 1)$, then we let

$$prio(X_{e,r}) := 100.5 - |r' - r + 0.5|,$$

so that $prio(X_{e,r'}) = prio(X_{e,r'+1}) = 100$, $prio(X_{e,r'-1}) = prio(X_{e,r'+2}) = 99$, $prio(X_{e,r'-2}) = prio(X_{e,r'+3}) = 98$,...

Our data sets are `shamir`, `hanoi` and `foss`. For `foss`, we have three variations: `foss_poly_0`, `foss_iron`, and `foss_poly_1`. Summary statistics and computational results using MINLP for the data sets are in Table 1. For comparison, Table 2 contains results using an MILP model.

For the small network `shamir`, we obtain an MINLP solution equal to the previously best known (and almost certainly optimal) one.

For `hanoi`, which is a significantly harder problem, we also perform well. We obtain an MINLP solution that is only slightly worse than the best known one. Previously computed solution values that we know of are:

6.073×10^6, Savic and Walters [8];
6.056×10^6, Cunha and Sousa [9];
6.327×10^6, MILP (see Table 2).

In particular, we do significantly better than the solution that we obtained by MILP. Also our solution can be compared to the "split-pipe" designs obtained in the literature:

6.056×10^6, Sherali et al. [12];
6.319×10^6, Fujiwara and Khang [10].

The `foss` data is from a real problem of the Fossolo neighborhood of Bologna. In Figure 2, we have a diagram of the Fossolo network made with EPANET 2.0 [17]. EPANET is free software distributed by the US Environmental Protection Agency. It is commonly used to model the hydraulic and water quality behavior of water distribution piping systems.

We have three instances for this network. Instance `foss_poly_0` consists of the original data provided to us for this network. The pipe material for that instance is polyethylene. Our solution compares quite favorably with the solution obtained using MILP. Not only is the objective value poor for the solution obtained by

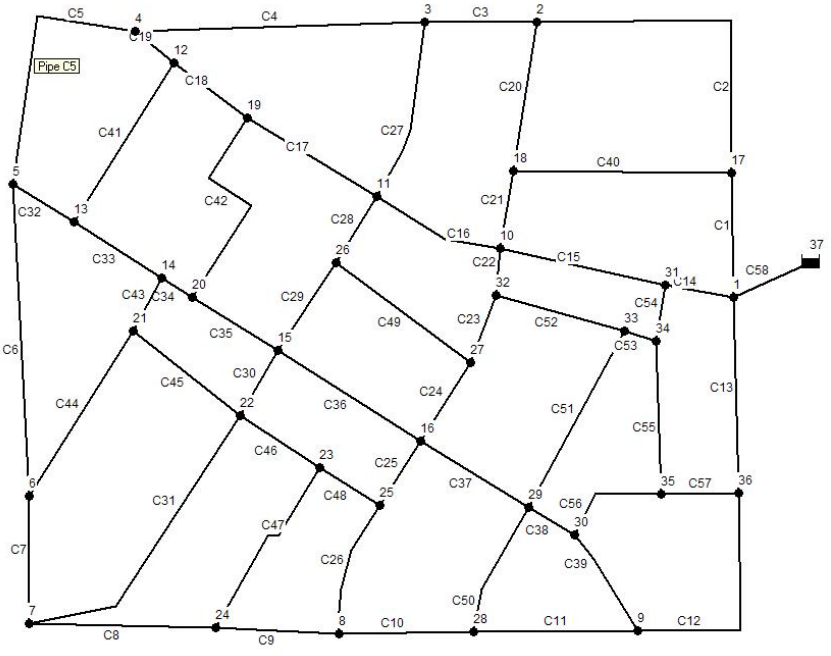


Fig. 2. Fossolo network

MILP, the piecewise-linear approximation is very coarse, and so the solution obtained can not really be considered as feasible. Instance `foss_iron` is for the same network, but with almost twice as many choices of pipe diameters and with the material being cast iron. Instance `foss_poly_1`, a polyethylene instance, is a much harder instance than the other two, with even more choices for the pipe diameters. Note that for instance `foss_poly_1`, there is a larger relative discrepancy between the value of the continuous optimum and the value of the MINLP solution that we were able to find. This suggests that there is a good possibility that we may be able to obtain a significantly better MINLP solution for this instance.

We note that the MILP model is entirely too difficult to work with for the `foss_iron` and `foss_poly_1` data sets.

The cost data for `foss_poly_0` is out of date, and so the solution values can not be directly compared to that of `foss_poly_1` and `foss_iron`, which can be reasonably compared. The value of the solution that we obtained for `foss_poly_1` is much lower than for `foss_iron`. At first this seems surprising, but this is explained by comparing the costs of the varying diameters of pipe. We see in Figure 3 that for small diameters, polyethylene is much cheaper than cast iron, and we note that the data is such that there are feasible solutions with very low flows. Although polyethylene is generally a much cheaper material than cast iron, its life is rather limited, and so cast iron is strongly preferred as a long-term solution.

The MINLP results were obtained under the following computing environment: Windows XP, Pentium M, 1.70 GHz, 1 GB RAM. The instance `shamir` required just a few seconds, and each of the other instances took 3-4 minutes for the solutions obtained. The MILP results were obtained with `CPLEX` 9.0.3 [18] under the following computing environment: Windows XP, Pentium IV, 1.70 GHz, 512 MB RAM. The MILP run times for `shamir` (resp., `hanoi`, `foss_poly_0`) were 262 (91,730, 176,960) seconds.

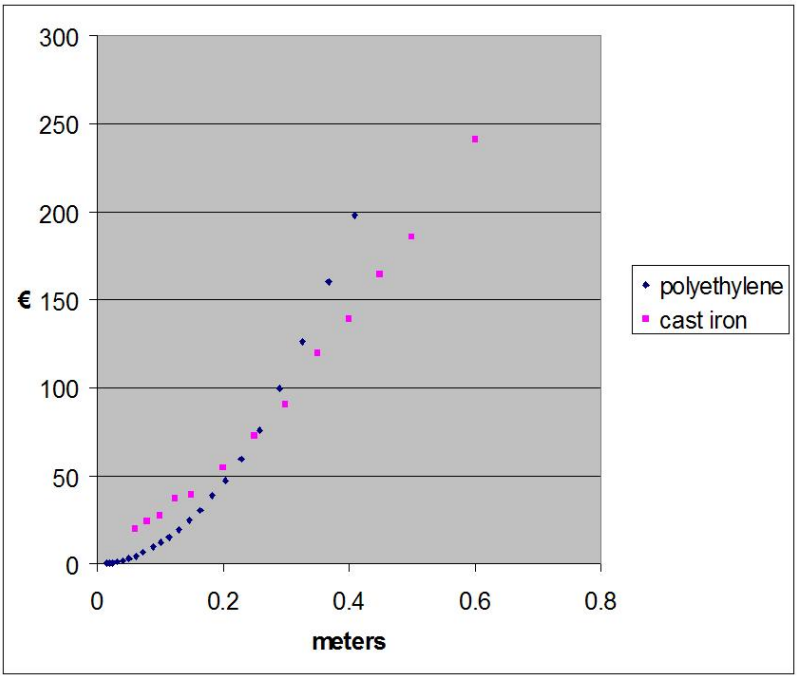


Fig. 3. Cast iron vs. polyethylene

We have experimented with restricting the range of discrete diameters to ones nearby the diameters chosen in the continuous optimum; this seems to be a very useful approach for difficult instances like `foss_poly_1`.

7 Conclusions

We are able to get good solutions to practical instances of water-network optimization problems, with very low development time. We attribute our success to:

1. The availability of software for finding good solutions to MINLP problems.
2. The easy interface to such software via the modeling language AMPL.

Table 1. Computational results for the MINLP model

Network	# junctions	# pipes	# diameters	NLP (fitted obj)	MINLP (fitted obj)	MINLP (actual obj)	Previously best known
shamir	7	8	14	425,103.06	443,295.95	419,000.00	419,000.00
hanoi	32	34	6	6,013,430.03	6,109,620.90	6,109,620.90	6,056,000.00
foss_poly_0	37	58	7	35,403.19	36,503.44	36,503.44	(46,533.38)*
foss_iron	37	58	13	178,829.52	180,373.35	178,673.70	—
foss_poly_1	37	58	22	27,827.06	31,442.21	31,178.89	—

Table 2. Computational results for MILP model

Network	Best MILP solution	LP solution	Lower bound	Gap (%)	# nodes	# nodes remaining	# linearization points
shamir	419,000.0	307,897.7	419,000.0	0.00	35,901	0	15
hanoi	6,327,613.3	5,508,664.4	6,117,905.6	3.31	4,532,718	2,592,716	7
foss_poly_0	(46,533.4)*	33,882.7	34,851.8	25.10	1,845,254	1,299,426	3

*piecewise-linearization is too coarse for us to rely on this solution as being truly feasible to the MINLP as discussed in detail in Section 3.

3. The natural framework of MINLP allows for an easy-to-develop and close model of the real system — to some extent we give up on a MILP model that seeks a globally-optimal solution, so that we can get a close MINLP model of the system which is tractable for finding good local solutions.
4. Smoothing mild nonlinearities (of the head-loss constraints) makes for good behavior of typical codes that solve the NLP subproblems.
5. Reparameterizing (by cross-sectional area rather than diameter) leads to a less nonlinear and more convex model.
6. Modeling discrete choices (of pipes) by (cross-sectional area) increments, and then setting appropriate branching priorities, enables us to mimic SOS branching while using only simple single-variable branching of the MINLP solvers.

Our belief is that much of this wisdom (omitting the parenthetical remarks above) applies to other instances of optimization problems with significant discrete and nonlinear aspects.

Acknowledgments

We thank Roger Fletcher and Sven Leyffer for giving us access to `MINLP_BB`. We thank Andreas Wächter for helping us get set up to use `MINLP_BB` and `BONMIN`. We thank Andreas for also working with us on tuning `Ipopt` to handle some nasty subproblems. Finally, we thank Pierre Bonami for making some accommodations to `BONMIN` to handle nonconvexities.

References

1. Fourer, R., Gay, D., Kernighan, B.: AMPL: A Modeling Language for Mathematical Programming. Second edn. Duxbury Press/Brooks/Cole Publishing Co. (2003)
2. Leyffer, S.: User manual for `MINLP_BB`. Technical report, University of Dundee (April 1998; revised, March 1999)
3. Bonami, P., Biegler, L., Conn, A., Cornuéjols, G., Grossmann, I., Laird, C., Lee, J., Lodi, A., Margot, F., Sawaya, N., Wächter, A.: An algorithmic framework for convex mixed integer nonlinear programs. Technical report, IBM Research Report RC23771 (2005)
4. Bonami, P., Lee, J.: `BONMIN` users' manual. Technical report (June 2006)
5. (COIN-OR) www.coin-or.org.
6. Walski, T.: Analysis of Water Distribution Systems. Van Nostrand Reinhold Company, New York, N.Y. (1984)
7. Artina, S., Walker, J.: Sull'uso della programmazione a valori misti nel dimensionamento di costo minimo di reti in pressione. In: Atti dell'Accademia delle Scienze dell'Istituto di Bologna. (Anno 271, Serie III, Tomo X, 1983)
8. Savic, D., Walters, G.: Genetic algorithms for the least-cost design of water distribution networks. ASCE Journal of Water Resources Planning and Management **123(2)** (1997) 67–77
9. Cunha, M., Sousa, J.: Water distribution network design optimization: Simulated annealing approach. J. Water Res. Plan. Manage. Div. Soc. Civ. Eng. **125(4)** (1999) 215–221
10. Fujiwara, O., Khang, D.: A two-phase decomposition method for optimal design of looped water distribution networks. Water Resources Research **26(4)** (1990) 539–549
11. Eiger, G., Shamir, U., Ben-Tal, A.: Optimal design of water distribution networks. Water Resources Research **30(9)** (1994) 2637–2646
12. Sherali, H., Subramanian, S., Loganathan, G.: Effective relaxations and partitioning schemes for solving water distribution network design problems to global optimality. Journal of Global Optimization **19** (2001) 1–26
13. Xu, C., Goulter, I.: Reliability-based optimal design of water distribution networks. Journal of Water Resources Planning and Management **125(6)** (1999) 352–362
14. Lansey, K., Mays, L.: Optimization model for water distribution system design. Journal of Hydraulic Engineering **115(10)** (1989) 1401–1418
15. (NEOS) www-neos.mcs.anl.gov/neos.
16. Beale, E., Tomlin, J.: Special facilities in a general mathematical programming system for non-convex problems using ordered sets of variables. In Lawrence, J., ed.: Proc. of the 5^{th} Int. Conf. on Operations Research. (1970) 447–454
17. (EPANET) www.epa.gov/ORD/NRMRL/wswrd/epanet.html.
18. (CPLEX) www.ilog.com/products/cplex.

Skewed Binary Search Trees

Gerth Stølting Brodal[1,⋆] and Gabriel Moruz[1]

BRICS⋆⋆, Department of Computer Science, University of Aarhus, IT Parken, Åbogade 34, DK-8200 Århus N, Denmark
{gerth, gabi}@daimi.au.dk

Abstract. It is well-known that to minimize the number of comparisons a binary search tree should be perfectly balanced. Previous work has shown that a dominating factor over the running time for a search is the number of cache faults performed, and that an appropriate memory layout of a binary search tree can reduce the number of cache faults by several hundred percent. Motivated by the fact that during a search branching to the left or right at a node does not necessarily have the same cost, e.g. because of branch prediction schemes, we in this paper study the class of skewed binary search trees. For all nodes in a skewed binary search tree the ratio between the size of the left subtree and the size of the tree is a fixed constant (a ratio of 1/2 gives perfect balanced trees). In this paper we present an experimental study of various memory layouts of static skewed binary search trees, where each element in the tree is accessed with a uniform probability. Our results show that for many of the memory layouts we consider skewed binary search trees can perform better than perfect balanced search trees. The improvements in the running time are on the order of 15%.

1 Introduction

In this paper we discuss the problem of building binary search trees that achieve good running times in practice for random queries. Theoretically, the minimum number of comparisons is achieved by perfectly balanced binary search trees, where for each given node the number of nodes in the left subtree is approximately equal to the number of nodes in the right subtree [15]. We show that in practice better running times can be achieved if we allow the search tree to be skewed, i.e. allow one of the subtrees to have more nodes than the other subtree.

When analyzing the complexity of an algorithm, usually the number of instructions performed by the CPU is counted. However, in practice there are other hardware issues besides the amount of computation that can affect the running time. During a search in a binary search tree, it is usually assumed that branching left and right at any given node inflicts the same cost on the running time. This does not always hold, since modern processors prefetch the

⋆ Supported by the Danish Natural Science Foundation.
⋆⋆ Basic Research in Computer Science, www.brics.dk, funded by the Danish National Research Foundation.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 708–719, 2006.

instructions in a pipeline and therefore must predict the outcome of the conditional branches as their outcome is not known when they enter the instruction pipeline. If a branch is incorrectly predicted, the entire pipeline must be flushed, which results in a performance loss which is proportional with the pipeline size.

In the design of algorithms, in the RAM model it is assumed that all the memory accesses take constant time. Due to the memory hierarchy on modern computers, this hardly happens in practice. The access time for a given item can vary from one CPU cycle if it is stored in a CPU register to over 10,000,000 CPU cycles if the item must be fetched from the hard-disk. Due to the high costs of memory transfers between the different levels, data is not transfered in individual items, but in contiguous *blocks*. If the memory size and the block size are known, B-trees [5] support random searches in $O(\log_B N)$ block transfers, where B is the block size. If the memory parameters are not known, cache-oblivious B-trees [6, 7] achieve the same bound. Given a tree stored in memory, Gil and Itai [14] gave optimal algorithms for computing optimal layouts, while Alstrup et al. [2] introduced faster approximate algorithms for minimizing the expected number of memory transfers and Demaine et al. [12] proved worst case bounds. Brodal et al. [9] introduced an efficient version of cache-oblivious search trees and gave experimental results on the performance of some different memory layouts for search tress.

Recently, Sanders and Winkel [16] studied the influence that branch mispredictions have over the running time of algorithms in practice. They gave a distribution based sorting algorithm and show that in certain cases branch mispredictions can be avoided by using certain processor specific instructions, namely predicative instructions. Some other works focused on the influence of branch mispredictions over the running time of sorting algorithms, both theoretically and experimentally [10, 11].

Outline. The paper is structured as follows. In Section 2 we describe skewed balanced search trees and give an upper bound on the running time performed for a random query. In Section 3 we give brief insights on the hardware issues that affect the running time in practice. For a random query we give upper bounds on the number of branch mispredictions in Section 4, while in Section 5 we introduce different memory layouts and give upper bounds on the number of cache misses. In Section 6 we describe the setup for the experiments we perform and in Section 7 we show and discuss our experimental results.

2 Skewed Binary Search Trees

A skewed binary search tree is a binary search tree where there exists a constant α, $0 < \alpha \leq 1/2$, such that for each node v there is a fixed ratio between the number of nodes in the subtree rooted in the left child and the subtrees rooted at v. More precisely, $\text{size}(\text{left}(v)) = \lfloor \alpha \cdot \text{size}(v) \rfloor$, where $\text{size}(v)$ denotes the number of nodes in the subtree rooted at v.

Skewed binary search trees are the extreme unbalanced cases of BB[α] trees of Nievergelt and Reingold [15].

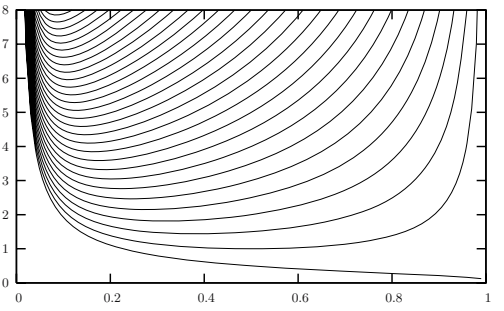


Fig. 1. Bound on the expected cost for a random search, where the cost for visiting the left child is $c_l = 1$ and the cost for processing the right child is $c_r = 0, 1, 2, \ldots, 28$ ($c_r = 0$ being the lowest curve)

Theorem 1 (Mehlhorn, Theorem 2, Section III.5.1). *The average path length P, is at most $(1 + 1/n)\log(n + 1)/H(\alpha)$, where $H(\alpha) = -\alpha\log\alpha - (1 - \alpha)\log(1 - \alpha)$.*

In practice, due to hardware issues, the running time spent at a given node might depend on the next node to process, i.e. the left or right child. In Corollary 1 we analyze the running time for a random search in the case where the costs for visiting the left and right children of a given node are different.

Corollary 1. *Consider a skewed search tree T of balance α, and let c_l and c_r be the costs for branching left and right respectively. A random search has*

$$O((\alpha c_l + (1 - \alpha)c_r)\log n/H(\alpha)) \tag{1}$$

expected cost, where $H(\alpha) = -\alpha\log\alpha - (1 - \alpha)\log(1 - \alpha)$.

Proof. Due to the linearity of expectation, the expected number of comparisons performed for a random search is equal to the average path length, which is $O(\log n/H(\alpha))$ cf. Theorem 1. If for branching left and right we have costs c_l and c_r, we obtain at a given node an expected cost of $\alpha c_l + (1 - \alpha)c_r$, since the probabilities of branching left and right are α and $1-\alpha$ respectively. We conclude that the expected cost of a random search is $O((\alpha c_l + (1-\alpha)c_r)\log n/H(\alpha))$. □

In Figure 1 we show the function from the bound (1) on the expected cost for a random search where we consider different costs for visiting the left and the right child respectively. We note that in all the cases where $c_l \neq c_r$ the minimum occurs for α values different than $1/2$.

3 Hardware Discussion

The running time of algorithms is usually analyzed by counting the instructions performed by the CPU. However, in practice, the running time of an algorithm

can be severely affected by some other hardware factors besides the CPU instructions. We show that the branch mispredictions that occur in the CPU and the cache faults can have a major effect over the running time of searching in skewed binary search trees.

To increase the clock speed, modern CPUs include instruction pipelines in their architecture, where the instructions are prefetched before being executed. When a conditional branch enters the pipeline, its outcome is not known prior to its execution and thus its direction must be predicted to ensure the prefetching of the following instructions. If the branch is incorrectly predicted, the whole pipeline must be flushed, since the instructions in the pipeline correspond to a wrong execution path. This obviously leads to a performance loss, which increases proportionally with the length of the pipeline. In such a case, we say that a *branch misprediction* occurs. Since the pipelines are getting longer and longer (e.g. 18 instructions for Pentium P4 and 31 for Intel Prescott), branch mispredictions are having an increasing influence over the running time of algorithms in practice.

In the traditional RAM model, all memory accesses are considered to have equal access times. In practice, nowadays computers have a hierarchy of memory layers, each of them having smaller size and access time than the next one, from the CPU registers to the hard-disk. The data can be transfered only between consecutive layers, and is performed in *blocks* of consecutive data rather than individual items.

4 Branch Mispredictions

Branch mispredictions can dramatically affect the running time in practice. Even though in most of the cases the branch predictors incorporated in the CPU architectures are accurate and yield good performances, in certain algorithms the outcome of certain branches is hard to guess. Sorting and searching are two such examples, since they involve comparisons among elements and the outcome of an element comparison is usually hard to predict.

There are two major types of branch prediction schemes, namely static and dynamic. In static branch predictors, each branch is predicted in the same direction at all times, and the direction of the branch is either given at compile time or it follows some simple heuristics, e.g. forward branches predicted taken and backward branches predicted not taken. On the other hand, the dynamic branch prediction schemes predict the direction of the branches at runtime, taking advantage of the execution history. In the case of searching in a balanced search tree, since the number of nodes in the left and right subtrees of a given node are approximately the same, the outcome of any branch is hard to predict and hence we expect branch mispredictions in around half of the cases. On the other hand, for the skewed search trees, we expect the number of branch mispredictions to decrease when increasing the skewness, since the probability that the search key lies in the larger subtree is increasing. In Theorem 2 we prove an upper bound on the number of branch mispredictions performed for a skewed binary search tree when a static branch predictor is used.

Theorem 2. *The expected number of branch mispredictions performed for a random search in a skewed binary search tree of balance α is $O(\alpha \log n / H(\alpha))$, where $H(\alpha) = -\alpha \log \alpha - (1-\alpha)\log(1-\alpha)$, assuming a static branch predictor and $0 \leq \alpha \leq 1/2$.*

Proof. Since we consider $\alpha \leq 1/2$, for each non-leaf node of the search tree, the right subtree will have more nodes than the left subtree, hence visiting the right subtree next is more likely than visiting the left subtree. We use a static prediction scheme where for each node we predict that the search key is larger than the key stored at the given node. Using Corollary 1 with $c_l = 1$ and $c_r = 0$, we obtain that for a random search we perform expected $O(\alpha \log n / H(\alpha))$ branch mispredictions. □

5 Memory Layouts

The difference in access times between the different layers of the memory hierarchy, especially from the internal memory to the hard disk, has led to several models that deal with capturing the cache effect. One of the most successful is the I/O model introduced by Aggarwal and Vitter [1] and consist of a two level memory hierarchy, containing a fast memory of bounded size M and a slow, infinite memory. The computation is performed in the fast memory and the data is transfered between the slow and fast memories in blocks of B consecutive items. The I/O complexity of an algorithm is given by the number of blocks transfered. Since in practice hardware architectures contain several memory levels with different values for the fast memory size M and the block size B, Frigo et al. [13] introduced the cache oblivious model. A cache oblivious algorithm is an algorithm whose analysis holds for any values of M and B. Most of the algorithms in this model assume a tall cache, i.e. $M = \Omega(B^2)$. For a comprehensive list of efficient external memory algorithms, e.g. refer to [3, 4, 8, 17].

We analyze different memory layouts for the static skewed binary search trees. For all the layouts the tree is stored as an array of n nodes, where each node is a structure containing two pointers to the left and the right subtree respectively together with an integer key. We note that the number of comparisons and branch mispredictions performed for searches is not affected by the way the tree is laid in memory, as they only depend on the height of the tree and the number of left turns on a path from the root to a certain leaf (for $\alpha < 1/2$, assuming a static branch prediction scheme). However, the number of cache faults can be dramatically affected by the memory layout, ranging from $O(1/\log B)$ to $O(1)$ I/Os for each node on a search path.

Consider a balanced binary search tree T of n nodes. The different memory layouts that we consider together with the expected number of I/Os for a random search are introduced below.

Random. Each node of T is stored at a random position in the array.

Since in this layout the nodes are stored at random locations in the array, for each node on a search path we perform an I/O, hence the expected number of I/Os is given by the average path length.

BFS. In this layout the nodes of the tree are stored according to the BFS traversal of T, where the nodes at a level are processed in a left-to-right order.

The first B nodes of the array contain the topmost subtree. In any practical setting, i.e. the tree is not severely skewed, the length of any path in this subtree is $\Theta(\log B)$. The top subtree is loaded into memory using in a single I/O, hence for the first $O(\log B)$ nodes on any path we use $O(1)$ I/Os. Afterward, for the remaining nodes on any search path we consume $O(1)$ I/Os per node, thus obtaining expected $O(1 + |P| - \log B)$ I/Os for a following a search path P.

Inorder. The tree is stored in the array according to the inorder traversal, i.e. the array is sorted.

Following a path from the root to a leaf takes $O(1)$ I/Os per node, except for possibly the last subtree of $\Theta(B)$ nodes, since they will be loaded using a single I/O. Considering the case when in a subtree of size B the length of a search path is $O(\log B)$, we obtain that for a search path P in this layout we perform between $O(|P|)$ and $O(1 + |P| - \log B)$ I/Os, where $|P|$ denotes the length of P, depending whether P reaches the bottom levels of the tree or not.

DFSl. The tree is laid out in the array according to a DFS traversal, where after visiting the root, the left child is traversed before the right child.

Since the left child is stored next to the parent, they are stored in the same block, hence branching left takes $O(1/B)$ I/Os. In what concerns the right child, accessing it requires $O(1)$ I/Os. Using Corollary 1 we obtain that for a random search we perform expected $O((\alpha/B + (1-\alpha)) \log n/H(\alpha))$ I/Os, where $H(\alpha) = -\alpha \log \alpha - (1-\alpha)\log(1-\alpha)$.

DFSr. This layout is similar to DFSl, except for the fact that the right child is traversed first and the left child afterwards. Using a similar argument, we obtain that the number of I/Os performed for a random search is expected $O(\alpha + (1-\alpha)/B) \log n/H(\alpha)$.

k-level grouping. Given a tree T, in this layout we first store the first k levels of T in the order given by a BFS traversal and then recursively store the subtrees rooted in the nodes at level $k+1$, in a right-to-left order.

Choosing $k = \log B$, we obtain that following a search path P takes $P(1 + |P|/\log B)$ I/Os, each block is loaded using $O(1)$ I/Os and in each block we process $\Theta(\log B)$ nodes of the search path, except for possibly the last block loaded. Since the expected length of P is $O(\log n/H(\alpha))$, we obtain that the expected number of memory transfers is $O(1 + (1/H(\alpha)) \cdot \log_B n)$.

pqDFS. In a preprocessing phase, for each node v we assign its weight $w(v)$ as the number of nodes contained by the subtree rooted at v. Given a parameter p, we first store consecutively the p heaviest nodes in decreasing order with respect to their weights. The subtrees rooted at the children of the nodes on the frontier, if any, are then recursively stored. The children are laid out in decreasing order of their weights. If two or more nodes have the same weight, no assumption can

be made with respect to the order in which they will be stored. To implement this layout we use a priority queue, hence its name.

To optimize the number of memory transfers, we choose $p = \Theta(B)$ and thus the group of the p heaviest nodes is stored in $O(1)$ memory blocks. For the children of the frontier of a group of p nodes the ratio between the weight of the lightest and heaviest child0 is at least α (for $0 \leq \alpha \leq 1/2$). This implies that each subtree in the frontier of the group has at most a fraction of $1/(B\alpha + 1)$ of the size of the subtree rooted at the root of the group. It follows that a search uses $O(\log_{B\alpha+1} n)$ I/Os.

Skewed van Emde Boas. This layout is a variation of the van Emde Boas layout, which is known to match in the cache-oblivious model, i.e. where the parameters M and B are not known, the best bounds known for searching in the I/O-model. Given a node v and a tree, the weight of the node is given by the number of nodes in the subtree rooted in v. Given a tree of n nodes, we split it into a top subtree containing $\lceil \sqrt{n} \rceil$ nodes and $O(\sqrt{n})$ bottom subtrees. The top subtree contains the nodes with the highest weights and the bottom subtrees have as roots the children of the leaves of the top subtree. After the splitting phase, the top and the botttom subtrees are recursively stored in consecutive memory locations.

Since the top subtree contains the heaviest $\lceil \sqrt{n} \rceil$) nodes, by a similar argument to pqDFS the ratio between the weights of the lightest and heaviest root of the bottom subtrees is at least α (for $0 \leq \alpha \leq 1/2$). If the root of the tree has weight n, we obtain that the number of nodes in each of the bottom subtrees is at most $n/(\alpha\sqrt{n} + 1)$ nodes. In the recursive layout, when $n = \Theta(B)$ searching in the corresponding subtree takes $O(1)$ I/Os. We obtain that a search takes $O(\log_{B\alpha+1} n)$ I/Os.

6 Experimental Setup

We analyze how the skewness factor α of the binary tree affects the running time in practice for the different layouts. To avoid additional costs inflicted over the running time by recursive calls, we use the iterative searching procedure in Figure 2. We generate a large sequence of random successful queries and measure the running time together with the number of comparisons, the number of branch mispredictions and the L1 data cache misses performed. We conduct our experiments on two standard Linux machines, having two different architectures. One of them has a P4 3.4 GHz CPU and 1 GB RAM, running linux 2.6.10. The other one has an AMD Athlon XP 2400+ 2.0 GHz CPU with 1GB RAM, running linux 2.6.8.1. To count the number of branch mispredictions and L1 data cache misses we use the PAPI 3.0 library. The code is compiled with gcc 3.3.2 using optimization level -O3. We will restrict ourselves to showing in the paper empirical results for AMD architecture. For the Pentium 4 processor the same behavior was observed as for the AMD architecture. The source code together with the scripts running the experiments and the plotted resulting data are available at `www.daimi.au.dk/~gabi/esa06.tar.gz`.

```
while(root!=NULLV)
{
if(key==t[root].key)
   return root;
if(key>t[root].key)
   root=t[root].right;
else
   root=t[root].left;
}
```

Fig. 2. An iterative C source code for searching

7 Experimental Results

We demonstrate experimentally that in practice the skewed binary search trees can outperform the theoretically better balanced binary search trees, because of the different costs for branching left or right.

Since the number of branch mispredictions and the amount of computation (i.e. the number of comparisons) are independent on the memory layout, we can count them on any layout. The charts in Figure 3 are obtained by counting the number of comparisons (left) and the number of branch mispredictions (right) for a tree of 25×10^3 items and 10^6 queries. As expected, the number of comparisons achieves a minimum for perfectly balanced trees, i.e. for $\alpha \approx 0.5$, and increases with the skewness of the tree. In what concerns branch mispredictions, their number increases by a factor of 350% when decreasing the skewness, following the expectation in Theorem 2. Intuitively, this happens because the more nodes one of the subtrees rooted at the children of a given node has, the more likely is that a random search path will contain that child, hence the more likely the searching conditional branch will be correctly predicted. We observe that the number of branch mispredictions has a maximum for $\alpha \approx 0.52$ and that for very high values of α the number of branch mispredictions is greater by about 25% than for very low values. This is because of the rounding for small instances, i.e. the number in the left subtree is $\lfloor \alpha n \rfloor$ which yields a rightmost path for $\alpha n < 1$.

As previously stated, the number of cache faults performed for a random search depends not only on the skewness factor α, but also on the memory layout of the tree. We first analyze the layouts that do not use blocking, that is DFSl, DFSr, BFS, Inord and Rand. In Figure 4 we give the running time (left) and the number of cache misses (right) performed by 10^6 queries in a skewed search tree of 25×10^3 nodes. As expected, the Rand layout achieves the worst running time, since it performs one cache fault for each element on a given path. Inord and BFS achieve competitive running times, whereas DFSl and DFSr are best layouts that do not use blocking, with respect to both running time and cache misses performed. We note that the Inord layout performs less cache faults and achieves better running times than BFS for very skewed trees, i.e. very small or very large values of α, whereas when the trees are almost balanced BFS outperforms Inord. Also, it is expected that DFSl and DFSr have symmetric

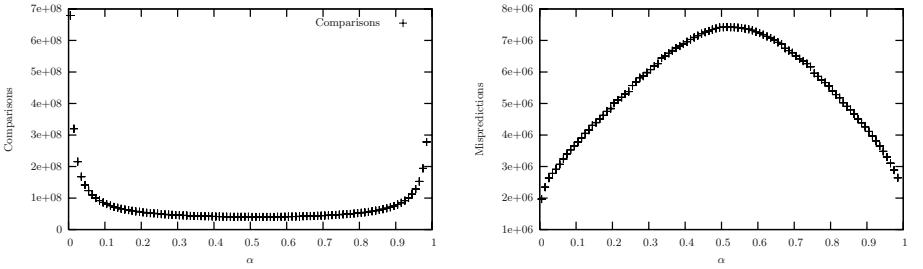


Fig. 3. The number of comparisons (left) and branch mispredictions (right) performed by a skewed search tree of 25×10^3 items for 10^6 queries

charts for the number of cache misses and implicitly the running time, since they are symmetric layouts, where DFSr is efficient for $\alpha < 0.5$, since there are more nodes in the right subtree, and DFSl is more efficient for $\alpha > 0.5$. We recall that in the case of DFSr, since the right child is recursively stored after the root, branching right takes $O(1/B)$ I/Os whereas we spend $O(1)$ I/Os for branching left, whereas in the case of DFSl we spend $O(1/B)$ I/Os for branching left and $O(1)$ I/Os for branching right. We note that the minimum running time is achieved for $\alpha \approx 0.2$ in the case of DFSr and for $\alpha \approx 0.75$, and is better by around 15% compared to $\alpha = 0.5$. In DFSr, for $0.2 < \alpha \leq 0.5$, even though less comparisons are performed, both cache faults and branch mispredictions increase and the overall running time increases too.

We now analyze the blocked layouts. We conduct experiments for tuning the parameterized layouts, i.e. k-level grouping and pqDFS. Again, we perform 10^6 queries on a skewed search tree of 25×10^3 nodes, for different values of the parameters. For k-level grouping, we give experimental data for different values of the parameter k, i.e. the number of levels grouped together in the layout, for different values of α. For each pair of values for k and α, we perform three series of queries and select the median of the running times. For each value of the parameter k we choose the smallest running time among the different possible skewness factors. The data we obtained is shown in Figure 5 (left). The differences in the running times are up to 5%, and the minimum running time is achieved for $k = 2$, i.e. when two levels of the tree are grouped together. Thus, in our further experiments involving this layout we use this value.

We perform the same experiments for the pqDFS layout, varying the number p of the heaviest nodes grouped in a block, see Figure 5 (right). Unlike the k-level grouping, in this case the differences in the running times are very small. Since the minimum running time was obtained when grouping $p \approx 40$ nodes together, in the further experiments we are using $p = 40$.

We perform a comparative study for the blocked layouts, i.e. k-level grouping, pqDFS, and skewed van Emde Boas, together with DFSr, since it is the non-blocked layout that achieved the best running time. In Figure 6 we show the running times (left) and the number of cache misses (right) performed for these layouts on a skewed binary search tree of 25×10^3 nodes for 10^6 queries. We note that even though all layouts achieve approximately the same running times, at

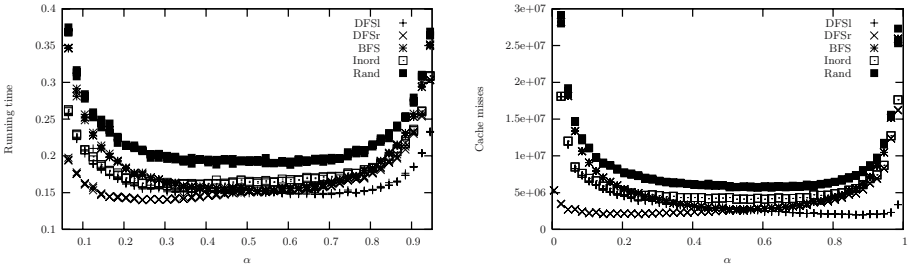


Fig. 4. The running time (left) and the number of L1 data cache misses (right) performed by a skewed search tree of 25×10^3 items for 10^6 queries for the non-blocked layouts

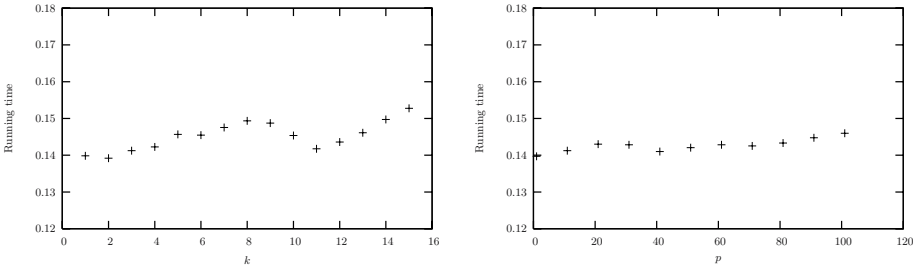


Fig. 5. The best running times for k-level grouping (left) and pqDFS where p nodes are grouped together (right), for 10^6 queries and a skewed search tree of 25×10^3 nodes

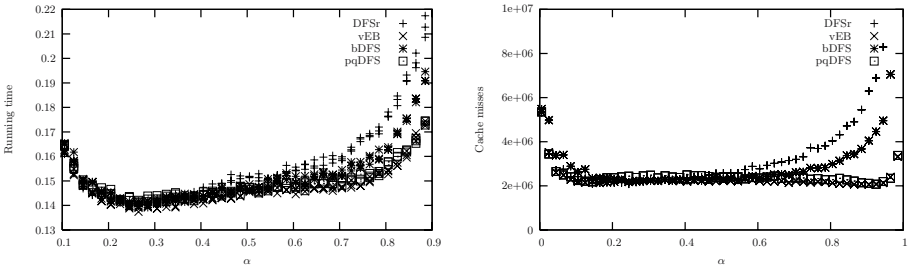


Fig. 6. The running time (left) and the number of L1 data cache misses (right) performed by a skewed search tree of 25×10^3 nodes for 10^6 queries for the blocked layouts

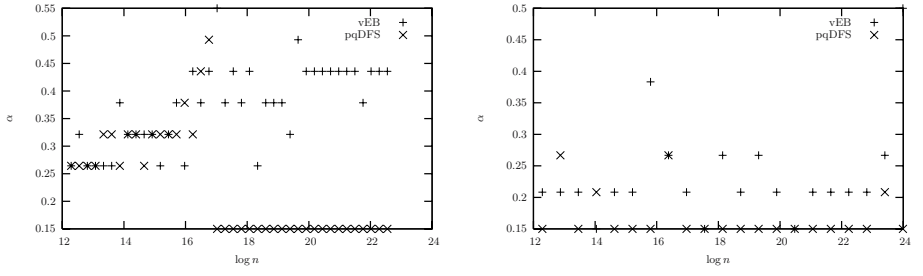


Fig. 7. The skewness factors that achieved the minimum running times for different tree sizes for the Athlon (left) and P4 (right) architectures

all times the skewed van Emde Boas is the fastest. The heuristics of grouping the heavy nodes achieves good results in practice, since pqDFS is faster than blocking k levels (bDFS). Finally, we note that DFSr is slightly slower than the blocked layouts. In what concerns the data cache misses, for all the algorithms the number of data cache misses is almost similar and is approximately the same regardless of the skewness factor for $\alpha < 0.5$, except for the case when the tree is extremely skewed, i.e. for very small values of α. We note when increasing the skewness factor α up to 0.5, the number of comparisons decreases, the number of cache misses is approximately the same except for extremely low values of α, whereas the number of branch mispredictions is increasing. The resulting effect is that the minimum running time is achieved for $\alpha \approx 0.3$, and is better by a factor of 5% compared to the perfectly balanced search trees for all the blocked layouts. As stated before, for DFSr, the observed improvement in the running time is up to 15%. In what concerns the number of caches, the blocked layouts performed much better than the non-blocked layouts, as the skewed van Emde Boas and pqDFS layouts achieve significant improvements against BFS, Inord and Rand.

Finally, we study for which values of the skewness factor α we achieve the minimum running time when varying the size of the tree. We choose to perform our experiments on two of the layouts that achieved the best running times, namely pqDFS and the skewed van Emde Boas. For a given tree size, we vary the skewness factor α and for each value of α we perform three series of 10^6 queries and pick the median of the running times. We then measure the skewness factor for which the minimum running time was achieved. In Figure 7, we show the resulting data for both the AMD (left) and P4 (right) architectures. We notice that for both architectures the pqDFS achieves its best running time for smaller values of α than skewed van Emde Boas. Also, the best skewness factor is increasing while increasing the input size in the case of the AMD architecture, whereas for the P4 it has a constant behavior when increasing the input size.

References

1. A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
2. S. Alstrup, M. A. Bender, E. D. Demaine, M. Farach-Colton, J. I. Munro, T. Rauhe, and M. Thorup. Efficient tree layout in a multilevel memory hierarchy. Manuscript, 2003.
3. L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*, pages 313–358. Kluwer Academic Publishers, 2002.
4. L. Arge, G. S. Brodal, and R. Fagerberg. Cache-oblivious data structures. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, page 27. CRC Press, 2004.
5. R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.

6. M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proc. 41st Annual Symposium on Foundations of Computer Science*, pages 399–409, 2000.
7. M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. In *Proc. 13th Annual ACM-SIAM symposium on Discrete algorithms*, pages 29–38, 2002.
8. G. S. Brodal. Cache-oblivious algorithms and data structures. In *Proc. 9th Scandinavian Workshop on Algorithm Theory*, volume 3111 of *Lecture Notes in Computer Science*, pages 3–13. Springer Verlag, Berlin, 2004.
9. G. S. Brodal, R. Fagerberg, and R. Jacob. Cache-oblivious search trees via binary trees of small height. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 39–48, 2002.
10. G. S. Brodal, R. Fagerberg, and G. Moruz. On the adaptiveness of quicksort. In *Proc. 7th Workshop on Algorithm Engineering and Experiments*, pages 130–140, 2005.
11. G. S. Brodal and G. Moruz. Tradeoffs between branch mispredictions and comparisons for sorting algorithms. In *Proc. 9th International Workshop on Algorithms and Data Structures*, volume 3608 of *Lecture Notes in Computer Science*, pages 385–395. Springer Verlag, Berlin, 2005.
12. E. D. Demaine, J. Iacono, and S. Langerman. Worst-case optimal tree layout in a memory hierarchy. Manuscript, August 2004.
13. M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache oblivious algorithms. In *40th Annual IEEE Symposium on Foundations of Computer Science*, pages 285–298, 1999.
14. J. Gil and A. Itai. How to pack trees. *Journal of Algorithms*, 32(2):108–132, 1999.
15. J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. In *Proc. 4th Annual ACM Symposium on Theory of Computing*, pages 137–142, 1972.
16. P. Sanders and S. Winkel. Super scalar sample sort. In *Proc. 12th European Symposium on Algorithms*, volume 3221 of *Lecture Notes in Computer Science*, pages 784–796. Springer Verlag, Berlin, 2004.
17. J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.

Algorithmic Aspects of Proportional Symbol Maps

S. Cabello[1], H. Haverkort[2], M. van Kreveld[3], and B. Speckmann[2]

[1] Department of Mathematics, Institute for Mathematics, Physics and Mechanics, Ljubljana, Slovenia. sergio.cabello@imfm.uni-lj.si
[2] Department of Mathematics and Computer Science, TU Eindhoven, The Netherlands, cs.herman@haverkort.net, speckman@win.tue.nl
[3] Institute for Information and Computing Sciences, Utrecht University, The Netherlands, marc@cs.uu.nl

Abstract. Proportional symbol maps visualize numerical data associated with point locations by placing a scaled symbol—typically opaque disks or squares—at the corresponding point on a map. Overlapping symbols need to be drawn in such a way that the user can still judge their relative sizes accurately.

We identify two types of suitable drawings: *physically realizable drawings* and *stacking drawings*. For these we study the following two problems: Max-Min—maximize the minimum visible boundary length of each symbol—and Max-Total—maximize the total visible boundary length over all symbols. We show that both problems are NP-hard for physically realizable drawings. Max-Min can be solved in $O(n^2 \log n)$ time for stacking drawings, which can be improved to $O(n \log n)$ or $O(n \log^2 n)$ time when the input has certain properties. We also experimented with four methods to compute stacking drawings: our solution to the Max-Min problem performs best on the data sets considered.

1 Introduction

Proportional symbols maps, also known as *graduated symbol maps*, are a well established cartographic tool to visualize quantitative data that is associated with specific (point) locations. A symbol, most commonly a disk or a square, is scaled such that its area corresponds to the data value associated with a point and then placed at exactly that point on a geographic map. The spatial distribution of the data can then be observed by studying the spatial distribution of the differently sized symbols. Typical data

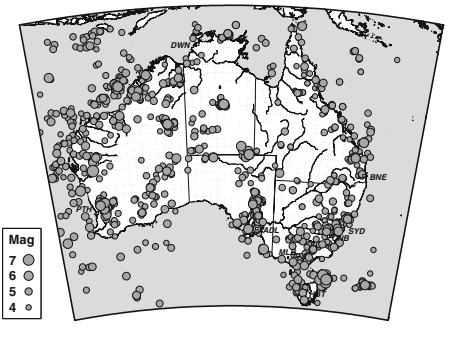


Fig. 1. A proportional symbol map depicting Australian earthquakes of size > 4.0 on the Richter scale [12].

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 720–731, 2006.

that are visualized in this way include the magnitude of earthquakes (see Fig. 1), the production of oil wells, or the temperature at weather stations.

A proportional symbol map communicates its message via the sizes of its symbols—both the actual size of the symbols and the ratio between symbol sizes. There exists a large amount of theory and user studies that discuss which sizing communicates the difference between quantities in the most effective way. See the books by Dent [5] and Slocum et al. [13] for an extensive overview.

While it is commonly agreed upon that a map should appear "neither 'too full' nor 'too empty'" [13] it is unclear how much the symbols on a proportional symbol map should overlap. Small symbols create little or no overlap but spatial patterns are difficult to detect. On the other hand, large symbols result in a cluttered map where it is difficult to identify and judge individual symbols. Determining the ideal size for the symbols is a major issue when constructing proportional symbol maps, but every "good" map will contain at least some overlapping symbols (see the discussion in [13]).

In principle any two- or three-dimensional shape can be used as a symbol on a proportional symbol map. However, circles (transparent) and disks (opaque) are used most frequently, since they are visually stable, they conserve map space, and users do prefer them. Also squares and triangles are occasionally seen. Although opaque symbols obscure each other and also the map below them, users indicate a preference for opaque symbols [8].

Clearly there are many different ways to arrange opaque symbols with respect to each other and any choice of (partial) order makes some symbols more visible than others. In this paper we address the algorithmic question how to arrange a given set of overlapping disks or squares such that all of them can be seen "as well as possible".

Definitions and notation. Before we can formally state the problem we first introduce some definitions and notation. To simplify the presentation we give all definitions for disks, but they naturally extend to opaque squares. Let S be a set of n disks $D_1, \ldots, D_n$ in the plane. We denote by $\mathcal{S}$ the arrangement formed by the boundaries of the disks in S. A *drawing* $\mathcal{D}$ of S is a sub-arrangement of $\mathcal{S}$ which is drawn on top of the filled interiors of the disks in S.

Not every drawing is suitable for the use on a proportional symbol map. A suitable drawing needs to include at least the boundary of the union of the disks in S. It should be locally correct at the vertices: every vertex v of the drawing is formed by the intersection of the boundaries of two disks D_i and D_j; a drawing

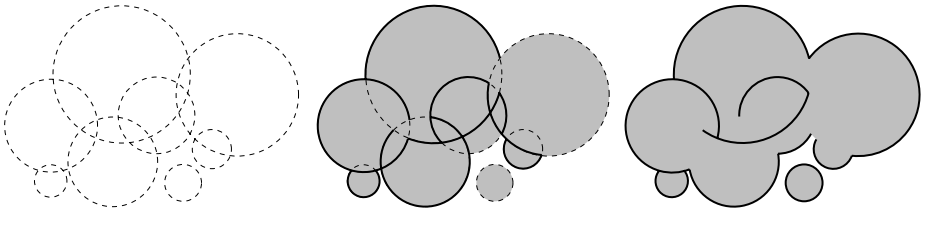


Fig. 2. An arrangement $\mathcal{S}$, a drawing with $\mathcal{S}$ visible, and a bounded drawing

is locally correct at v if it corresponds locally around v to stacking D_i onto D_j or vice versa. Furthermore, a suitable drawing must have only correct faces: a face of the drawing is correct if there is an order in which all disks in S that contain the face can be drawn on top of each other such that the face appears. We call drawings that satisfy these conditions *face correct*.

Figure 3 shows that even a face correct drawing can still have an "Escher-like" quality which we would like to avoid on a proportional symbol map. Hence we need to enforce even stronger requirements on what constitutes a proper drawing. We consider two types of drawings.

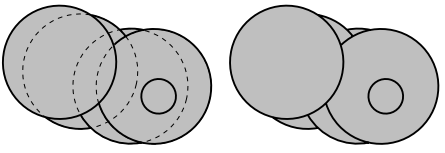


Fig. 3. A face correct drawing shown with and without $\mathcal{S}$ visible

Physically realizable drawings. A face correct drawing is physically realizable if and only if for every face f of the arrangement $\mathcal{S}$ there exists a total order on the disks in S_f (the disks in S that contain f) such that the topmost disk is visible and the orders associated with any two faces of $\mathcal{S}$ do not conflict. That is, the order in which the disks in S are stacked upon each other is uniquely determined at every face of $\mathcal{S}$ and no two of such orders conflict. In particular, any two or more disks that have a common intersection have a unique ordering.

We can show that this definition is in fact equivalent to the following. We associate a *pr-disk* D_i' with every disk D_i in S. D_i' is a surface patch that is the image of a continuous function of the points in the input disk, that is, $(x, y) \in D_i$ maps to $(x, y, f_i(x, y))$ where $f_i(\cdot, \cdot)$ is continuous. The boundary of D_i' is a closed curve that lies in a cylinder erected vertically up on the boundary of D_i. A drawing $\mathcal{D}$ is physically realizable if functions $f_1, \ldots, f_n$ exist so that the pr-disks $D_1', \ldots, D_n'$ are disjoint and the view vertically down from infinity is $\mathcal{D}$. That is, if we imagine that we are working with actual physical disks then we are allowed to warp them in a "Dali-like" fashion, but we cannot cut them.

Stacking drawings. A stacking drawing is a natural restriction of a physically realizable drawing and also the one most frequently found on proportional symbol maps. A physically realizable drawing $\mathcal{D}$ is a stacking drawing if there exists a total order on the disks in S such that $\mathcal{D}$ is the result of stacking the disks in this order.

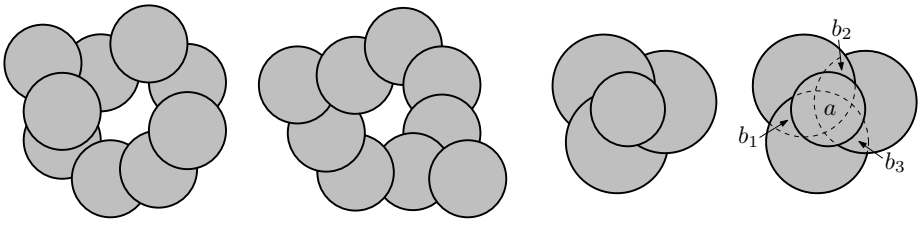


Fig. 4. A stacking drawing (left), a physically realizable drawing that is not a stacking drawing (middle), a drawing that may seem physically realizable, but is not—any order for face a will conflict with one of b_1, b_2, or b_3 (right)

Quality of a drawing. Intuitively, a good drawing should enable the viewer to see at least some part of all symbols and to judge their sizes as correctly as possible. The accuracy with which the size of a symbol can be judged is proportional to the portion of its boundary that is visible. This leads us to the following two optimization problems. Assume that we are given a set S of n opaque symbols $S_1, \ldots, S_n$.

Max-Min: Find a physically realizable or a stacking drawing that maximizes the minimum visible boundary length of each symbol, that is, $\max \min_{1 \le i \le n} \{\text{visible length of the boundary of } S_i\}$.

Max-Total: Find a physically realizable or a stacking drawing that maximizes the total visible boundary length over all symbols.

Figure 5 illustrates why we consider only visible boundary length and not visible area of symbols. The boundary of the center disk is completely covered but a significant part of its area is still visible. It is, however, impossible to judge its size or to determine the location of its center. Figure 6 shows that a stacking drawing can be arbitrarily much worse than a physically realizable drawing with respect to the Max-Min problem. At least half of the boundary of every disk in Figure 6 (left) is visible, whereas the lowest disk in any stacking drawing is covered by its two neighbors and hence has only a very short visible boundary.

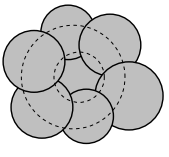


Fig. 5. Visible perimeter is more important than visible area

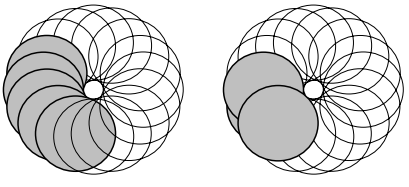


Fig. 6. An optimal physically realizable drawing (left), an optimal stacking drawing for the same disks (right)

Formal problem statement. Assume that we are given a set S of n disks or opaque squares that overlap. Construct a physically realizable drawing or a stacking drawing for the elements of S that either maximizes the minimum visible boundary of each symbol (Max-Min) or maximizes the total visible boundary over all symbols (Max-Total).

Results. We show in Section 2 that for physically realizable drawings both the Max-Min and the Max-Total problems are NP-hard. For stacking drawings the Max-Min problem can be solved in $O(n^2 \log n)$ time. If the symbols are disks and have the property that no point in the plane is covered by more than $O(1)$ disks, then it can be solved in $O(n \log n)$ time. If the symbols are unit-size squares it can be solved in $O(n \log^2 n)$ time. These algorithmic results are presented in Section 3. The status of the Max-Total problem for stacking drawings is open. We performed experiments to compare the results of four different methods that compute a stacking drawing. One of these is our solution to the Max-Min problem, and this one performs best on our data sets. These results are presented with various figures in Section 4.

2 NP-Hardness

We show that the Max-Min and the Max-Total problem are NP-hard for physically realizable drawings. It is better for our discussion to use the standard RAM model of computation and consider disks or squares with integer radius and centers located at integer coordinates. In this model, the visible boundary of a set of squares is easily computable, since it involves only integer numbers. However, for problems involving disks, the visible perimeter is a sum of the lengths of circular arcs, and therefore, it is unclear if they belong to the class NP. This is not surprising, since many basic geometric problems are not known to be in NP [4, 6]. Both reductions are from planar 3-SAT, which was proved NP-hard by Lichtenstein [10]. Since the ideas are standard and often used (see for example [1, 7, 9]), our discussion concentrates mostly on the gadgets.

Theorem 1. *It is NP-hard to decide if a given collection of congruent disks has a physically realizable drawing where at least some given length of the perimeter of each disk is visible.*

Proof. We sketch a construction with disks of perimeter 1 and radius $1/2\pi$, such that it is NP-hard to decide if there is a physically realizable drawing with at least 3/4 of each disk's boundary visible. We explain in the full paper how an equivalent construction can be made in polynomial time in the RAM model.

A Boolean variable x_i is represented by an even cycle of disks, as shown in Figure 7. We say that two disks overlap for a fraction f if a fraction f of the boundary of one disk is covered by the other disk. Any two adjacent disks overlap for 1/8 or 1/4, such that any disk overlaps for 1/8 with one neighbor and for 1/4 with the other neighbor. Hence, to achieve that each disk has 3/4 of its perimeter visible, the cycle must be either clockwise overlapping (signifying that x_i is TRUE) or counterclockwise overlapping (signifying that x_i is FALSE).

In the TRUE state, every second disk has 1/4 of its boundary covered by the next disk in the cycle—if 3/4 of the boundary of each of these disks is to remain visible, no more disks (other than those in the cycle) must cover them. For the other half of the disks in the cycle, only 1/8 of their boundaries are covered and disks outside the cycle may cover another fraction 1/8 of them. In the FALSE state, it is precisely the other set of disks that can be covered for another 1/8.

A channel for x_i may start at a disk that has 1/8 overlap with a disk in the cycle that can take another 1/8 overlap in the TRUE state. A channel for $\overline{x_i}$ may

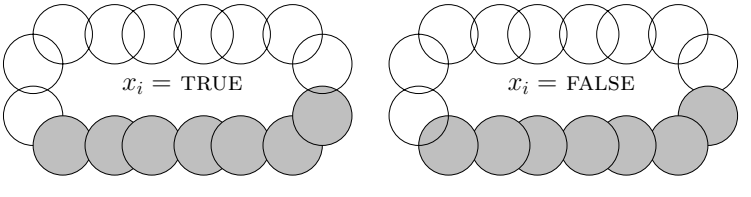


Fig. 7. Representation of a Boolean variable and its TRUE and FALSE states

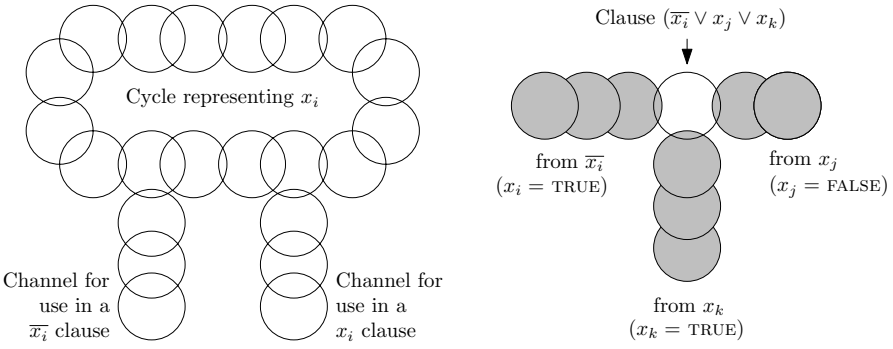


Fig. 8. Representation of a clause

start at a disk that has 1/8 overlap with a disk in the cycle that can take another 1/8 overlap in the FALSE state. If the variable has enough disks in its cycle, then any number of channels can be connected and in any order for x_i and $\overline{x_i}$.

At a clause like $(\overline{x}_i \vee x_j \vee x_k)$, the channels for $\overline{x}_i$, x_j, and x_k come close and are connected with a single disk that has overlap 1/8 with each of the last disks of the channel, see Figure 8. For the clause disk to be uncovered for at least 3/4, at least one of the three channels must represent TRUE, and the last disk of that channel can go under the clause disk.

The details of the construction are easily filled in. The disks have a physical realization with a free perimeter of at least 3/4 if and only if the planar 3-SAT formula is satisfiable, and only a polynomial number of disks are needed in the reduction. Moreover, note that if it is not possible to achieve a free perimeter of at least 3/4 in all disks, then some disk has free perimeter of exactly 5/8. □

Note that in the proof, no point in the plane is contained in more than two disks. Furthermore, the decision whether a physically realizable drawing exists with free perimeter strictly between some values A and B is equally difficult as the decision whether a physically realizable drawing exists with free perimeter at least A, for some constants A and B. This shows that we cannot expect to find an approximation algorithm that finds a solution better than a constant factor from the optimum in polynomial time. In the given construction, with an appropriate scaling, the constant can be brought arbitrarily close to 6/5, but with some fine-tuning it can be raised.

A reduction for Max-Total when the input consists of squares can be found in the full version of this paper. Here we only state the result.

Theorem 2. *It is NP-complete to decide if a given collection of bounded-size squares has a physically realizable drawing whose visible perimeter is at least a given value T.*

Our construction uses coordinates and squares of polynomial size. Therefore, there is no fully polynomial time approximation scheme (FPTAS) for the optimization problem, unless P=NP.

3 Algorithms

We can compute the stacking order that maximizes the minimum of the visible boundary in polynomial time. We present the algorithms in this section. We first give a general algorithm for disks, then we deal with special cases and squares.

The general idea to compute a stacking order of n disks that maximizes the minimum of the boundary length uncovered is simple: for each disk, we determine how much boundary would be seen if it were the bottommost disk. We choose that disk with the maximum value, make it the bottommost disk, and then recurse on the $n-1$ remaining disks. To implement this greedy approach efficiently, we maintain for each disk a data structure that represents all of its uncovered boundary intervals. For technical reasons, we consider a disk boundary c_i to be an interval from its topmost point clockwise around. Any other disk d_j intersects c_i in zero, one or two intervals (two if d_j contains the topmost point of c_i). All intersection points on c_i define a set of elementary intervals between two consecutive intersection points. The data structure T_i that stores c_i is a variation of a segment tree that stores the elementary intervals in its leaves. An internal node ν corresponds to an interval $int(\nu)$ that is the union of elementary intervals below it in T_i. (See [3] for a detailed description of segment trees.)

Every node (internal and leaf) stores the boundary length of $int(\nu)$ and a counter that stores the number of other disks that contain $int(\nu)$, but not $int(\text{parent}(\nu))$. It also stores an interval *vis-int*(ν) that is the visible boundary length of $int(\nu)$ that would remain if only the disk intervals of other disks that occur in the subtree rooted at ν would hide parts of $int(\nu)$ from view. Disk intervals at ancestors of ν may still cause that no part of $int(\nu)$ is actually visible. The root of T_i stores the total perimeter length of d_i if it were placed bottommost in *vis-int*$(\text{root}(T_i))$.

Initially, we construct a segment tree T_i for each disk d_i, storing the disk intervals for all disks d_j with $j \neq i$. By inspecting *vis-int*(root) for all trees $T_1, \ldots, T_n$, we determine the one with the largest boundary length if it were bottommost, and select it. When a disk d_j is chosen, we delete the disk interval of d_j from all structures T_i of disks d_i that intersect d_j and were not yet chosen. To this end, we find the canonical nodes of the disk interval of d_j in T_i. For each canonical node ν, we lower the counter. When the counter becomes 0, we also update *vis-int*(ν) by taking the *vis-int*(..) values of the two children of ν, and adding their values. By the standard analysis of segment trees and tree augmentation, deletion of a disk interval from T_i takes $O(\log n)$ time. Therefore, the process of choosing a disk to be placed bottommost, and updating all trees takes $O(n \log n)$ time.

Theorem 3. *Given n disks in the plane, a stacking order maximizing the boundary length of the disk that is least visible can be computed in $O(n^2 \log n)$ time.*

If no point in the plane is contained in more than C disks, where C is some constant, and the ratio in size of the largest and smallest disk is also a constant, then we can prove with a packing argument that any disk intersects only a constant number of other disks. The whole arrangement of disk boundaries has

complexity $O(n)$, and for each disk d_i, we find the ones that intersect its boundary in constant time by walking in the arrangement around the boundary of d_i and inside it. We store all disks in a priority queue, sorted on visible boundary length. This allows us to select the next bottommost disk in $O(1)$ time, find the intersecting disks in $O(1)$ time as well, recompute their visible boundary length in $O(1)$ time, and recompute their position in the priority queue in $O(\log n)$ time. In this way, the whole algorithm takes only $O(n \log n)$ time overall.

If we do not assume that the ratio in size of the smallest and largest disks are bounded, we can prove the same result. A disk may now intersect many more than $O(1)$ disks, but the arrangement still has $O(n)$ complexity, and hence all traversals to find the disks that intersect a selected disk take only $O(n)$ time in total. This implies that in total, only $O(n)$ visible boundary lengths are recomputed. We need to take care that this can be done efficiently for disks that intersect many other disks. To this end, we use the segment tree given above for the general algorithm, and we again obtain an $O(n \log n)$ time algorithm (note that all segment trees together have size $O(n)$ in this case).

Theorem 4. *Given a set of n disks in the plane such that no point is contained in more than $O(1)$ disks, a stacking order that maximizes the boundary length of the disk that is least visible can be computed in $O(n \log n)$ time.*

For unit squares we can give an $O(n \log^2 n)$ time algorithm without the assumption that any point is covered by only a constant number of squares. So the arrangement of squares may have quadratic complexity. We first compute the union of all squares, and determine for each square the visible boundary length that it contributes to the boundary of the union. We store the squares in a priority queue. Note that any square has at most one visible interval on each side. We select the next bottommost square S from the priority queue and update the union of squares explicitly. Up to four segments disappear, but possibly many more appear, see Figure 9. We find these by repeated ray shooting. Up to eight squares have a visible interval enlarged because they ended on a side of S. All other squares that have a change in their visible interval have a vertex exposed on the contour, or have an interval on a side exposed for the first time; these cases can arise at most four times for each square. Hence, the total change in intervals is $O(n)$ throughout the whole process.

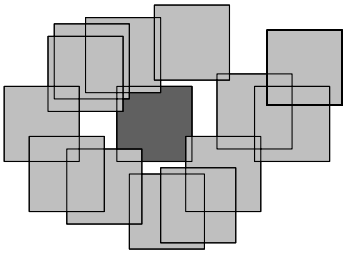


Fig. 9. Deletion of a square from the union of squares

We preprocess all vertical sides of squares in a semi-dynamic data structure for horizontal ray shooting. Similarly, we preprocess all horizontal sides of squares in a semi-dynamic data structure for vertical ray shooting. Using an augmented segment tree we can implement this to run in $O(n \log^2 n)$ time overall.

Theorem 5. *Given a set of n unit squares in the plane, a stacking order that maximizes the visible boundary of the square that is least visible can be computed in $O(n \log^2 n)$ time.*

4 Experiments

We have examined stacking orders based on different methods experimentally. We first describe our data sets and then the stacking methods, followed by an evaluation of their quality.

Data sets. In principle there are three different types of data sets. Either all disks have the same size, or disk sizes are taken from a small number of classes, or the disk sizes are all different. Equal size disks are uncommon because they do not show a value with locations, only an occurrence. We therefore omit such data sets from our experiments.

We used several different data sets; see the table. First, we took two data sets with the cities of the USA, namely the 156 and the 538 largest ones by population. The area of each disk is proportional to the population of the city. Two other data sets consist of 602 disks corresponding to earthquakes in the world. Disks are centered at the epicenter and the areas of the disks are proportional to the magnitude (scale of Richter) and to the death count [11]. Second, we used versions of the 156 cities and the earthquake magnitudes where disk sizes were classified into five different classes.

Test data	number of disks	smallest radius	largest radius
City 156	156	1.428	12
City 538	538	0.279	6
Earthquake magnitude	602	8.075	10
Earthquake death count	602	0.123	100
City 156, classed	156	1.897	6
Earthquake mag., classed	602	4.472	10

Stacking methods. Proportional symbol maps that are published in books or on the internet do not seem to follow any method consistently. Some appear to be stacked from the left to the right, others appear to be random. For maps with differently sized disks, often the stacking order is from large to small (small on top). For disks of arbitrary sizes we compare four different stacking methods.

Left-to-right by center: The disk with leftmost center is put at the bottom of the stacking order, and the remaining disks are stacked recursively on top.

Left-to-right by leftmost: The disk with leftmost left extreme is put at the bottom, with the remaining disks stacked recursively on top.

Large-to-small: The disks are stacked from bottom to top in order of non-increasing radius.

Max-Min: We maximize the visible boundary length of the disk with least visible boundary length, using the greedy approach presented in Section 3.

All the left-to-right methods could of course also be executed from right-to-left with different results. The stacking methods and the results pertaining to the two classed data sets can be found in the full paper.

Evaluation. To evaluate the stacking drawings we measured the visible boundary length of the top-10 of the least visible disks and we measured the total

Table 1. Results on disks of arbitrary sizes, given as *average visible boundary length of top-10 / total visible boundary length*

	City 156	City 538	Earthquake magnitude	Earthquake death count
Left-to-right by center	0.00 / 1405	0.00 / 1533	0.00 / 14446	0.00 / 4697
Left-to-right by leftmost	2.14 / 1711	0.44 / 1815	4.96 / 15061	0.66 / 8568
Large-to-small	2.72 / 1730	0.00 / 1809	0.00 / 12126	0.78 / 9049
Max-Min	4.42 / 1759	0.88 / 1868	12.45 / 16608	0.78 / 9016

boundary length that is visible. Disks that do not intersect any other disk were excluded from the top-10. For sets of disks with different sizes it makes a difference if 1 cm of the boundary of a small disk or 1 cm of the boundary of a larger disk is visible. This implies a difference in absolute visibility of a disk (in length units) and relative visibility (in percentages). We measured both absolute visibility and relative visibility. Because the comparative performance of the different methods turned out to be roughly the same in both cases, we only show results for absolute visibility and leave results for relative visibility to the full version of this paper.

Table 1 summarizes the results for the four stacking methods for the four unclassed data sets. It is clear that the Max-Min method performs best on the top-10 of least visible disks. The left-to-right by center method performs worst, except for the case where disks have roughly the same size (earthquake magnitudes), where the large-to-small method performs poorly. The same observations hold for data sets that are not shown in this paper (1260 cities, tsunami death counts (39 disks), tsunami height events (33 disks)).

Another important aspect is the visual quality of the resulting map. Since this cannot be measured, user experiments would be needed to evaluate it. In this paper, we only show a few figures for comparison purposes. Figure 10 shows the 156 largest cities of the USA with disk areas proportional to the population using the four different methods (the differences can be seen most clearly in the upper right corners of the maps). The figures correspond to the top four rows of Table 1. It is noticeable that the left-to-right methods produce maps that seem 'unbalanced" or 'asymmetric". A left-to-right structure is visible that has no cartographic meaning. This artifact can be perceived even more clearly on maps where the disk sizes vary less (not shown here).

The Max-Min method has a higher computational cost ($O(n^2 \log n)$ time) than the simple left-to-right or large-to-small methods, which require only sorting. The implementation effort is also significantly higher for the Max-Min method. However, it scores better than the other methods according to Table 1, especially for the least visible disks. Furthermore, for sets of disks with not too much difference in size, the Max-Min method is better because it does not have visual artifacts like the left-to-right methods, and it clearly outperforms the large-to-small method on visible perimeter.

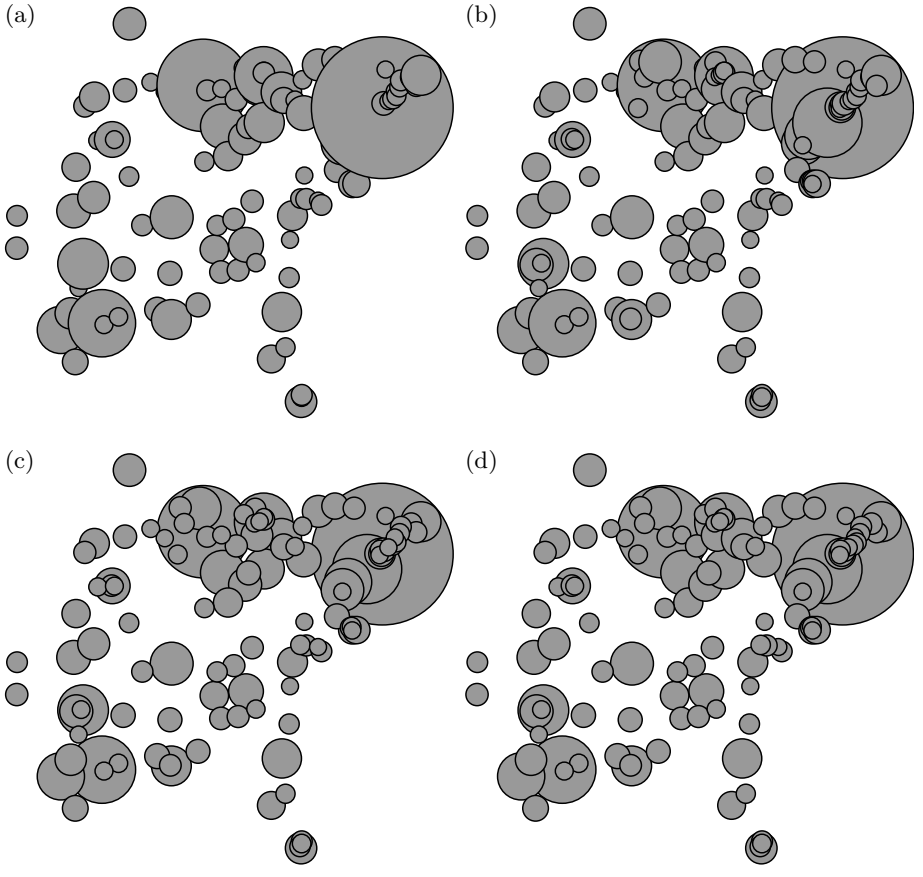


Fig. 10. USA, 156 biggest cities (only showing half of the map), stacked: (a) left-to-right by center; (b) left-to-right by leftmost; (c) large-to-small; (d) Max-Min method

5 Conclusions and Open Problems

We described an algorithm that solves the Max-Min problem for stacking drawings in $O(n^2 \log n)$ time. In our experiments, comparing this algorithm with three heuristics, we found that our method performed best on our test data. However, we did not experiment with methods that compute physically realizable drawings, and it is unclear how they would perform in comparison with our methods for stacking drawings. Solving the Max-Min problem (or the Max-Total problem) for physically realizable drawings is NP-hard, and developing good heuristics for such drawings is not trivial.

Among the open problems that remain are the computation of optimal Max-Min stacking drawings in $o(n^2 \log n)$ time, the computation of optimal Max-Total stacking drawings (or approximations thereof) in polynomial time, and the development of approximation algorithms for physically realizable drawings.

Acknowledgement. The authors thank Christian Vossers for implementing the methods and running the experiments.

References

1. P. K. Agarwal and S. Suri. Surface approximation and geometric partitions. *SIAM J. Comput.*, 27:1016–1035, 1998.
2. K. C. Clarke. *Analytical and Computer Cartography*. Prentice Hall, Englewood Cliffs, NJ, 2nd edition, 1995.
3. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, Germany, 2nd edition, 2000.
4. E. D. Demaine, J. S. B. Mitchell, and J. O'Rourke. The open problems project. Problem 33. `http://maven.smith.edu/ orourke/TOPP/P33.html`.
5. B. Dent. *Cartography - thematic map design*. McGraw-Hill, 5th edition, 1999.
6. L. Fortnow. Computational Complexity Blog. Post of Friday, February 14, 2003. `http://weblog.fortnow.com/archive/2003_02_01_archive.html`.
7. R. J. Fowler, M. S. Paterson, and S. L. Tanimoto. Optimal packing and covering in the plane are NP-complete. *Inform. Process. Lett.*, 12(3):133–137, 1981.
8. T. Griffin. The importance of visual contrast for graduated circles. *Cartography*, 19(1):21–30, 1990.
9. D. E. Knuth and A. Raghunathan. The problem of compatible representatives. *SIAM J. Discrete Math.*, 5:422–427, 1992.
10. D. Lichtenstein. Planar formulae and their uses. *SIAM J. Comput.*, 11(2):329–343, 1982.
11. NOAA Satellite and Information Service. National geophysical data center, 2005. `http://www.ngdc.noaa.gov/`.
12. Queensland University Advanced Centre for Earthquake Studies. `http://www.quakes.uq.edu.au`.
13. T. A. Slocum, R. B. McMaster, F. C. Kessler, and H. H. Howard. *Thematic Cartography and Geographic Visualization*. Prentice Hall, 2nd edition, 2003.

Does Path Cleaning Help in Dynamic All-Pairs Shortest Paths?*

C. Demetrescu[1], P. Faruolo[2], G. F. Italiano[3], and M. Thorup[4]

[1] Dip. di Informatica e Sistemistica, Univ. Roma "La Sapienza", Italy
demetres@dis.uniroma1.it
[2] Dip. di Informatica ed Applicazioni, Univ. Salerno, Italy
pomfar@dia.unisa.it
[3] Dip. di Informatica, Sistemi e Produzione, Univ. Roma "Tor Vergata", Italy
italiano@disp.uniroma2.it
[4] AT&T Labs-Research, 180 Park Avenue, Florham Park, NJ 07932
mthorup@research.att.com

Abstract. In the dynamic all-pairs shortest path problem we wish to maintain information about distances in a weighted graph subject to dynamic operations such as edge insertions, edge deletions, and edge weight updates. The most efficient algorithms for this problem maintain a suitable superset of shortest paths in the graph. This superset retains information about the history of previous graph updates so as to avoid pathological situations where algorithms are continuously forced to rebuild large portions of their data structures. On the other hand, the set of maintained paths may grow too large, resulting in both prohibitive space consumption and inefficient updates. To circumvent this problem, the algorithms perform suitable path cleaning operations. In this paper, we implement and experiment with a recent efficient algorithm by Thorup, which differs from the previous algorithms mainly in the way path cleaning is done, and we carry out a thorough experimental investigation on known implementations of dynamic shortest path algorithms. Our experimental study puts the new results into perspective with respect to previous work and gives evidence that path cleaning, although crucial for the theoretical bounds, appears to be instead of very limited impact in practice.

1 Introduction

In this paper we consider dynamic shortest path problems. In particular, our goal is to maintain information about all-pairs shortest paths (APSP) in a weighted directed graph subject to edge weight updates (which includes as a special case edge insertions and deletions). This seems an important problem on its own right, and it finds applications in many areas (see, e.g., [15]), including transportation networks, where weights are associated with traffic/distance; database

* Work supported in part by the Sixth Framework Programme of the EU under contract number 507613 (Network of Excellence EuroNGI) and under contract number 001907 (project DELIS), and by Italian MIUR under project ALGO-NEXT.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 732–743, 2006.

systems, where one is often interested in maintaining distance relationships between objects; data flow analysis and compilers; document formatting; and network routing [7, 14].

The fully dynamic APSP problem was first studied in 1967 [13], but the first algorithms that were provably faster than recomputing the solution from scratch were proposed only more than thirty years later: in 1999 King [12] presented a fully dynamic algorithm for maintaining APSP in directed graphs with positive integer weights less than C: the running time of her algorithm is $O(n^{2.5}\sqrt{C \log n})$ per update and $O(1)$ per query in a graph with n vertices. Recently, Demetrescu and Italiano [5] presented a fully dynamic shortest paths algorithm that requires $O(n^2 \cdot \log^3 n)$ amortized time per update and constant time per query. After that work, Demetrescu *et al.* [3] conducted a thorough empirical study on dynamic APSP algorithms, focusing in particular on the theoretical work in [5, 12]. This study has shown that dynamic shortest path algorithms and their techniques can be really of practical value in many situations: in practice the speed up of dynamic algorithms is much higher than the one predicted by the theoretical analysis, and they can be even several orders of magnitude faster than repeatedly computing a solution from scratch with a static algorithm. The theoretical bound of [5] has been later improved to $O(n^2(\log n + \log^2(m/n)))$ amortized time per update by Thorup [16], where m is the number of edges.

Our Results. The objective of this paper is to advance our knowledge on dynamic shortest paths algorithms by following up the recent theoretical progress of Thorup [16] with a thorough empirical study. We performed extensive tests under several variations of graph and update parameters in order to gain a deeper understanding of the experimental behavior of dynamic shortest paths algorithms. To this end, we produced a rather general framework in which the dynamic shortest paths algorithms available in the literature can be implemented and tested. Our experiments were run both on randomly generated inputs, on more structured (non–random) inputs, which tried to enforce bad update sequences on the algorithms, and on real-world inputs.

We note that the algorithm of Thorup builds on the same approach as the algorithm by Demetrescu and Italiano, i.e., by maintaining a suitably defined superset of shortest paths in the graph. The main goal of this superset is to retain information about the history of previous graph updates in order to avoid pathological situations where algorithms are continuously forced to rebuild large portions of their data structures. On the other hand, throughout the sequence of updates the set of maintained paths may grow out of hand and become too large, resulting in both prohibitive space consumption and inefficient updates. To circumvent this problem, both algorithms perform suitable path cleaning operations, although with rather different approaches. In this experimental work we try to assess the practical value of the two approaches: in particular, our experiments give evidence that path cleaning, even if crucial for the theoretical analysis, appears to be of very limited impact in practice.

Related Work. Besides the extensive computational studies on static shortest path algorithms (see, e.g., [10]), many researchers have been complementing the

wealth of theoretical results on dynamic shortest paths with empirical studies in the effort of bridging the gap between the design and theoretical analysis and the actual implementation, experimental tuning and practical performance evaluation. In particular, Frigioni *et al.* [9] proposed efficient implementations of dynamic transitive closure and shortest path algorithms, while Frigioni *et al.* [8] and later Demetrescu *et al.* [4] conducted an empirical study of dynamic single-source shortest path algorithms. Many of these shortest path implementations refer either to partially dynamic algorithms or to special classes of graphs. More recently, Buriol *et al.* [1] have conducted a thorough computational analysis to study the effects of heap size reduction techniques in dynamic single-source shortest path algorithms, and Demetrescu *et al.* [3] conducted a computational study on dynamic APSP.

2 Algorithms Under Investigation

In this section we briefly survey the two algorithms for fully dynamic APSP that will be considered in this computational study. We also discuss suitable heuristics for improving their practical performance. Our implementations work on graphs with non-negative edge weights. Unless otherwise stated, we assume that each update operation on the input graph consists of either inserting or deleting a vertex and all its incident edges. We notice that this is essentially equivalent to updating the weights of all edges with a common endpoint.

The algorithm by Demetrescu and Italiano (`DI`). The dynamic shortest path algorithm in [5] (which we refer to as `DI`) works on directed graphs with nonnegative real-valued edge weights and hinges on the notion of *locally shortest paths* (LSP): we say that a path π is locally shortest if every proper subpath of π is a shortest path (note that π is not necessarily a shortest path). Locally shortest paths can be used to generate the set of shortest paths. If we only know a subset of the shortest paths but have generated all the locally known shortest paths, then these contain the unknown shortest paths. The above leads to a quite efficient static APSP algorithm [3] and it works perfectly as a deletions-only APSP algorithm spending $O(mn + n^2 \log n)$ time in total on deleting all n vertices. A key to the efficiency is that all locally shortest paths between two vertices are internally vertex disjoint. This implies that we have at most n^2 locally shortest paths through any vertex, hence that each vertex deletion can destroy at most that many paths. However, a vertex insertion can stop many paths from being shortest by introducing new shortest paths, e.g., a single vertex insertion may introduce new shortest paths between all pairs of nodes. We call *historical* a path that stops being shortest due to a vertex insertion. More formally, a *historical path* is a path that has been a shortest path at some point during the sequence of updates, and none of its edges have been touched by an update since then. We further say that a path π in a graph is a *locally historical path* (LHP) if every proper subpath of π is a historical path.

The main idea behind the `DI` algorithm is to maintain dynamically the set of locally historical paths, which include locally shortest paths, and therefore

shortest paths, as special cases. The following theorem from [5] bounds the number of paths that become locally historical after each update:

Theorem 1. *Let G be a graph with n vertices subject to a sequence of update operations. If at any time throughout the sequence of updates there are at most h historical paths between each pair of vertices, then the amortized number of paths that become locally historical at each update is $O(hn^2)$.*

To keep changes in locally historical paths small, it is then desirable to have as few historical paths as possible throughout the sequence of updates. To do so, after a vertex has been inserted, we perform dummy updates on it at exponentially spaced updates. That is, if v is inserted at update i, then for $\ell = 1, 2, 3, \ldots$, we perform a dummy update on v in connection with update $i + 2^\ell$. The effect of a dummy update is to terminate all historical paths through v that are no longer shortest. This path cleaning technique, which we call *smoothing*, leaves at any time during a sequence of k updates at most $O(\log k)$ historical paths between each pair of vertices in the graph (see [5]).

To support a vertex insertion or deletion, the algorithm works in two phases. In case of deletion, it first removes all maintained paths that contain the deleted vertex. Then it runs a dynamic modification of Dijkstra's algorithm [6] simultaneously from all vertices: at each step a shortest path with minimum weight is extracted from a priority queue and it is combined with existing historical paths to form new locally historical paths. The update algorithm spends $O(\log n)$ time for each of the $O(hn^2)$ new locally historical paths. Since the smoothing strategy described above lets $h = O(\log n)$ and increases the length of the sequence of updates by an additional $O(\log n)$ factor, this yields $O(n^2 \log^3 n)$ amortized time per update. Even with smoothing, there can be as many as $O(mn \log n)$ locally historical paths in a graph: this implies that the space required by the algorithm is $O(mn \log n)$ in the worst case. We refer the interested reader to [5] for the low-level details of the method.

Our implementation. In this paper, we consider the implementation of `DI` developed for the experimental study in [3], which follows closely the theoretical algorithm in [5]. For the sake of simplicity, this implementation restricts update operations to change the weight of a single edge at a time, rather than all edges with a common endpoint as described in [5]. To improve the practical performance of the algorithm, we implemented a *smoothing threshold heuristic*, which consists of skipping dummy updates whenever the ratio between the number of created LHP's and deleted LHP's exceeds a certain *smoothing threshold*. In particular, when the smoothing threshold is 0, no smoothing is in place, while a smoothing threshold equal to 1 corresponds to full smoothing.

The algorithm by Thorup (`T`). Thorup's algorithm for the fully-dynamic APSP problem [16] follows a general idea of Henzinger and King [11] reducing a fully-dynamic problem to a logarithmic number of decremental problems. Recall here that the Demetrescu-Italiano approach [5] works well if only deletions are performed using locally shortest paths instead of locally historical paths.

The decremental structure we need here takes a graph where some vertices are centers. As vertices are deleted, it will only maintain locally shortest paths that use some of these centers. It will assume that someone else identifies shortest paths not running through any center. Using its locally shortest paths, the decremental structure will identify any shortest path using one of its centers. The basic theorem is that if the graph starts with n vertices and c centers, then during the course of $\Theta(c)$ vertex deletions, the total number of generated centered locally shortest paths is $\Theta(cn^2)$.

To use such a centered data structure in the fully dynamic case, we divide the updates into levels. Updates are numbered $t = 1, 2, 3, \ldots$ and the *birth date* of a vertex is the number of the update inserting it. The graph is rebuilt whenever $t \geq 2n$, with n being the current number of vertices. We then set $t = 1$ and rebuild the graph with n reinserts. Asymptotically, this does not affect our amortized time bounds.

We impose a standard binary hierarchy over the update sequence. We say that *level* I is active after update t if bit I is set in t. Here bit 0 is the least significant bit. Also, t *activates the level* of its least significant set bit, that is, if L is the least significant set bit of t, then level L is inactive before t and active after t. Also, t *deactivates* the active levels lower than L. If level I is active, we let t_I denote the update that activated level I. Note that if level $J > I$ is also active, then $t_J < t_I$. Hence, among active levels, we sometimes refer to active higher levels as *older levels*.

When we activate a level I, we construct a *level* I *graph* G_I as a copy of the current graph G. The vertices from G_I are called *level* I *vertices*. While level I is active, we do not add any vertices to G_I but if a level I vertex is deleted from G, it is also deleted from G_I. Thus G_I is a decremental dynamic graph. We destroy G_I when level I is deactivated. We will often identify an active level I with its decremental level graph G_I.

For each active level graph G_I with centers C_I, we run the above mentioned centered decremental data structure. The older level graphs, will maintain shortest paths not using C_I. For G_I, we generate shortest paths through C_I via locally shortest paths through C_I.

For the efficiency we note that G_I has at most 2^I centers and lasts for 2^I updates, so the number of locally shortest paths it generates is $O(2^I n^2)$, or $O(n^2)$ per update. Moreover, we have only $O(\log n)$ levels. It follows that the total number of paths generated is $O(n^2 \log n)$ per updates whereas the best bound for the `DI` algorithm is $O(n^2 \log^2 n)$.

A more sophisticated argument says that for sparse graphs with $m \ll n$ edges, we should in fact rebuild from scratch, resetting t, once in every $2\lceil m/n \rceil$ updates. Then the number of paths generated per update is at most $O(n^2 \log(m/n))$.

Our implementation. Since the algorithm by Thorup [16] builds on the algorithm by Demetrescu and Italiano [5], our implementation of `T` is based on the `DI` implementation described above. Differently from `DI`, `T` maintains paths on different levels, rather than on a single level. To improve the practical performance of `T`, we have implemented the following two heuristics:

• *Level shift heuristic.* One performance bottleneck of `T` is the global rebuilding done every $2n$ operations, which consists of the deletion of the entire path system followed by n insert operations. The *level shift heuristic* is based on the observation that each global rebuilding is equivalent to shifting all levels one level down, i.e., level i becomes level $(i-1)$ for each $i > 1$: by accessing levels through pointers, this operation can be simply done in $O(\log n)$ steps by just updating up to $O(\log n)$ pointers every $2n$ operations.

• *Activation delay heuristic.* Differently from `DI`, which uses smoothing operations, algorithm `T` does not have any explicit mechanism to keep the size of the path system under control: this is done implicitly via level activations and deactivations at each update. The *activation delay heuristic* simply forces the algorithm to perform activations/deactivations only every k updates, where k is a user-defined parameter. During this interval, no existing levels are deleted and no new levels are created, and thus `T` ends up performing essentially the same steps as `DI`, with all insertions centered on level 1. As we will see, this heuristic can greatly improve the performance of `T` on real-world and random instances, where the rate of path generation is substantially small in nature.

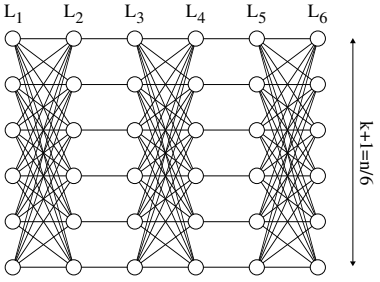
3 Experimental Setup

Test sets. Following previous computational studies on dynamic graph algorithms, in our experiments we considered three kinds of test sets with non-negative edge weights: random inputs, real-world inputs, and worst-case inputs.

• *Random inputs.* We considered random graphs with n nodes, m edges, under the $G_{n,m}$ model. Edge weights are integers chosen uniformly at random in the interval $[1, 10m]$. To generate the update sequence, we select at random one operation among edge insertion and edge deletion. If the operation is an edge insertion, we select at random a pair of nodes x, y such that edge (x, y) is not in the graph, and we insert edge (x, y) with random weight. If the operation is a deletion, we pick at random an edge in the graph, and delete it.

• *Real-world inputs.* We considered two kinds of real-world inputs: US road networks and Internet networks. The US road networks were obtained from `ftp://edcftp.cr.usgs.gov`, and consist of graphs having 148 to 952 vertices and 434 to 2,896 edges. The edge weights can be as high as 200,000, and represent physical distances. As Internet networks, we considered snapshots of the connections between Autonomous Systems (AS) taken from the University of Oregon Route Views Archive Project (`http://www.routeviews.org`). The resulting graphs (AS_500, ..., AS_3000) have 500 to 3,000 vertices and 2,406 to 13,734 edges, with edge weights as high as 20,000. The update sequences we considered in real-world graphs were random weight updates on their edges in the interval $[\min_w, \max_w]$, where $\min_w$ and $\max_w$ are the minimum and maximum edge weight in the original graph, respectively.

• *Worst-case inputs.* The worst-case inputs we considered consist of synthetic graphs and sequences of updates that force $\Theta(n^3)$ historical paths in the graph and $\Theta(n^3)$ new locally historical paths per update [5]. Graphs in the family are made of six layers $L_1 \ldots L_6$ containing $(k+1)$ vertices each as shown on the right, where $k = n/6-1$. The update sequence alternates two phases.



In the first phase, decreases are made on edges connecting L_2 to L_3 so that all shortest paths from L_1 to L_5 go through the first edge, then through the second edge, and so on. At the end of this phase, there are $\Theta(n)$ historical paths in the graph between each pair of vertices in L_1 and L_5. In the second phase, increases are made on edges connecting L_4 to L_5 so that all shortest paths from L_2 to L_6 go through the first edge, then through the second edge, and so on. During this phase, for each pair of vertices in L_1 and L_6, $\Theta(n)$ paths connecting them become locally historical, leading to a total of $\Theta(n^3)$ new locally historical per update. The update sequence can be made arbitrarily long by repeating the two phases back and forth many times. In our tests, the two phases are repeated 20 times, yielding sequences of $40 \cdot k$ updates.

Performance measures. To evaluate the performance of the algorithms considered in this computational study, we consider five measures:

• *Path cleaning time:* average time in milliseconds spent to perform path cleaning during each operation. For `DI`, this is the time required to perform dummy updates, while for `T` this is the time required to activate/deactivate levels;

• *Update time:* average time in milliseconds spent to update the data structures during each operation; this time does not include the path cleaning time;

• *Time per operation:* average time in milliseconds per operation, including both path cleaning and update time;

• *Space:* maximum amount of memory allocated by the algorithms during a sequence of updates;

• *Number of generated paths:* average number of paths generated during an operation.

Times were measured using the standard system call `getrusage`. Path cleaning times and update times were measured separately by properly instrumenting the code of `DI` and `T`.

Computing platform. We have conducted our experiments on a PC equipped with a 2.4 GHz *Intel Xeon Dual-Processor*, 2 GB of physical RAM, 8 KB L1-cache, and 256 KB L2-cache running Linux Kernel 2.4.21. All our implementations are coded in `C` using the *Leonardo Library* [2], and are homogeneously written by the same people, i.e., with the same algorithmic and programming skills. We compiled our codes with `gcc 3.4.6` using the standard optimization flag (`-O3`).

Code availability. The experimental package used in this computational study, including the source code of our implementations, the input generators, the real-world test beds, the scripts used for running the experiments, and additional performance charts are publicly available at the URL `http://polluce.dia.unisa.it/dsp/`.

4 Implementation Tuning

Before comparing the two algorithms, we analyze them separately, and we discuss how different settings of their parameters may affect their performance. Our goal is to try to identify the best parameter setting for the different classes of inputs considered in this study. As we will see, the performance is mostly dependent on the path cleaning rate. In particular, we will study how the degree of smoothing for `DI` and the activation delay for `T` affect their performance on different input families. Our experiments showed a substantial difference between worst-case inputs on one side, and real-world and random inputs on the other side. For worst-case inputs, which exhibit a very high path generation rate, path cleaning is very important in order to keep the size of the data structures and the operation time small. On the other hand, path cleaning seems to be largely useless in real-world and random inputs.

Algorithm `DI`. We have considered four variants of `DI` with decreasing degree of smoothing: `DI1` (full smoothing), `DI0.7` (70% smoothing), `DI0.3` (30% smoothing), and `DI0` (no smoothing). We conducted many experiments to investigate the impact of path cleaning on the running time of those variants. Our experiments show that, on random and real-world inputs, path cleaning appears to be just an overhead and does not contribute substantially to reducing the number of generated paths. In contrast, path cleaning becomes very important on worst-case inputs: in this case, reducing the degree of smoothing causes a substantial increase of the path generation rate, and consequently a clear degradation in the update times. This is not surprising as the worst-case inputs are especially designed to create many locally historical paths at each operation. An example of this behavior is illustrated in the left hand side of Figure 1, which shows the time per operation required by the four variants of algorithm `DI` with different degrees of smoothing on three different graphs: a dense random graph with 500 vertices and 62500 edges (top), a sparse graph with 490 vertices and 1388 edges representing the Maryland (MD) road network (middle), and a worst-case graph with $k + 1 = 70$, i.e., 420 vertices and 14840 edges (bottom). The charts show both the path cleaning time (light grey bars), and the update time (dark grey bars).

Algorithm `T`. We have first observed experimentally that the *level shift heuristic* consistently improves the running time of the `T` algorithm: an example of this is illustrated in Figure 2 (a), where we consider random, real-world and worst-case inputs. For this reason, in the remainder of this paper we assume that `T` includes the *level shift heuristic*. To study how the *activation delay heuristic* affects the

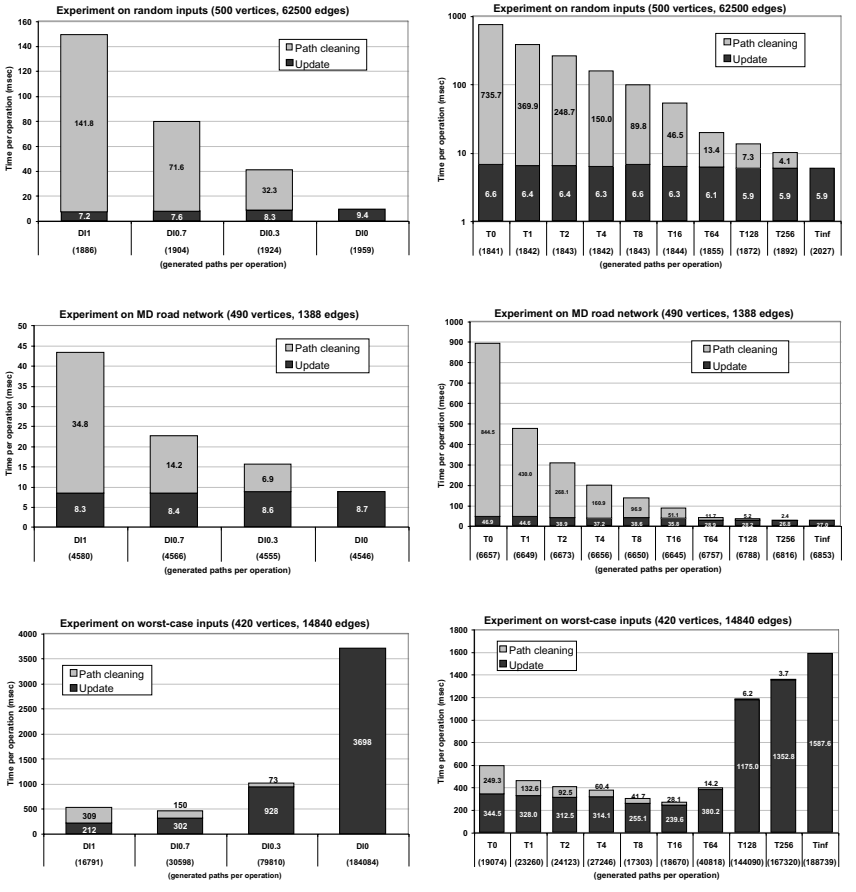


Fig. 1. Experiments on a random graph (top), Maryland road network (middle), and a worst-case graph (bottom), aimed at studying the influence of path cleaning on the time per operation of `DI` and `T`

time per operation, we have considered ten variants of algorithm `T` with delay values: 0, 1, 2, 4, 8, 16, 64, 128, 256, and ∞. We denote the corresponding variants of `T` by `T0`, `T1`, `T2`, `T4`, `T8`, `T16`, `T64`, `T128`, `T256`, and `T`∞, respectively. We note that `T0` is just `T` without the activation delay heuristic. On the other side, `T`∞ performs no activation/deactivation, making the path system collapse into the single level 1. On the right hand side of Figure 1, we show the running time of the different variants of `T` on the same inputs as in the experiments reported for `DI`. The charts show that `T` has a similar trend as `DI`, i.e., path cleaning appears mostly as an overhead for random and real-world inputs: however, the effect here is greatly amplified due to the more complex path system structure of the algorithm. Indeed, avoiding path cleaning may result in speedups up to a factor of 120 for the random graphs considered, and up to a factor of 30 for the Maryland road network. On worst-case inputs, our experiment revealed instead

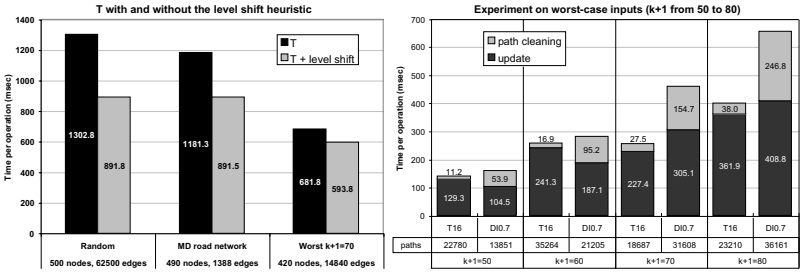


Fig. 2. (a) Performance improvement resulting from the application of the *level shift heuristic* to algorithm `T`. (b) Overall comparison of `T` and `DI` on worst-case graphs with $k + 1 = 50, 60, 70, 80$.

a clear trade-off between path cleaning time and update time. Increasing the values of the activation delay reduces the path cleaning rate, causing the number of generated paths to grow, which in turn produces an increase of the update times. However, the interaction between the two effects this time is much more complicated: as we reduce the rate of path cleaning, we pay less overhead but we incur in a steeper performance degradation. As a result, in this case the fastest algorithm is not `T0`, as one may expect for this class of inputs. While the best tradeoff appears to be at `T16` for $k + 1 = 70$, we observed that the optimal activation delay is likely to be a non-decreasing function of the graph size for our worst-case inputs. We remark that this phenomenon is neither well captured nor predicted by the theoretical analysis.

5 Overall Experimental Evaluation

The parameter tuning experiments of Section 4 suggested that path cleaning can be an unnecessary overhead when dealing with input instances that do not suffer from pathological patterns. However, a robust implementation should be able to deal with heterogeneous and impredictable structural configurations, and thus should be able to cope efficiently with worst-case inputs that force the algorithms to generate many paths in their data structures. The most robust variants of `DI` and `T` for the graph sizes considered in this study seem to be `DI0.7` and `T16`. In this section, we therefore study the relative performance of these two implementations on different synthetic and real-world input families.

Worst-case inputs. Figure 2 (b) shows the relative performance of `DI0.7` and `T16` on our worst-case graphs. Notice that `T16` is consistently faster. We recall that `DI` and `T` build their path system by finding at each step a path that shares all edges but the endpoints with two other paths already in the system. The better behavior of `T` can be explained by observing that it generates paths by combining only paths on a subset of the levels in the hierarchical path decomposition, rather than all paths in the path system as `DI` does. For inputs that tend to generate many paths, this can substantially reduce the update time.

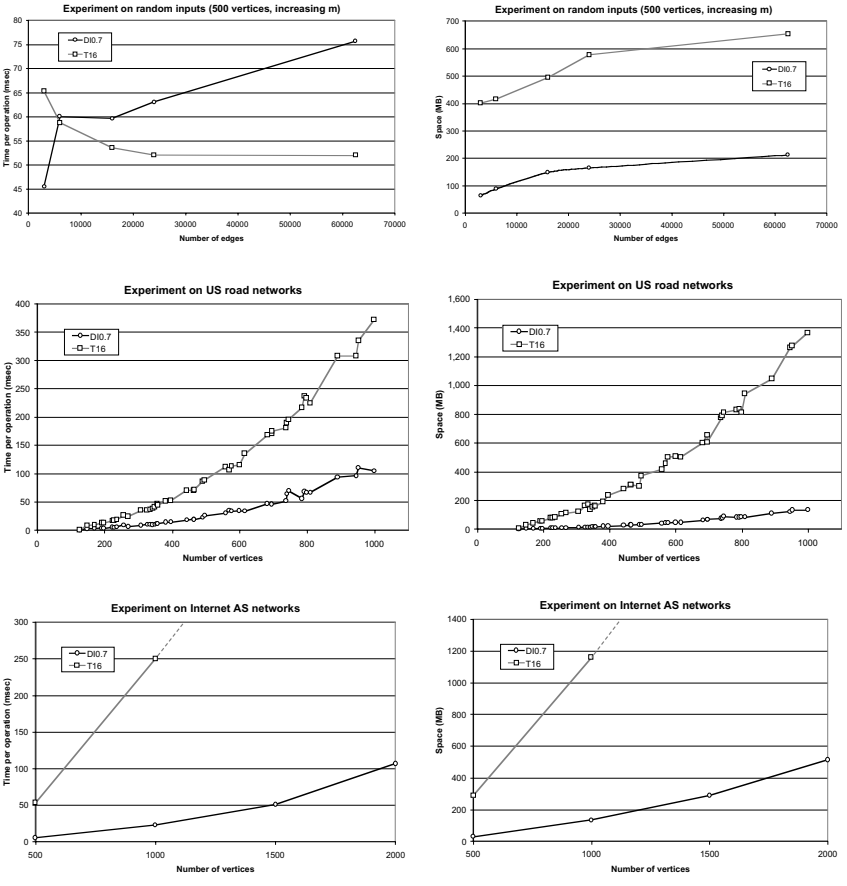


Fig. 3. Overall experimental evaluation of `DI` and `T` on random graphs (top), road networks (middle), and Internet AS networks (bottom)

Random inputs. The top chart of Figure 3 shows the time and space requirements of `T16` and `DI0.7` on an evenly mixed sequence of 2000 random insertions and deletions on random graphs with 500 vertices and number of edges increasing from 3000 to 62500. We notice that the relative performance of the two algorithms depends on the graph density: on sparse random graphs, `DI0.7` outperforms `T16`. On the other hand, `T16` becomes faster than `DI0.7` as the edge density grows. This can be explained by the fact that dense graphs tend to generate many locally historical paths, and thus we can expect similar results as in the case of worst-case inputs. Notice that the better time behavior of `T16` is paid in terms of a substantially higher space demand for maintaining a multi-level path system.

Real-world inputs. The mid and bottom charts of Figure 3 show the time and space requirements of `T16` and `DI0.7` on a sequence of 1000 random edge weight updates on US road networks and AS networks. Since our real-world graphs are

very sparse, the path generation rate is small, making DI0.7 faster than T16 as in the case of sparse random graphs discussed above. While the maximum size of our road graphs is 1000 vertices, the size of AS networks approaches 3000 vertices. We remark that on the largest AS instances the space demands of T exceeded the available internal memory, thus incurring into space swap problems that prevented us from completing the experiments.

References

1. L. Buriol, M. Resende, and M. Thorup. Speeding up dynamic shortest path algorithms. Technical report, AT&T Labs Research Report TD5RJ8B, 2003.
2. C. Demetrescu, S. Emiliozzi, I. Finocchi, and A. Ribichini. The Leonardo Library. http://www.leonardo-vm.org.
3. C. Demetrescu, S. Emiliozzi, and G. F. Italiano. Experimental analysis of dynamic all pairs shortest path algorithms. In *Proc. 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'04)*, pp. 362–371, 2004.
4. C. Demetrescu, D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Maintaining shortest paths in digraphs with arbitrary arc weights: An experimental study. In *Proc. 4th Workshop on Algorithm Engineering (WAE'00)*, 2000.
5. C. Demetrescu and G. F. Italiano. A new approach to dynamic all pairs shortest paths. *J. ACM*, 51(6):968–992, 2004. Preliminary version in STOC'03.
6. E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
7. B. Fortz and M. Thorup. Internet traffic engineering by optimizing OSPF weights. In *Proc. 19th IEEE INFOCOM*, pp. 519–528, 2000.
8. D. Frigioni, M. Ioffreda, U. Nanni, and G. Pasqualone. Analysis of dynamic algorithms for the single source shortest path problem. *ACM Journal on Experimental Algorithmics*, 3, 1998.
9. D. Frigioni, T. Miller, U. Nanni, G. Pasqualone, G. Shaefer, and C. D. Zaroliagis. An experimental study of dynamic algorithms for directed graphs. In *Proc. European Symposium on Algorithms (ESA'98), LNCS 1461*, pages 320–331, 1998.
10. A.V. Goldberg. Shortest path algorithms: Engineering aspects. In *Proc. 12th Int. Symposium on Algorithms and Computation (ISAAC'01), LNCS 2223*, 2001.
11. M. R. Henzinger and V. King. Maintaining minimum spanning forests in dynamic graphs. *SIAM J. Computing*, 31(2):364–374, 2001.
12. 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. 81–99, 1999.
13. P. Loubal. A network evaluation procedure. *Highway Research Record 205*, pages 96–109, 1967.
14. P. Narvaez, K. Y. Siu, and H. Y. Tzeng. New dynamic SPT algorithm based on a ball-and-string model. *IEEE/ACM Transactions on Networking*, 9:706–718, 2001.
15. G. Ramalingam. Bounded incremental computation. In *LNCS 1089*, 1996.
16. M. Thorup. Fully-dynamic all-pairs shortest paths: Faster and allowing negative cycles. In *Proc. 9th Scandinavian Workshop on Algorithm Theory (SWAT'04)*, pp. 384–396, 2004.

Multiline Addressing by Network Flow

Friedrich Eisenbrand[1], Andreas Karrenbauer[2],
Martin Skutella[3], and Chihao Xu[4]

[1] Fachbereich Informatik, Universität Dortmund
friedrich.eisenbrand@cs.uni-dortmund.de
[2] Max-Planck-Institut für Informatik, Saarbrücken
karrenba@mpi-inf.mpg.de
[3] Fachbereich Mathematik, Universität Dortmund
martin.skutella@uni-dortmund.de
[4] Lehrstuhl für Mikroelektronik, Universität Saarbrücken
xu@ee.uni-saarland.de

Abstract. We consider an optimization problem arising in the design of controllers for OLED displays. Our objective is to minimize amplitude of the electrical current through the diodes which has a direct impact on the lifetime of such a display. Modeling the problem in mathematical terms yields a class of network flow problems where we group the arcs and pay in each group only for the arc carrying the maximum flow. We develop (fully) combinatorial approximation heuristics suitable for being implemented in the hardware of a control device that drives an OLED display.

1 Introduction

Organic Light Emitting Diode (OLED) displays are considered as the displays of the future. The image and video displayed is brilliant, has a very high contrast and a viewing angle of nearly 180 degrees. It reacts within 10 microseconds which is much faster than the eye can catch and is therefore perfect for video applications. The display is flexible and above all can be produced at low cost. One major reason why there are only small-size displays on the market is the insufficient lifetime of state of the art OLED displays.

What causes this short lifetime? Briefly, this is because of the high electrical currents through the diodes that occur with the traditional addressing techniques. Though it seems that every pixel shines continuously with a certain brightness, the images are displayed row-after-row. This works at a sufficiently high frame rate since the perception of the eye is the average intensity emitted by each diode. The problem is that this row-by-row activation scheme causes long idle times of the diodes and extreme stress when they are activated.

To overcome this problem one considers now to activate two, or more consecutive rows simultaneously [1]. For passive matrix displays rows can only be simultaneously displayed if their content is equal.

Therefore the goal is to decompose an image into several images, the overlay (addition) of which is equal to the original image. Fig. 1 shows such a decomposition. The first image (single) is traditionally displayed row-after-row. In the

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 744–755, 2006.

Fig. 1. Decomposition of an image with $k = 2$ such that every two rows of the double parts have the same content and the sum of all row maxima is minimized

other two images (double) every two rows have the same content so that one can display these rows simultaneously. As one can see in Fig. 1, the images on the right hand side are much darker than the original one. The decomposition should be in such a way that the amplitudes of the electrical current which are needed to display the picture are as small as possible.

In this paper we develop an algorithm to tackle this optimization problem. Our objective is to come up with a combinatorial approximation algorithm that will be implemented in hardware to actually drive such an OLED display. This imposes some restrictions on the methods and techniques we shall use. First of all, such an algorithm has to compute a feasible solution in realtime, i.e. below the perception of a human eye. Moreover, it should be implemented on a chip of low cost meaning that we are not able to e.g. use a general purpose LP solver or a general purpose CPU with an IEEE floating point unit. Therefore, we look for algorithms that are sufficiently simple and easy to implement. Moreover, the algorithms should not suffer from numerical instabilities. Also exact rational arithmetic is not an option for such a realtime application. We rather want to use only fixed precision numbertypes, i.e. integers of fixed size. Hence, we aim at a fully combinatorial algorithm using only addition, subtraction, and comparison.

Contributions of This Paper

First, we model this optimization problem as a certain network-flow problem where the arcs are partitioned into groups and only the arc with the highest flow in each group is charged. This model yields an equivalent formulation as a covering (integer) linear program with an exponential number of constraints. We develop linear time separation routines which are required to solve the problem exactly with the ellipsoid method [2, 3] or with a cutting-plane approach [4] or to solve it approximately [5] with known frameworks. We then propose an efficient fully combinatorial heuristic which is based on the separation of these constraints and satisfies the above requirements. Our implementation shows that this heuristic is very close to the optimum.

2 Technical Background

To understand the objective of our optimization problem, we need to explain in an informal way how OLED displays work. An OLED display has a matrix structure with n rows and m columns. At any crossover between a row and a column there is a vertical diode which works as a pixel.

The image itself is given as an integral $n \times m$ matrix $(r_{ij}) \in \{0, \ldots, \varrho\}^{n \times m}$ representing its RGB values. The number ϱ determines the *color depth*, e.g. $\varrho = 255$ for 16.7 million colors. Since there are only $n + m$ contacts available, a specific addressing technique is needed. We explain one technique (*pulse width modulation*) in a simplified way in the following. Consider the contacts for the rows and columns as switches. If the switch of row i and column j is closed, the pixel (i, j) shines with a brightness or *intensity* I which is common to all row-column pairs. An image has to be displayed within a certain time frame T_f. The value r_{ij} determines that within the time-frame T_f, the switches i and j have to be simultaneously closed for the time

$$t_{ij} = r_{ij} \cdot \frac{T_f}{I} \tag{1}$$

in total. At a sufficient high frame rate e.g. 50 Hz, the perception by the eye is the average value of the light.

Currently, drivers for OLED displays display the image in a row-by-row fashion. This means that the switches for the rows are activated one after the other. While row i is active, column j has to be active for the time t_{ij} so row i is finished after $\max\{t_{ij} \mid j = 1, \ldots, m\}$ time units. The time which is required to display the image is consequently $T(I) = \sum_{i=1}^{n} \max\{t_{ij} \mid j = 1, \ldots, m\}$. The intensity I has to be high enough such that $T(I) \leq T_f$ holds.

Equation (1) shows that the time t_{ij} is anti-proportional to the value I. The aforementioned short lifetime of todays OLED displays is mainly due to the high value of I which is necessary to display images with a sufficient frame rate. This means that the peak energy which has to be emitted by the diodes of the display is very high while on the other hand, the diodes stay idle most of the time, see [6]. The high amplitudes of electrical current which alternate with long idle times put the diodes under a lot of stress, which results in a short lifetime.

In this paper, we aim to overcome this problem by a different driving mechanism. The simple but crucial observation is that, if two rows have the same content, we could drive them simultaneously and thereby we would save half of the time necessary for the two rows. Therefore the value of I could be reduced until $T(I) = T_f$.

$$\begin{pmatrix} 109 & 238 & 28 \\ 112 & 237 & 28 \\ 150 & 234 & 25 \\ 189 & 232 & 22 \\ 227 & 229 & 19 \end{pmatrix} = \begin{pmatrix} 0 & 82 & 25 \\ 0 & 82 & 25 \\ 0 & 41 & 22 \\ 0 & 41 & 22 \\ 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 \\ 112 & 155 & 3 \\ 112 & 155 & 3 \\ 189 & 191 & 0 \\ 189 & 191 & 0 \end{pmatrix} + \begin{pmatrix} 109 & 156 & 3 \\ 0 & 0 & 0 \\ 38 & 38 & 0 \\ 0 & 0 & 0 \\ 38 & 38 & 19 \end{pmatrix}$$

Fig. 2. An example decomposition

Consider the example in Fig. 2. Suppose here that initially $T_f/I = 1$. If the image is displayed row-by-row, then the minimum time which is needed to display the image is $238 + 237 + 234 + 232 + 229 = 1170$ time units. But we can do better by a suitable decomposition of the image into three matrices. In

the first one every even row is equal to its odd predecessor and in the second one every odd row is equal to its even predecessor with zero-rows, where there is no predecessor available respectively. The remainder is put into an offset matrix that is driven in the traditional way. By driving the equal rows simultaneously, we require only $82+41+155+191+156+38+38=701$ time units. This means that we could reduce I and therefore the amplitude of the electrical current by roughly 40%.

We could save even more by driving $3, 4, 5\ldots$ rows simultaneously. Of course there is a saturation somewhere, unless the image is totally homogeneous. On our benchmark pictures, we observed that it is not worth to consider more than 6 simultaneously driven rows.

3 The Network Model

For the sake of simplicity, we first consider the case in which two consecutive rows can be activated simultaneously. Let $R = (r_{ij}) \in \{0, \ldots, \varrho\}^{n\times m}$ be the matrix representing the picture. To decompose R we need to find matrices $F^{(1)} = (f_{ij}^{(1)})$ and $F^{(2)} = (f_{ij}^{(2)})$ where $F^{(1)}$ represents the offset part and $F^{(2)}$ the common part. More precisely, the i-th row of matrix $F^{(2)}$ represents the common part of rows i and $i+1$. In order to get a valid decomposition of R, the matrices $F^{(1)}$ and $F^{(2)}$ must fulfill the constraint $f_{ij}^{(1)} + f_{i-1,j}^{(2)} + f_{ij}^{(2)} = r_{ij}$ for $i = 1, \ldots, n$ and $j = 1, \ldots, m$, where we now and in the following use the convention to simply omit terms with indices running out of bounds. The fixed boundary conditions in our application require $f_{1j}^{(1)} + f_{1j}^{(2)} = r_{1j}$ and $f_{nj}^{(1)} + f_{n-1,j}^{(2)} = r_{nj}$ for the first and the last row, respectively. Notice that the matrix $F^{(2)}$ has only $n-1$ rows. We sometimes assume that there is in addition a row numbered n containing only zeros.

Since we cannot produce "negative" light we require also non-negativity of the variables $f_{ij}^{(\ell)} \geq 0$ where we now and in the following use the superscript $\ell = 1, 2$ for statements that hold for both matrices. The goal is to find an integral decomposition that minimizes

$$\sum_{i=1}^{n} \max\{f_{ij}^{(1)} : 1 \leq j \leq m\} + \sum_{i=1}^{n-1} \max\{f_{ij}^{(2)} : 1 \leq j \leq m\} \ .$$

This problem can be formulated as an integer linear program by replacing the objective by $\sum_{i=1}^{n} u_i^{(1)} + \sum_{i=1}^{n-1} u_i^{(2)}$ and by adding the constraints $f_{ij}^{(\ell)} \leq u_i^{(\ell)}$. This yields

$$\begin{aligned} \min \quad & \sum_{i=1}^{n} u_i^{(1)} + \sum_{i=1}^{n-1} u_i^{(2)} & & \\ \text{s.t.} \quad & f_{ij}^{(1)} + f_{i-1,j}^{(2)} + f_{ij}^{(2)} = r_{ij} & \text{for all } i,j & \qquad (2) \\ & f_{ij}^{(\ell)} \leq u_i^{(\ell)} & \text{for all } i,j,\ell & \\ & f_{ij}^{(\ell)} \in \mathbb{Z}_{\geq 0} & \text{for all } i,j,\ell & \end{aligned}$$

The corresponding linear programming relaxation is not integral in general as an example with

$$R = \begin{pmatrix} 1\,0\,1 \\ 1\,1\,1 \\ 0\,1\,1 \end{pmatrix}$$

shows. The optimal solution is obtained by setting each $u_i^{(\ell)} = \frac{1}{2}$ yielding an objective value of $\frac{5}{2}$.

Observe that the constraints (2) can be represented by a blockdiagonal 0/1-matrix with one block for each $j = 1, \ldots, m$. We thus have m blocks with identical structure of the form illustrated on the left of equation (3).

$$\begin{pmatrix} 1\,1\,0\,0\,0\,0\,0\,0\,0 \\ 0\,1\,1\,1\,0\,0\,0\,0\,0 \\ 0\,0\,0\,1\,1\,1\,0\,0\,0 \\ 0\,0\,0\,0\,0\,1\,1\,1\,0 \\ 0\,0\,0\,0\,0\,0\,0\,1\,1 \\ 0\,0\,0\,0\,0\,0\,0\,0\,0 \end{pmatrix} \rightsquigarrow \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & -1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & -1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 \end{pmatrix} \tag{3}$$

Notice that the columns have the consecutive ones property. Hence, there is a natural transformation by row-operations (preserving the solution space like in Gaussian elimination) into a node-arc incidence matrix, see, e.g. [7]. In other words, we add a zero dummy row at the end and subtract from each row its predecessor and obtain in each column exactly one 1 and one -1 as depicted on the right in equation (3). Recall that this matrix is just the block for one $j \in \{1, \ldots, m\}$. The resulting graph $G = (V, A)$, which is common to all j, is called the *displaygraph*; see Fig. 3 for an illustration. The displaygraph has node set $V = \{1, \ldots, n+1\}$ and arcs $(i, i+1)$ for $i = 1, \ldots, n$ and $(i, i+2)$ for $i = 1, \ldots, n-1$.

In the forthcoming, we refer to the u variables as *capacities*. The new right-hand sides of the equality constraints which we call *demands* are given by $d_j(i) = r_{ij} - r_{i-1,j}$. The generalization when we drive $k \geq 2$ consecutive lines together is straightforward and depicted in Fig. 3.

The optimization problem can now be understood as follows. Assign integral capacities $u : A \to \mathbb{Z}_{\geq 0}$ to the arcs of the displaygraph at minimum cost (each unit of capacity costs 1 for each arc) such that each network flow problem defined by the displaygraph together with the demands $d_j : V \to \mathbb{Z}$ has a feasible solution for each $j = 1, \ldots, m$. Let $\delta^{out}(X)$ denote the outgoing arcs of node set $X \subset V$. We use the standard notation $u(\delta^{out}(X))$ and $d(X)$ for the sums over the corresponding capacities and demands respectively. It follows now from MaxFlow/MinCut duality that our optimization problem can be rewritten as (cf. chapter 11 in [8])

$$\begin{aligned} \min \quad & \sum_{a \in A} u(a) \\ \text{s.t.} \quad & u(\delta^{out}(X)) \geq d_j(X) \qquad \text{for all } X \subset V, \text{ for all } j \\ & u \in {\mathbb{Z}_{\geq 0}}^A \end{aligned} \tag{4}$$

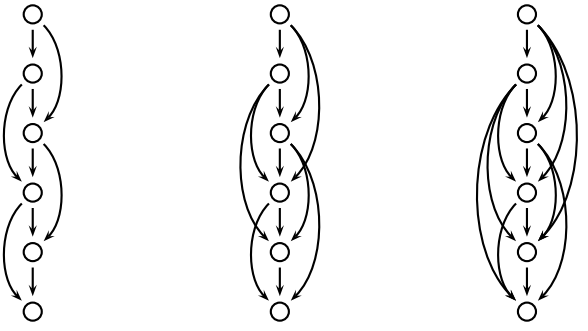


Fig. 3. Examples for a displaygraph with $n = 5$, $k = 2, 3, 4$

For general graphs, this problem contains DIRECTEDSTEINERTREE as a special case. It remains NP-complete for *directed acyclic graphs* by a reduction to VERTEXCOVER. Note that there is a trivial k-approximation with respect to the displaygraph when we just use the singleline variables.

Since the number of cuts in a graph is exponential in its size, we have to look for an efficient way to solve the separation problem.

4 Efficient Algorithms for the Separation Problem

Finding violated inequalities for a given assignment to the variables is a key idea for solving linear programs. It is well known [9] that the *linear optimization* problem over a given polyhedron is polynomial time equivalent to the *separation* problem for this polyhedron. Also our heuristics (see Sec. 5) rely on the solution of the separation problem for inequalities (4) and updating the solution iteratively until we have found a feasible solution. In our setting, the separation problem is the following:

> Given a capacity assignment $u \geq 0$ of the displaygraph, determine whether u is feasible and if not, compute a subset $X \subset V$ of the nodes in the displaygraph such that there is a $j \in \{1, \ldots, m\}$ with $u(\delta^{out}(X)) < d_j(X)$.

4.1 Separation by MAXFLOW/MINCUT

Observe that we can consider the separation problem for each column $j \in \{1, \ldots, m\}$ independently. For a given j this can be done with a MAXFLOW computation as follows. We construct a network G_j by adding new vertices s and t to the displaygraph. The capacities of the arcs in the displaygraph are given by u. There is an arc (s, i) if $d_j(i) > 0$ with capacity $d_j(i)$ and there is an arc (i, t) if $d_j(i) < 0$ with capacity $-d_j(i)$.

If the maximum s, t-flow in this network is less than $\delta_j = \sum_{d_j(i)>0} d_j(i)$, then the vertices of a corresponding MINCUT which belong to the displaygraph

comprise a set $X \subset V$ with $u(\delta^{out}(X)) < d_j(X)$. If the value of a MaxFlow in G_j is equal to δ_j for all $j = 1, \ldots, m$, then the capacity assignment u is feasible.

In our implementation we iteratively increase the capacity of one arc by some integral constant c. We use a *Blocking Flow* approach in our implementation. Thereby, we can benefit from an efficient treatment of capacity adjacent problems, see, e.g. [7]. Note that in this case there only exist at most c augmenting paths and moreover these paths have to take the adjacent arc. Since each of these paths can be found in linear time by depth-first search, the update takes only $O(n)$ for one column. In practice, the performance is even better since on average the paths are rather short. Moreover, we have to consider only the columns that have this arc in their current MinCut since otherwise the additional capacity would not have any effect.

Theorem 1. *Given two capacity adjacent assignments $0 \leq u \leq \bar{u}$, i.e. u and $\bar{u}$ differ in exactly one arc, and suppose that one is given a maximum flow f w.r.t. u, then the separation problem can be solved in linear time. More precisely one can compute a maximum flow $\bar{f} \leq \bar{u}$ and a minimum cut $\bar{X} \subset V$ with respect to $\bar{u}$ in linear time.*

One issue of this approach is that we maintain the flow variables f. On ordinary PC hardware this is an advantage since we have to compute the decomposition of the image which is represented by the flow variables at the end anyways. But we need roughly k times more memory than the input size which makes the implementation on a chip more expensive. We address this issue in the next section.

4.2 A Linear Time Algorithm for Fixed k

The number of simultaneously activated lines k is relatively small, in fact up to 6. We now show how to solve the separation problem for fixed k in linear time. The key feature of the displaygraph which allows such an efficient algorithm is the following. The arcs are of the form (i, i'), where $i' \leq i + k$. It is sufficient to find for each $j = 1, \ldots, m$ a subset $X \subset V$ such that $u(\delta^{out}(X)) - d_j(X)$ is as small as possible. If one of these values is negative, we have found a violated inequality. Otherwise, all constraints are fulfilled.

In order to find such a subset X, we partition the vertices $V = \{1, \ldots, n+1\}$ into consecutive blocks $B_1, \ldots, B_{\lceil (n+1)/k \rceil}$ of size k. That is, $B_i := \{(i-1) \cdot k + 1, (i-1) \cdot k + 2, \ldots, i \cdot k\}$ for $i = 1, \ldots, \lfloor (n+1)/k \rfloor$ and $B_{\lceil (n+1)/k \rceil} := \{\lfloor (n+1)/k \rfloor + 1, \ldots, n+1\}$ if k does not divide $n+1$.

We now consider a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ where the vertex set $\mathcal{V}$ contains all subsets of the sets B_i and there is an arc (S_1, S_2) if there exists an index i such that $S_1 \subseteq B_i$ and $S_2 \subseteq B_{i+1}$. Furthermore $\mathcal{G}$ has an additional vertex s and an additional vertex t together with arcs (s, S) for each $S \subseteq B_1$ and (S, t) for each $S \subseteq B_{\lceil (n+1)/k \rceil}$. A path from s to t specifies a subset $X \subseteq V$ of the displaygraph in a natural way (and also vice versa): Given a path from s to t, take the union of the subsets represented by the inner vertices of the path.

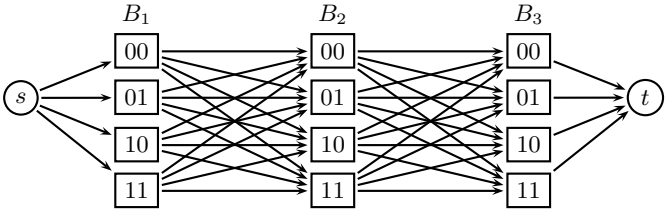


Fig. 4. Solving the separation problem by a shortest path computation in a DAG (here $k = 2$)

It remains to define arc weights in $\mathcal{G}$ such that the weight of such a path is exactly $u(\delta^{out}(X)) - d_j(X)$. For this consider an arc (S_1, S_2) with head and tail different from s and t respectively. The weight of this arc is defined as $-\sum_{i \in S_2} d_j(i) + u(S_1 : S_2)$, where $S_1 : S_2$ denotes the subset of arcs of the displaygraph which run from S_1 to S_2. The weight of an arc (s, S) is defined as $-\sum_{i \in S} d_j(i)$ and the weight of an arc (S, t) is zero.

In this way, the weight of a path from s to t in $\mathcal{G}$ is equal to the value $u(\delta^{out}(X)) - d_j(X)$, where X is the set which is represented by the path. Thus, the separation problem can be reduced to m shortest path problems in graph $\mathcal{G}$ with roughly $2^k \cdot (n + 1)/k = O(n)$ vertices, if k is fixed. We have proved the following theorem.

Theorem 2. *The separation problem for the inequalities* (4) *can be solved in linear time for fixed* k*.*

5 Implementation

In practice we have a display of fixed size and the requirement on the running time to be below the perception of a human eye. According to this time constraint, we have to design our algorithm such that the cost of the integrated circuit which implements it is minimized. Parallelization, e.g. on several columns concurrently, and reusing results of certain computations, e.g. shortest path trees of previous iterations, decrease the running time but increase the complexity and memory usage of the circuit and hence the production costs.

Since we use only fixed precision datatypes, we consider the variables to be integral all the time. A higher intermediate precision is easily achieved by scaling the constraints, i.e. the demands. Briefly speaking our heuristics work as follows. We start with an initial assignment of the capacities returned by the function Initialize. We will discuss it later. Consider it to simply return the zero vector for now. Afterwards, we iterate until we find a feasible solution. In each iteration, we first solve the separation problem. Since we have to compute the lifting to the flow-variables at the end we use the blocking flow approach for this task. Depending on the outcome and on the chosen strategy, we augment one or several capacities. In Fig. 5, we describe this general framework of our heuristics with pseudo-code.

```
Input: d = (d_1, ..., d_m)
Output: f
f = 0
u = INITIALIZE(d)
while(f is not feasible){
    f = MAXFLOW(V, A, d, u)
    C = MINCUT(V, A, d, u)
    for(k = 1, ..., p){
        u(k) = u(k) + Δu(k, C)
    }
}
return f
```

Fig. 5. Framework for the approximation heuristics

In short, d denotes the demands of the nodes and thereby implicitly defines the underlying graph as we consider k to be fixed. The notion of the function Δu covers several variants of our heuristics for augmenting the capacities.

5.1 Capacity Augmentation

After having found a violated cut, the question arises which capacity variables to augment. Unlike in the framework of Garg and Könemann [5] where all capacities of the cut would be multiplied with a constant, we select only one variable to augment since we want to benefit from the capacity adjacency mentioned before.

Our strategy is to augment the most prospective variable that is in any of the cuts of the different columns. We measure the potential impact of a capacity by the number of different columns, i.e. cuts, it appears in. This would be equivalent to summing up all the cuts over all columns which gives us a valid violated inequality too. The *basic greedy* approach selects the capacity having the highest potential impact, i.e. to increase the capacity with the highest coefficient in the sum of the cuts. A slightly different variant of this approach takes only the variables into account which appear in the cut of a column with the maximum deficit (referred to as *max-column greedy update*), i.e. that attains the minimum in the separation problem.

Since we maintain the integrality of the variables throughout the algorithm, we have to increase the variables at least by 1 in each iteration. With this value, the running time is then proportional to the size of the display and the difference between the achieved objective value and the one from the initial solution. By adding a greater constant or a certain fraction of the previous capacity like in the framework of Garg and Könemann, the running time would improve, however we observed that the quality of our approximation deteriorated.

5.2 Initialization

The naive way to initialize the capacities is to set them to the zero vector. The other extremal case would be to solve the LP and round up each frac-

tional capacity with the result that we have just one iteration where we compute the flow variables. Though, we are within an additive error not greater than the number of variables then, we could round down instead of rounding up and solve or approximate the remaining 0/1 integer program with a possibly better solution for the whole problem. However, as mentioned before solving (or approximating) linear programs involving fractional numbers is not really what we want. It is natural to ask whether there is a way inbetween that allows us to stick with integers and can be attacked with a fully combinatorial algorithm.

Indeed, if we restrict ourselves to *easy* constraints, i.e. constraints describing an integral polyhedron, we accomplish the first goal of remaining integral. If we only consider constraints permitting fully combinatorial algorithms (e.g. flow problems or their duals) then we could also achieve the second goal.

Having solved such an easy subproblem, we can initialize our heuristics with the thereby computed solution. The next theorem, which we prove in the full version of this paper, identifies such an easy subset of the constraints.

Theorem 3. *The linear program* (4) *restricted to the setsystem*

$$\mathcal{C} = \bigcup_{i=1}^{n} \{X_i, Y_i, X_i \cap Y_i, X_i \cup Y_i : X_i = \{1, \ldots, i\}, Y_i = \{i, i+2, i+3, \ldots, n+1\}\}$$

has an integral optimal solution for $k = 2$ which can be found in $O(n)$ time by a fully combinatorial algorithm.

The proof of this theorem is based on the observation that, after a suitable reordering of the variables, the constraint matrix has the consecutive ones property and that the optimization problem can be solved via a shortest path computation in a directed acyclic graph. This consecutive ones property does not hold for $k > 2$. However, we can restore this property by adding the missing capacities on the left-hand-side of the inequalities and the corresponding lower bounds on the right-hand-side giving (weaker) valid inequalities yielding an approximate solution.

6 Computational Results

As of yet the computational results are based on ordinary PC hardware. Therefore, we only present the running times of the variant that performs best on a Pentium M with 2GHz and 2MB L2 cache.

As a testset we used the portraits of 197 employees of the MPI. While the original images have a resolution of 180×240, we scaled them down to $n = 60, 90, 120, 150$ keeping the aspect ratio such that we have $m = 4n$ for all images. We observed that, among the different initialization and augmentation strategies, a combination of the aforementioned max-column greedy update and the initialization using the easy constraints performs best. We recommend these strategies on the basis of an implementation on PC-hardware.

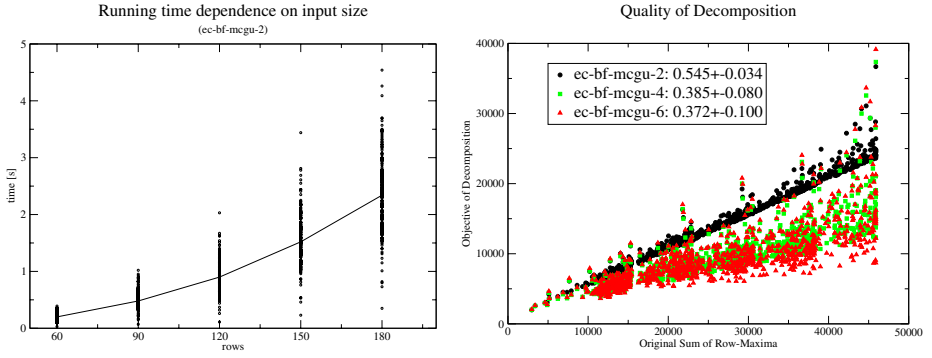


Fig. 6. We initialize the capacities by the easy constraints (ec), used a blocking flow algorithm (bf), performed the capacity augmentation by the maximum column greedy update method (mcgu), and $k = 2, 4, 6$. The left plot shows the dependence of the running time on n in case of $k = 2$. The inset in the right graph shows the average ratios and their standard deviations for $k = 2, 4, 6$ respectively.

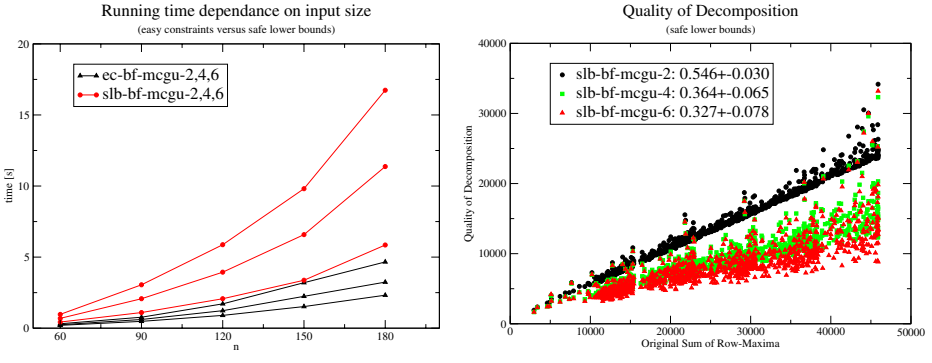


Fig. 7. Using safe lower bounds for initialization yields better ratios (right) but worser running times (left)

On the left of Fig. 6, one can see the dependence of the running time on the input size together with the distribution of the instances. We connected the median running time to guide the eye. The fit of these medians with respect to a power function yields $t = 21\mu s \cdot n^{2.23}$ which is almost linear in the number of pixels. The graph on the right of Fig. 6 has on its x-axis the initial intensity I and on the y-axis the reduced intensity I' which we achieve with multiline addressing using our algorithm. The black dots are the results obtained by addressing two rows simultaneously, i.e. for $k = 2$. Here one can see that the average ratio I'/I is roughly 0.545 with a standard deviation of 0.034. The green squares are the results for $k = 4$. Here the average ratio I'/I is 0.385 ± 0.065. The red triangles represent $k = 6$ with $I'/I = 0.372 \pm 0.100$. The theoretical lower bound for these ratios being $1/k$.

Not using the easy constraints but safe lower bounds yields slightly better ratios as depicted in the inset of the right graph of Fig. 7. But the running time grows faster with the number of pixels as one can see in the left graph of Fig. 7 where the three top curves belong to the median runtimes using save lower bounds with $k = 2, 4, 6$ respectively. Whereas the median runtimes using the easy constraint initialization and $k = 2, 4, 6$ yield the three lower curves.

References

1. Xu, C., Wahl, J., Eisenbrand, F., Karrenbauer, A., Soh, K.M., Hitzelberger, C.: Verfahren zur Ansteuerung von Matrixanzeigen. Patent 10 2005 063 159, Germany, pending (2005)
2. Grötschel, M., Lovász, L., Schrijver, A.: Geometric Algorithms and Combinatorial Optimization. Volume 2 of Algorithms and Combinatorics. (1988)
3. Khachiyan, L.: A polynomial algorithm in linear programming. Doklady Akademii Nauk SSSR **244** (1979) 1093–1097
4. Dantzig, G., Fulkerson, R., Johnson, S.: Solution of a large-scale traveling-salesman problem. J. Operations Res. Soc. Amer. **2** (1954) 393–410
5. Garg, N., Könemann, J.: Faster and simpler algorithms for multicommodity flow and other fractional packing problems. FOCS (1998) 300 – 309
6. Murano, S., Burghart, M., Birnstock, J., Wellmann, P., Vehse, M., Werner, A., Canzler, T., Stübinger, T., He, G., Pfeiffer, M., Boerner, H.: Highly efficient white OLEDs for lighting applications. SPIE 2005, San Diego (2005)
7. Ahuja, R.K., Magnanti, T.L., Orlin, J.B.: Network flows. Prentice Hall Inc., Englewood Cliffs, NJ (1993) Theory, algorithms, and applications.
8. Schrijver, A.: Combinatorial Optimization - Polyhedra and Efficiency. Volume 24. Springer Verlag (2003)
9. Grötschel, M., Lovász, L., Schrijver, A.: The ellipsoid method and its consequences in combinatorial optimization. Combinatorica **1**(2) (1981) 169–197

The Engineering of a Compression Boosting Library: Theory vs Practice in BWT Compression

Paolo Ferragina[1,⋆], Raffaele Giancarlo[2,⋆⋆], and Giovanni Manzini[3,⋆⋆⋆]

[1] Dipartimento di Informatica, Università di Pisa, Italy
ferragina@di.unipi.it
[2] Dipartimento di Matematica ed Applicazioni, Università di Palermo, Italy
raffaele@math.unipa.it
[3] Dipartimento di Informatica, Università del Piemonte Orientale, Italy
manzini@mfn.unipmn.it

Abstract. Data Compression is one of the most challenging arenas both for algorithm design and engineering. This is particularly true for Burrows and Wheeler Compression a technique that is important in itself and for the design of compressed indexes. There has been considerable debate on how to design and engineer compression algorithms based on the BWT paradigm. In particular, Move-to-Front Encoding is generally believed to be an "inefficient" part of the Burrows-Wheeler compression process. However, only recently two theoretically superior alternatives to Move-to-Front have been proposed, namely Compression Boosting and Wavelet Trees. The main contribution of this paper is to provide the first experimental comparison of these three techniques, giving a much needed methodological contribution to the current debate. We do so by providing a carefully engineered compression boosting library that can be used, on the one hand, to investigate the myriad new compression algorithms that can be based on boosting, and on the other hand, to make the first experimental assessment of how Move-to-Front behaves with respect to its recently proposed competitors. The main conclusion is that Boosting, Wavelet Trees and Move-to-Front yield quite close compression performance. Finally, our extensive experimental study of boosting technique brings to light a new fact overlooked in 10 years of experiments in the area: a fast adapting order-zero compressor is enough to provide state of the art BWT compression by simply compressing the run length encoded transform. In other words, Move-to-Front, Wavelet Trees, and Boosters can all be by-passed by a fast learner.

⋆ Partially supported by Italian MIUR Italy-Israel FIRB Project "Pattern Discovery Algorithms in Discrete Structures, with Applications to Bioinformatics"; and by Yahoo! Research grant on "Data compression and indexing in hierarchical memories".

⋆⋆ Partially supported by Italian MIUR grants PRIN "Metodi Combinatori ed Algoritmici per la Scoperta di Patterns in Biosequenze" and FIRB "Bioinformatica per la Genomica e La Proteomica" and Italy-Israel FIRB Project "Pattern Discovery Algorithms in Discrete Structures, with Applications to Bioinformatics".

⋆⋆⋆ Partially supported by Italian MIUR Italy-Israel FIRB Project "Pattern Discovery Algorithms in Discrete Structures, with Applications to Bioinformatics".

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 756–767, 2006.

1 Introduction

In the quest for the ultimate data compressor, Algorithmic Theory and Engineering go hand in hand. This point is well illustrated by the amount of results and implementations originated by the fundamental results by Lempel and Ziv. A more recent example is provided by the fundamental contributions given by Burrows and Wheeler to data compression [3], via their transform (denoted for short bwt). In their seminal paper Burrows and Wheeler proposed to compress the output of the bwt using Move-to-Front Encoding (shortly mtf), followed by an order zero compressor (usually Arithmetic or Huffman coding). As pointed out by Fenwick [5] in the first systematic study of that new type of compression, the technique is so powerful that it yields nearly state-of-the-art compression results without any particularly sophisticated engineering of the coding step. This should be contrasted with PPM-based compressors that involve quite a bit of engineering. From that point on, the research on bwt compression has focused on two aspects: faster bwt computation, and the identification and exploitation of potential inefficiencies in the use of mtf. While substantial progress has been made on the first point, both theoretically and experimentally (e.g. [2, 17]), the second point experienced a plethora of heuristically-designed proposals (see [1, 4] and references therein) which improved over the original proposal but often lacked of analytical justification.

Recently, two theoretical results [7, 8] have shed new light on the role of mtf within the bwt-based compression paradigm, paving the way to the (*analytically justified*) design of more powerful bwt-based compressors. In particular, [8] proposed a new technique, named *compression boosting*, that fully uses the power of bwt to show that the performance of *any* order zero compressor can be automatically, and optimally, boosted to higher order entropy compression. On the other hand, [7] proved that combining the bwt with the Wavelet Tree data structure [10] we can achieve high-order entropy bounds without using mtf or the boosting technique. At the same time, a novel and very recent analysis of classic bwt compression [12] showed that mtf may not be as inefficient as initially thought. Summing this with the fact that the theoretical results in [7, 8] require some sophisticated algorithmic machinery, it is not at all clear how much computational/compression *gain* can be achieved by shaving off the mtf-step from the bwt-based compressors.

The above is the main question addressed in the present paper, whose key contribution is first of all *methodological*. We provide the first carefully engineered compression boosting library that can be used, on the one hand, to investigate the myriad new compression algorithms that can be based on boosting, and on the other hand, to make the first experimental assessment of how mtf behaves with respect to its recently proposed competitors: Boosting and Wavelet Trees. The boosting library is available under the GPL license at the page http://www.mfn.unipmn.it/~manzini/boosting and it is highly modular in the sense that it can be used to create a powerful high order compressor even without any knowledge of the bwt.

In order to highlight our additional technical contributions, we need to recall a few facts about compression boosting [8]. Additional details are given in Section 3. The boosting technique builds upon three main ingredients: bwt, the Suffix Tree data structure, and a greedy algorithm to process them. Specifically, it is shown that there exists a proper partition of the bwt of a string s exhibiting a deep combinatorial relation with the k-th order entropy of s. That partition can be identified via a greedy processing of the suffix tree of s. The final compressed string is then obtained by compressing individually each substring of the partition by means of the base (order zero) compressor A we wish to boost. The proper design of a compression booster is a bit trickier than it sounds:

(**A**) The greedy algorithm alluded to before is a bottom up visit of the suffix tree. In practice, on large files, the memory requirements for the construction of the suffix tree would be prohibitively large. We use suffix arrays instead and procedures that efficiently simulate the bottom up visit of the suffix tree [13].

(**B**) Given the algorithm A we wish to boost, we also need an objective function that estimates how well A compresses a given string. In [8], the objective function is given in terms of two parameters λ and μ, and the order zero empirical entropy of the string (see Section 3 for details). In practice, λ and μ may either be not available or be too conservative. This point is discussed in Section 4, where we propose two cost models and the relative objective functions.

(**C**) Another important aspect of the boosting process is the ability of the algorithm A to quickly adapt to the statistics of a string to be compressed. Faster adaptation means better compression. This learning process is usually governed by parameters establishing how fast A "forgets the past". We limit our experimentation to range coding and arithmetic coding. The somewhat intuitive, yet surprising, results are reported in Section 5 and outlined in point (**F**) below.

Using our library we have compared the performance of the compression booster against bwt compressors based on mtf (e.g. Bzip2 [19] and variants), bwt compressors based on Wavelet Trees (e.g., Wzip [9]), and state-of-the-art PPM compressors (e.g. PPMd [21]). We show that:

(**D**) As predicted by Theory [8], boosting is superior to classic bwt approaches that use mtf in terms of compression ratio but not by much. It is also slower, as it is to be expected, because of the significant time cost for building the optimal bwt-partition (as observed in **B**). Therefore, those results give a strong indication that mtf may actually be a time-efficient way to effectively "approximate" the optimal partition computed by the boosting technique.

(**E**) As predicted by Theory [7, 10, 11], the simple combination of bwt with Wavelet Trees is effective both in time and compression ratio, and does not benefit from the use of the booster. However, the Wavelet Tree approach is outperformed by classic bwt approaches that use mtf. This further confirms the effectiveness in time and compression ratio of mtf, and leaves open the problem of investigating the more powerful approach proposed in [7], namely *Generalized Wavelet Trees*, which are based on sophisticated combinations of binary (like,

Run Length encoders) versus non-binary (like, Huffman or Arithmetic encoders) compressors and Wavelet Trees of properly-designed shapes.

(**F**) The experiments performed to estimate the best adaptation parameters for range and arithmetic coding show clearly that a fast adaptation yields state-of-the-art compression by simply compressing a run length encoded bwt. This is somewhat intuitive, yet surprising: to our knowledge no one observed experimentally the superiority of this strategy w.r.t. mtf, and no theoretical analysis has explained or suggested such behavior. Moreover, this result comes from the stronger finding that for a fast adapting range coder the optimal partition coming out of the booster is the bwt itself (data not shown, due to space limitations). That is, the strategy is optimal with respect to the boosting paradigm.

(**G**) All the bwt-based compressors we tested were inferior, in terms of compression ratio, to the highly engineered PPMd tool. The principle behind bwt and PPM techniques is the same: discover and encode according to the "best" contexts. However, bwt-based algorithms have the advantage of knowing the entire string, while PPMd "discovers" good contexts on-line. Yet bwt-based algorithms do not perform as well. This yields an extremely intriguing engineering problem for data compression practitioners. Note that there is a very good reason to stick with bwt-based compressors instead of embracing the, apparently superior, PPM-based compressors: the reason is that bwt-based compressors are a key tool for the construction of compressed indices which (informally) are compressed files offering the additional capability of very fast full-text search (see [18] for formal definitions and a comprehensive survey).

In conclusion our experiments show that Boosting, Wavelet Trees and mtf yield quite close compression performance. However, the boosting technique appears to be more robust and works well even with less effective order zero compressors (such as Huffman coding). Moreover, when used with range/arithmetic coding the boosting technique yields excellent compression somewhat irrespective of how fast the order-zero compressor adapts to the statistics of the string. These positive features are achieved using more resources (time and space) during compression: nevertheless our results show that a careful implementation of boosting can handle efficiently even very large files.

2 Background and Notation

Let s be a string over the alphabet $\Sigma = \{a_1, \ldots, a_h\}$ and, for each $a_i \in \Sigma$, let n_i be the number of occurrences of a_i in s. The 0-*th order empirical entropy* of the string s is defined as[1] $H_0(s) = -\sum_{i=1}^{h} (n_i/|s|) \log(n_i/|s|)$. It is well known that H_0 is the maximum compression we can achieve using a fixed codeword for each alphabet symbol. We can achieve a greater compression if the codeword we use for each symbol depends on the k symbols preceding it, since the maximum compression is now bounded by the k-th order entropy $H_k(s)$ (see [15] for the formal definition). For highly compressible strings, $|s|\, H_k(s)$ fails to provide a

[1] We assume that all logarithms are taken to the base 2 and $0 \log 0 = 0$.

```
$ mississipp i
i $mississip p
i ppi$missis s
i ssippi$mis s
i ssissippi$ m
m ississippi $
p i$mississi p
p pi$mississ i
s ippi$missi s
s issippi$mi s
s sippi$miss i
s sissippi$m i
```

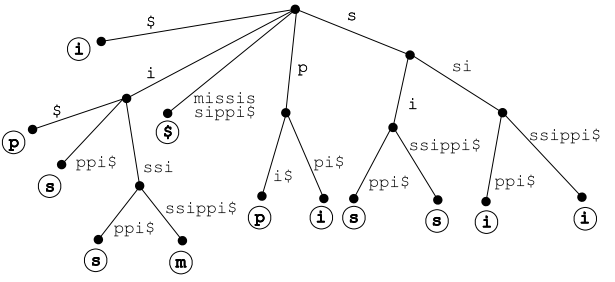


Fig. 1. The bwt matrix (left) and the suffix tree (right) for the string $s =$ `mississippi$`. Note that the output of the bwt is the last column of the bwt matrix, i.e., `ipssm$pissii`.

reasonable bound to the performance of compression algorithms (see discussion in [8, 15]). For that reason, [15] introduced the notion of 0-*th order modified empirical* entropy $H_0^*(s)$ which has the property that if $|s| > 0$, $|s|H_0^*(s)$ is at least equal to the number of bits needed to write down the length of s in binary. The *k-th order modified empirical entropy* H_k^* is then defined in terms of H_0^* as the maximum compression we can achieve by looking at *no more than* k symbols preceding the one to be compressed.

Given a string s, the Burrows-Wheeler transform (bwt for short) consists of three basic steps: (1) append to the end of s a special symbol \$ smaller than any other symbol in Σ; (2) form a *conceptual* matrix $\mathcal{M}$ whose rows are the cyclic shifts of the string s\$, sorted in lexicographic order; (3) construct the transformed text $\hat{s} = \mathtt{bwt}(s)$ by taking the last column of $\mathcal{M}$ (see Fig. 1). Although it is not obvious, from $\hat{s}$ we can always recover s, see [3] for details. The power of the bwt rests on the fact that equal contexts (substrings) of s are grouped together resulting in a few clusters of distinct symbols in $\mathtt{bwt}(s)$. That clustering makes $\mathtt{bwt}(s)$ a better string to compress than s. In their seminal paper Burrows and Wheeler proposed to compress the output of the bwt using Move-to-Front Encoding[2] (shortly mtf), followed by an order zero compressor (Arithmetic or Huffman coding). In [12] it is shown that if we use an order zero compressor A such that for any string x we have $|\mathtt{A}(x)| \leq |x|H_0(x) + c|x|$, then the bwt followed by mtf, followed by A produces an output bounded by

$$\mu|s|H_k(s) + (\log \zeta(\mu) + c)|s| + \log|s| + \mu g_k \tag{1}$$

where ζ is the Riemann zeta function. The bound (1) holds for any $k \geq 0$ and $\mu > 1$. In [15] it is shown that if we use Run Length Encoding (shortly rle) between mtf and the order zero compressor, the output is bounded by

$$(5 + \epsilon)|s|H_k^*(s) + \log_2|s| + g'_k \tag{2}$$

[2] Move-to-Front transforms the input encoding each symbol with the number of distinct symbols seen since its last occurrence, see [3] for details.

for any $k \geq 0$ and $\epsilon \approx 10^{-2}$. The bottom line is that combining the Burrows-Wheeler transform with mtf and an order zero compressor we can achieve the k-th order entropy, H_k or H_k^*, simultaneously for any $k \geq 0$. Note however, that the coefficient in front of the k-th order entropy in (1) and (2) is greater than 1 whereas we are assuming that A achieves H_0 without any multiplicative constant. This means that there is a small inefficiency as we go from H_0 and H_0^* to H_k and H_k^*. It is an open question whether this inefficiency can be removed with a more detailed analysis or is inherent in the use of Move-to-Front encoding.

3 A BWT-Based Compression Booster

Recently [8] has described a bwt-based compression booster that, starting from an order zero compressor, achieves the k-th order entropy without the inefficiency found in the mtf-based approach. In this section we quickly review how the boosting algorithm works; the details and proofs can be found in [8].

A crucial ingredient of the compression booster is the relationship between the bwt matrix and the suffix tree data structure. Let $\mathcal{T}$ denote the suffix tree of the string $s\$$. $\mathcal{T}$ has $|s|+1$ leaves, one per suffix of $s\$$, and edges labeled with substrings of $s\$$ (see Figure 1). Any node u of $\mathcal{T}$ has *implicitly associated* a substring of $s\$$, given by the concatenation of the edge labels on the downward path from the root of $\mathcal{T}$ to u. In that implicit association, the leaves of $\mathcal{T}$ correspond to the suffixes of $s\$$. We assume that the suffix tree edges are sorted lexicographically. Since each row of the bwt matrix is prefixed by one suffix of $s\$$ and rows are lexicographically sorted, the i-th leaf (counting from the left) of the suffix tree corresponds to the i-th row of the bwt matrix. We associate the i-th leaf of $\mathcal{T}$ with the i-th symbol of the string $\hat{s} = \mathtt{bwt}(s)$. The symbol associated to the leaf v is thus the symbol preceding in s the substring of $s\$$ associated with v. Such symbols are represented inside circles in Fig. 1. If we write $\hat{\ell}_i$ to denote the symbol associated to the i-th leaf, from the above discussion, it follows that $\hat{s} = \hat{\ell}_1\hat{\ell}_2\cdots\hat{\ell}_{|s|+1}$ (see Fig. 1 for an example).

For any suffix tree node u, let $\hat{s}\langle u\rangle$ denote the substring of $\hat{s}$ obtained concatenating, from left to right, the symbols associated to the leaves descending from node u. We say that a subset $\mathcal{L}$ of $\mathcal{T}$'s nodes is a *leaf cover* if every leaf of the suffix tree has a *unique* ancestor in $\mathcal{L}$. Any leaf cover $\mathcal{L} = \{u_1, \ldots, u_p\}$ naturally induces a partition of the leaves of $\mathcal{T}$ namely $\hat{s}\langle u_1\rangle, \ldots, \hat{s}\langle u_p\rangle$. Because of the relationship between $\mathcal{T}$ and the bwt matrix this is also a partition of $\hat{s}$.

Let C denote a function which associates to every string x over $\Sigma \cup \{\$\}$ the positive real value $C(x)$. For any leaf cover $\mathcal{L}$, we define its cost as: $C(\mathcal{L}) = \sum_{u\in\mathcal{L}} C(\hat{s}\langle u\rangle)$. In [8] it is shown a linear time greedy algorithm that computes a leaf cover $\mathcal{L}_{\min}$ of minimum cost. That is, $\mathcal{L}_{\min}$ is such that $C(\mathcal{L}_{\min}) \leq C(\mathcal{L})$, for any leaf cover $\mathcal{L}$. $\mathcal{L}_{\min}$ is called an *optimal leaf cover* and we say that $\mathcal{L}_{\min}$ induces an *optimal partition* of $\hat{s}$ with respect to the cost function C. The relevance of $\mathcal{L}_{\min}$ for achieving the k-th order entropy derives by the following Theorem [8].

Theorem 1. *Let* A *denote an order zero compressor such that for any string* x $|\mathtt{A}(x)| \leq \lambda|x|\ H_0^*(x) + \mu$ *where* λ *and* μ *are constants. Let* $\mathcal{L}_{\min}$ *denote an*

optimal partition of $\hat{s}$ with respect to $C(x) = \lambda|x|\, H_0^*(x) + \mu$. If we use algorithm `A` *to compress the substrings of the optimal partition induced by $\mathcal{L}_{\min}$, the overall output size is bounded by $\lambda|s|\, H_k^*(s) + g_k$ bits for any $k \geq 0$, where g_k only depends on the alphabet size $|\Sigma|$. A similar result holds for H_k as well.* □

4 The Compression Boosting Library

The efficient implementation of the compression booster algorithm is a non trivial engineering task. The main challenge is avoiding the explicit construction of the suffix tree which would require an unpractically large amount of working memory. We now detail our implementation discussing its space requirements in the "real world" model where we assume that every character takes one byte and every integer takes 4 bytes. Let $n = |s|$. We first compute the suffix array of s using the `ds` algorithm [17] that has a peak memory usage of only $5.03n$ bytes: n bytes for the text, $4n$ for the suffix array, and $0.03n$ working space.

Given the suffix array we compute and store $\hat{s} = \texttt{bwt}(s)$ using n bytes. The greedy algorithm computing the optimal partition of $\hat{s}$ consists of a properly defined post-order visit of the suffix tree of s. To avoid the explicit construction of the suffix tree we use the technique from [13] that allows one to emulate the post-order visit of the suffix tree using the Longest Common Prefix (shortly LCP) array. Thus, we use the `Lcp6` algorithm from [16] for computing in $O(n)$ time the LCP array given s, $\hat{s}$, and the suffix array. This algorithm overwrites the LCP array over the suffix array and has a peak space usage of $(6 + \delta)n$ bytes. The parameter δ is at most 4 and is bounded also by $|\Sigma|^k/n + 2H_k(s)$ for any $k \geq 0$. This means that the space usage is smaller for highly compressible inputs.

Having computed the LCP array we can discard the input string s; thus at this stage we are only storing $\hat{s}$ and the LCP array for a total space usage of $5n$ bytes. The computation of the optimal partition using the technique in [13] reduces to a left to right scan of the LCP array. This allows us to store the endpoints of intervals of the optimal partition in the same memory used for the LCP array (that is, overwriting the LCP array). Thus the only additional memory used during the "emulated" suffix tree visit is the space used to store the stack of the suffix tree nodes whose visit has started but not yet finished. This space could be $\Theta(n)$ in the worst case, but in practice is much smaller than n bytes overall.

Cost models. An important issue in the implementation of the compression booster is the choice of the parameters λ and μ in the cost function $C(x) = \lambda|x|\, H_0^*(x) + \mu$ of Theorem 1. Given a compressor `A`, theory dictates that λ and μ be chosen so that $|\texttt{A}(x)| \leq C(x)$ for any string x. However, if we strictly enforce this condition it is possible that for many strings x we have $|\texttt{A}(x)| \ll C(x)$. Since the optimal partitioning is computed minimizing $C(\mathcal{L}_{\min})$, if $C(x)$ is "too far" from $|\texttt{A}(x)|$ we could end up with a partition which does not exploit the full potential of the compressor `A`. To evaluate this phenomenon our boosting library supports two different cost models. In addition to the "entropy bound" model

outlined above, we provide a "real cost" model in which the optimal partition is computed with respect to the cost $C(x) = |\mathtt{A}(x)|$. Using the "real cost" model we get the best possible compression that we can achieve using the compressor `A`. The drawback of this model is that the computation of the optimal partition no longer takes linear time. The time cost might be quadratic in the worst case, although the experimental results show that the overall running time usually increases only by a factor 1.5.

User interface. Our library provides a simple interface to boost the performance of an arbitrary compressor using either mtf or the optimal partitioning strategy outlined in Sect. 3. This can be done even without any knowledge of the Burrows-Wheeler transform! The user simply needs to provide compression and decompression procedures and, for the computation of the optimal partition, a procedure evaluating the cost function $C(x)$ (see [6] for details).

5 Experimental Results

Using the boosting library described in the previous section we have implemented several bwt-based compressors. By means of extensive experiments we tried to assess to what extent mtf and the boosting algorithm are able to turn a generic order zero compressor into a state of the art compressor. We ran all experiments on a 2.6 GHz Pentium 4 CPU with 1.5 GB of main memory running Fedora Linux. All code was written in C and compiled using gcc Ver. 3.2.2. As a testbed we used the collection of files introduced in [17] for testing suffix array construction algorithms.

The following are the algorithms tested in our experiments.

Bzip2 is the well known tool based on the bwt developed by Julian Seward [19]. Bzip2 splits the input file into blocks of size 900Kb and computes the bwt followed by mtf on each block. The actual compression is done using rle0[3] followed by Multiple-Table Huffman coding [22].

MtfRleMth executes the same steps as Bzip2 operating on the whole input instead that on fixed length blocks.

MtfRleRc. The earliest versions of Bzip2 used arithmetic coding instead of multiple-table Huffman. Recently, range coding has been (re)discovered as a patent-free alternative to arithmetic coding. Range coding and arithmetic coding are based on similar concepts and achieve similar compression. MtfRleRc compresses the bwt using mtf followed by rle0, followed by range coding (we used the code from [14]). Note that MtfRleRc is identical to MtfRleMth except that, instead of Multiple-table Huffman coding, it uses range coding.

RleRc compresses the bwt using rle followed by range coding.

[3] We use rle to denote the run length encoding of the runs of any character, while we use rle0 to denote the run length encoding only of the runs of zeros. If a string was produced by mtf, rle0 is the natural choice because of the massive presence of 0-runs as observed by Fenwick [5].

BoostRleRc is the boosting algorithm applied to the compressor consisting of rle followed by range coding. Note that the difference in compression between RleRc and BoostRleRc gives the "added value" of the use of the booster.

MtfRleAc, RleAc, BoostRleAc are analogous respectively to MtfRleRc, RleRc, BoostRleRc except that they use the arithmetic coding routines from [23] instead of range coding.

MtfRleHuff, RleHuff, BoostRleHuff are analogous respectively to MtfRleRc, RleRc, BoostRleRc except that they use Huffman coding instead of range coding. Note that MtfRleHuff differs from MtfRleMth in that the former uses a single Huffman table whereas the latter uses up to six tables for the same file.

Wavelet. This algorithm computes the bwt of the whole input and compresses the resulting string using a wavelet tree [10]. The importance of wavelet trees stems from the fact that they have been used for the design of efficient bwt-based compressed indices [18] and that they also achieve the k-th order entropy for any $k \geq 0$. More precisely, from [7] follows that for a string s over the alphabet Σ the output size of Wavelet is bounded by $4|s|\, H_k^*(s) + 6|\Sigma|^{k+1} \log(|s|)$ bits.

BoostWav is an implementation of the boosting algorithm applied to the wavelet tree encoder using the "real cost" model. Thus the difference between Wavelet and BoostWav is that the former builds one wavelet tree on the whole bwt, whereas the latter finds an optimal partition of the bwt and builds one wavelet tree on each substring of the optimal partition. Again, the difference in compression between Wavelet and BoostWav is the "added value" of the booster.

PPMd is an implementation of the ppm encoder by Dmitry Shkarin [21, 20] which is the current state of the art for PPM compression. In our tests we used PPMd at its maximum strength, that is using a model of order 16 and 256Mb of working memory.

Range/arithmetic coding variants. The behavior of range and arithmetic coding depends on two parameters: MaxFreq and Increment. The ratio between these two values essentially controls how quickly the coding "adapts" to the new statistics. For range coding we set MaxFreq = 65536 (the largest possible value) and we experimented with three different values of Increment. Setting Increment = 256 we get a range coder with FAST adaptation, with Increment = 32 we get a range coder with MEDIUM adaptation, and finally setting Increment = 8 we get a range coder with SLOW adaptation. For arithmetic coding we set MaxFreq = 16383 (the largest possible value) and Increment = 64 obtaining therefore a FAST adaptation.

Compression ratio. Figure 2 reports the average compression ratio (in bits per symbol) and average (de)compression time (microseconds per symbol) for all the algorithms mentioned above. Looking at the average compression ratio we can see that both mtf and the boosting algorithm do a good job in transforming an order zero compressor into a state-of-the-art compressor. However, our data show some unexpected behaviors. Considering the three version of range coding (with FAST, MEDIUM, and SLOW adaptation) we see that mtf achieves the best compression using MEDIUM adaptation whereas the boosting algorithm "prefers"

	averg	ctime	dtime
Bzip2	1.424	0.53	0.14
MtfRleMth	1.167	0.96	0.46
RleAc FAST	1.126	0.97	0.59
MtfRleAc FAST	1.158	0.94	0.53
BoostRleAc FAST RC	1.125	7.43	0.59
RleHuff	1.596	0.89	0.47
MtfRleHuff	1.230	0.95	0.46
BoostRleHuff RC	1.195	5.04	0.45
BoostRleHuff EB	1.229	2.96	0.45
Wavelet	1.230	0.96	1.01
BoostWav RC	1.229	3.55	0.94
PPMd	1.080	0.60	0.66

	averg	ctime	dtime
RleRc FAST	1.129	0.90	0.48
MtfRleRc FAST	1.161	0.90	0.48
BoostRleRc FAST RC	1.129	4.11	0.48
BoostRleRc FAST EB	1.134	3.04	0.48
RleRc MED.	1.171	0.89	0.48
MtfRleRc MED.	1.153	0.96	0.48
BoostRleRc MED. RC	1.152	4.13	0.49
BoostRleRc MED. EB	1.158	3.02	0.48
RleRc SLOW	1.245	0.90	0.48
MtfRleRc SLOW	1.164	0.90	0.48
BoostRleRc SLOW RC	1.175	4.12	0.48
BoostRleRc SLOW EB	1.194	3.02	0.48

Fig. 2. Experimental results for the collection of files introduced in [17]. For each algorithm we report the average compression in bits per symbol and the average compression and decompression time in microseconds per symbol. The RC and EB acronyms indicate the cost model ("real cost" or "entropy bound") used by the booster.

	running time					peak memory	
	bwt	lcp	visit	cmpr	total	lcp	visit
sprot	0.70	0.60	1.67	0.11	3.08	7.01	5.00
rfc	0.60	0.51	2.35	0.11	3.57	6.86	5.00
howto	0.50	0.45	2.83	0.15	3.93	7.29	5.01
reut	1.24	0.55	1.92	0.08	3.79	6.58	5.00
linux	0.52	0.42	3.39	0.12	4.46	6.88	5.04
jdk13	1.15	0.40	2.10	0.05	3.70	6.26	5.00
etext	0.75	0.63	2.65	0.16	4.19	7.57	5.00
chr22	0.49	0.54	6.33	0.17	7.53	8.34	5.49
gcc	0.85	0.40	3.00	0.10	4.36	6.75	5.07
w3c	1.10	0.43	3.18	0.06	4.78	6.31	5.01

	running time				
	bwt	lcp	visit	cmpr	total
sprot	0.70	0.59	1.23	0.11	2.63
rfc	0.60	0.51	1.52	0.11	2.74
howto	0.50	0.46	1.86	0.15	2.96
reut	1.24	0.56	1.32	0.08	3.19
linux	0.52	0.42	2.17	0.12	3.23
jdk13	1.15	0.40	1.48	0.05	3.08
etext	0.75	0.64	1.59	0.16	3.14
chr22	0.49	0.54	0.89	0.18	2.10
gcc	0.86	0.40	1.64	0.10	3.00
w3c	1.10	0.43	2.34	0.06	3.94

Fig. 3. Running time and peak memory usage for the various stages of the BoostRleRc (MEDIUM adaptation) algorithm using the "real cost" model (left) and the "entropy bound" model (right, the table only shows running times since the memory usage is the same as for the "real cost" model). The running times of the four basic steps (bwt computation, LCP array computation, optimal partition computation via suffix tree visit, actual compression using range coding) and the total running time are given in microseconds per input byte. The peak memory usage is given for the LCP array computation and the suffix tree visit which are the steps using more memory. Memory usage is reported as number of used bytes per input byte.

FAST adaptation. It is also remarkable that RleRc with FAST adaptation achieves a very good compression, better indeed that mtf combined with any version of range coding (and the same is true for RleAc FAST). This means that the bwt can be compressed efficiently using rle and an order zero encoder that quickly

adapts to the new statistics. This is somewhat intuitive, but to our knowledge no one observed experimentally the superiority of this strategy w.r.t. mtf, and no theoretical analysis has explained or suggested such behavior. Overall the data show that the boosting algorithm is superior to mtf in terms of compression ratio and it is also more robust in the sense that it works well even with less effective order zero compressors (for example Huffman coding). This superiority is however paid in terms of running time as discussed below.

Running time. The data in Figure 2 show that for range coding the boosting algorithm with the "real cost" model is between 4 and 5 times slower than mtf in compression while there is no significant difference in decompression. For arithmetic and Huffman coding the ratio is even higher. Using the "entropy bound" model the compression time decreases significantly and there is a corresponding loss in compression efficiency. Summing up, mtf and the boosting algorithm (with the two different cost models) offer three different trade offs between compression ratio and compression time: the user can choose the one most suitable for the application at hand. Figure 3 reports the resource usage of the various stages of the boosting algorithm. We can see that the most time consuming step is the optimal partition computation via the suffix tree visit both in the "real cost" and "entropy bound" models. Note also that the peak memory usage is achieved during the LCP array computation.

Wavelet tree performance. The data in Figure 2 show that the algorithms Wavelet and BoostWav roughly achieve the same compression as the algorithms based on Huffman coding (RleHuff and BoostRleHuff) and are inferior to the algorithms based on range/arithmetic encoding. We point out that the similar compression ratio of Wavelet and BoostWav provide an experimental validation of the theoretical analysis of [7] which states that even using a single wavelet tree—as in the algorithm Wavelet—we already achieve the k-th order entropy.

PPMd performance. The results in Figure 2 show that PPMd outperforms all other compressors. Additional tests on the files of the Canterbury corpus (see [6]) show that the Weighted Frequency Count algorithm from [1] (which is based on the bwt) compresses better than mtf, boosting, and wavelet tree algorithms. This suggests that in the field of (bwt) compression Theory is currently a step behind Practice. Although we emphasize that for the construction of compressed indexes it is essential to have simple and efficient bwt-based algorithms whose performance are theoretically guaranteed, we take these results as a stimulus for further research!

References

1. J. Abel. Post BWT stages of the Burrows-Wheeler compression algorithm. Submitted. See also "A fast and efficient post BWT-stage for the Burrows-Wheeler compression algorithm". Proc. *IEEE DCC*, 2005, pag. 449.
2. S. Burkhardt and J. Kärkkäinen. Fast lightweight suffix array construction and checking. In *Proc. 14th Symposium on Combinatorial Pattern Matching (CPM '03)*, pages 55–69. Springer-Verlag LNCS n. 2676, 2003.

3. M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
4. S. Deorowicz. Context exhumation after the BurrowsWheeler transform. *Information Processing Letters*, 95:313–320, 2005.
5. P. Fenwick. Block sorting text compression — final report. Technical Report 130, Dept. of Computer Science, The University of Auckland New Zeland, 1996.
6. P. Ferragina, R. Giancarlo, and G. Manzini. The engineering of a compression boosting library: Theory vs practice in BWT compression. Technical Report TR-INF-2006-06-03-UNIPMN, http://www.di.unipmn.it, 2006.
7. P. Ferragina, R. Giancarlo, and G. Manzini. The myriad virtues of wavelet trees. In *Proc. of International Colloquium on Automata and Languages (ICALP)*, pages 561–572. Springer Verlag LNCS n. 4051, 2006.
8. P. Ferragina, R. Giancarlo, G. Manzini, and M. Sciortino. Boosting textual compression in optimal linear time. *Journal of the ACM*, 52:688–713, 2005.
9. L. Foschini, R. Grossi, A. Gupta, and J. Vitter. Fast compression with a static model in high order entropy. In *IEEE DCC*, pages 62–71. IEEE Computer Society TCC, 2004.
10. R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symp. on Discrete Algorithms (SODA '03)*, pages 841–850, 2003.
11. R. Grossi, A. Gupta, and J. Vitter. When indexing equals compression: Experiments on compressing suffix arrays and applications. In *Proc. 15th Annual ACM-SIAM Symp. on Discrete Algorithms (SODA '04)*, pages 636–645, 2004.
12. H. Kaplan, S. Landau, and E. Verbin. A simpler analysis of Burrows-Wheeler based compression. In *Proc. of the 17th Symposium on Combinatorial Pattern Matching (CPM '06)*. Springer-Verlag LNCS, 2006.
13. T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc. 12th Symposium on Combinatorial Pattern Matching (CPM '01)*, pages 181–192. Springer-Verlag LNCS n. 2089, 2001.
14. M. Lundqvist. Carryless range coding. http://hem.spray.se/mikael.lundqvist/.
15. G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.
16. G. Manzini. Two space saving tricks for linear time LCP computation. In *Proc. of 9th Scandinavian Workshop on Algorithm Theory (SWAT '04)*, pages 372–383. Springer-Verlag LNCS n. 3111, 2004.
17. G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40:33–50, 2004.
18. G. Navarro and V. Mäkinen. Compressed full text indexes. Technical Report TR/DCC-2006-6, Dept. of Computer Science, University of Chile, 2006.
19. J. Seward. The BZIP2 home page, 2006. `http://www.bzip.org`.
20. D. Shkarin. PPMd compressor Ver. J. http://www.compression.ru/ds/.
21. D. Shkarin. PPM: One step to practicality. In *IEEE Data Compression Conference*, pages 202–211, 2002.
22. D. Wheeler. Improving Huffman coding, 1997. `ftp://ftp.cl.cam.ac.uk/users/djw3/huff.ps`.
23. I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987.

The Price of Resiliency: A Case Study on Sorting with Memory Faults*

Umberto Ferraro-Petrillo[1], Irene Finocchi[2], and Giuseppe F. Italiano[3]

[1] Dipartimento di Statistica, Probabilità e Statistiche Applicate, Università di Roma "La Sapienza", P.le Aldo Moro 5, 00185 Rome, Italy
umberto.ferraro@uniroma1.it
[2] Dipartimento di Informatica, Università di Roma "La Sapienza", Via Salaria 113, 00198, Roma, Italy
finocchi@di.uniroma1.it
[3] Dipartimento di Informatica, Sistemi e Produzione, Università di Roma "Tor Vergata", Via del Politecnico 1, 00133 Roma, Italy
italiano@disp.uniroma2.it

Abstract. We address the problem of sorting in the presence of faults that may arbitrarily corrupt memory locations, and investigate the impact of memory faults both on the correctness and on the running times of mergesort-based algorithms. To achieve this goal, we develop a software testbed that simulates different fault injection strategies, and we perform a thorough experimental study using a combination of several fault parameters. Our experiments give evidence that simple-minded approaches to this problem are largely impractical, while the design of more sophisticated resilient algorithms seems really worth the effort. Another contribution of our computational study is a carefully engineered implementation of a resilient sorting algorithm, which appears robust to different memory fault patterns.

1 Introduction

A standard assumption in the design and analysis of algorithms is that the content of memory locations does not change throughout the algorithm execution unless it is explicitly written by the algorithm itself. This assumption, however, may not necessarily hold for very large and inexpensive memories used in modern computing platforms. The trend observed in the design of today's highest-speed memory technologies, in fact, is to avoid the use of sophisticated error checking and correction circuitry, that would impose non-negligible costs in terms of both performance and money: as a consequence, memories may be quite error-prone. Hardware or power failures, as well as environmental conditions such as cosmic rays and alpha particles, can temporarily affect the memory behavior resulting in unpredictable, random, independent failures known as soft memory errors [10].

* Work partially supported by the Sixth Framework Programme of the EU under Contract Number 507613 (Network of Excellence EuroNGI) and by MIUR, the Italian Ministry of Education, University and Research, under Project ALGO-NEXT.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 768–779, 2006.

For instance, a system with Terabytes of memory, such as a large cluster of computing platforms with a few Gigabytes per node, is likely to experience one soft error every few minutes. In the design of reliable systems, when specific hardware for fault detection is not available, it makes sense to assume that the algorithms themselves are in charge of dealing with memory faults. Designing resilient algorithms seems especially important in those large scale applications that demand for large memory capacities at low cost, such as Web search engines. Informally, 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. Classical algorithms are typically non-resilient, and the appearance of even few memory faults may jeopardize their correctness and affect their running time.

The problem of computing with unreliable information has been investigated in a variety of different settings, including the liar model [2, 4, 7, 9] and fault-tolerant sorting networks [1, 8]. In [5], we introduced a faulty-memory random access machine, i.e., a random access machine whose memory locations may suffer from memory faults. In this model, an adaptive adversary may corrupt up to δ memory words throughout the execution of an algorithm. The algorithm cannot distinguish corrupted values from correct ones and can exploit only $O(1)$ safe memory words, whose content gets never corrupted. In [5, 6] we presented matching upper and lower bounds for resilient sorting and searching in this model. In [5] 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. We also proved that we can sort resiliently in $O(n \log n + \delta^3)$ time: this yields an algorithm (Fast) whose running time is optimal in the comparison model as long as $\delta = O((n \log n)^{1/3})$. In [6] we closed the gap between the upper and the lower bound, designing a resilient sorting algorithm (Opt) with running time $O(n \log n + \delta^2)$.

Our results. In this paper we perform a thorough experimental evaluation of the resilient sorting algorithms presented in [5, 6], along with a carefully engineered version of Opt (named Opt-Nb). In order to study the impact of memory faults on the correctness and running time of sorting algorithms, we implemented a software testbed that simulates different fault injection strategies, allowing us to control the number of faults to be injected, the memory location to be altered, and the fault generation time. We performed experiments using a variety of combinations of these parameters and different instance families. In our investigation we first show experimentally that even very few random memory faults can make the sequence produced by a non-resilient sorting algorithm completely disordered: this stresses the need of taking care explicitly of memory faults in the algorithm implementation. We next evaluate the running time overhead of Fast, Opt, and Opt-Nb. Our main findings can be summarized as follows.

- A simple-minded approach to resiliency is largely impractical: it yields an algorithm (Naive) which may be up to hundreds of times slower than its non-resilient counterpart.
- The design of more sophisticated resilient algorithms seems worth the effort: Fast, Opt and Opt-Nb are always much faster than Naive and get close to

Table 1. Summary of the running times of the resilient algorithms under evaluation

Algorithm	NAIVE	FAST	OPT	OPT-NB
Running time	$O(\delta\, n \log n)$	$O(n \log n + \alpha\delta^2)$	$O(n \log n + \alpha\delta)$	$O(n \log n + \alpha\delta)$
Reference		[5]	[6]	[6], This paper

the running time of non-resilient sorting algorithms. In particular, OPT-NB is typically at most 3 times slower than its non-resilient counterpart.
- Despite the theoretical bounds, FAST can be superior to OPT in case of a small number of faults: this suggests that OPT has larger implementation constants. However, differently from OPT, the performance of FAST degrades quickly as the number of faults becomes larger.
- The time interval in which faults happen may influence significantly the running times of FAST, while this seems to have a negligible effect on the running times of OPT and OPT-NB.
- Our engineered implementation OPT-NB typically outperforms its competitors and seems to be the algorithm of choice for resilient sorting.

All the algorithms under investigation make explicit use of an upper bound δ on the number of faults in order to be correct. Since it is not always possible to know in advance the number of memory faults that will occur during the algorithm execution, we analyzed the sensitivity of the algorithms with respect to variations of δ, showing that rounding up δ (in absence of a good estimate) does not affect significantly the performances of OPT and OPT-NB. Finally, we considered a more realistic scenario where algorithms with larger execution times are likely to incur in a larger number of memory faults. In this model we observed the same relative performances of the algorithms, with even more remarked differences in their running times. Also in this case, OPT and OPT-NB appear to be more robust than FAST and can tolerate higher fault rates.

2 Resilient Sorting Algorithms

In this section we recall the mergesort-based resilient algorithms presented in [5, 6]. Their worst-case running times are summarized in Table 1 as a function of the number n of keys to be sorted, the upper bound δ on the total number of faults, and the actual number α of faults that happen during a specific execution.

We will say that a key is faithful if its value is never corrupted by any memory fault, and that a sequence is k-unordered, for some $k \geq 0$, if the removal of at most k faithful keys yields a subsequence in which all the faithful keys are sorted. A sorting or merging algorithm is resilient if its output is 0-unordered. A simple-minded resilient variant of standard merging takes the minimum among $(\delta + 1)$ keys per sequence at each merge step, and thus considers at least one faithful key per sequence. By plugging this into mergesort, we obtain a resilient sorting algorithm, called NAIVE, with running time $O(\delta\, n \log n)$.

Two Basic Tasks: Merging and Purifying. With respect to NAIVE, the algorithms FAST and OPT reduce the time spent to cope with memory faults

to an *additive* overhead (see Table 1). This is achieved by temporarily relaxing the requirement that the merging must produce a 0-unordered sequence and by allowing its output to be k-unordered, for some $k > 0$. We now describe the main subroutines used by algorithms FAST and OPT.

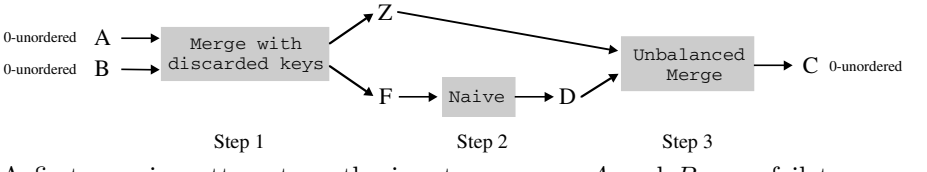
Weakly-resilient merge [5] is an $O(n)$-time merging algorithm, which, although unable to produce a 0-unordered sequence, can guarantee that not too many faithful keys are out of place in the output sequence. It resembles classical merging, with the addition of suitable checks and error recovery. Checks are performed when the algorithm keeps on advancing in one of the two sequences for $(2\delta + 1)$ consecutive steps: if a check fails, a faulty key can be identified and removed from further consideration. In [5] we proved that each faulty key may prevent $O(\delta)$ faithful keys from being returned at the right time: this implies that the output sequence is $O(\alpha\delta)$-unordered.

Purify [5] is a resilient variant of the Cook-Kim division algorithm [3]. Given a k-unordered sequence X of length n, it computes a 0-unordered subsequence S in $O(n + \delta \cdot (k + \alpha))$ worst-case time. It is guaranteed that the length of S is at least $n - 2(k + \alpha)$, i.e., only $O(k + \alpha)$ keys are discarded in order to purify X.

Purifying-merge [6] is a fast resilient merging algorithm that may nevertheless fail to merge all the input keys: the algorithm produces a 0-unordered sequence Z and a disordered fail sequence F in $O(n + \alpha\delta)$ worst-case time, where $|F| = O(\alpha)$, i.e., only $O(\alpha)$ keys can fail to get inserted into Z. This is an improvement over the weakly-resilient merge described above (obtained at a small price on the running time), and is achieved by a clever use of buffering techniques and more sophisticated consistency checks on data. Namely, the algorithm uses auxiliary buffers of size $\Theta(\delta)$, in which keys to be merged are copied and from which merged keys are extracted. Thanks to the use of buffers, the total cost of identifying faulty keys and moving them to the fail sequence is only $O(\alpha\,\delta)$ [6].

Unbalanced merge [5] requires superlinear time, but is particularly well suited at merging unbalanced sequences. It works by repeatedly extracting a key from the shorter sequence and placing it in the correct position with respect to the longer sequence: we need some care to identify this proper position, due to the appearance of memory faults. The algorithm runs in $O(n_1 + (n_2 + \alpha) \cdot \delta)$ time, where n_1 and n_2 denote the lengths of the sequences, with $n_2 \leq n_1$.

Fast Resilient Sorting. Consider the merging algorithm represented by the figure below:



A first merging attempt on the input sequences A and B may fail to merge all the input keys, producing a 0-unordered sequence Z and a disordered fail sequence F. The sequence F is sorted using algorithm NAIVE: this produces another 0-unordered sequence D. The two 0-unorderded unbalanced sequences, Z and D, can be finally merged using unbalanced merging.

Step 1 (the first merging) can be implemented either by using the weakly-resilient merge and then purifying its output sequence, or by invoking directly the purifying-merge algorithm: in the former case $|F| = O(\alpha\delta)$ and the merge running time is $O(n + \alpha\delta^2)$, while in the latter case $|F| = O(\alpha)$ and the merge running time is $O(n+\alpha\delta)$. A resilient sorting algorithm can be obtained by plugging these merging subroutines into mergesort: the two implementation choices for Step 1 yield algorithms FAST and OPT, respectively. We refer the interested reader to references [5, 6] for the low-level details of the method.

Algorithm Implementation Issues. We implemented all the algorithms in `C++`, within the same algorithmic and implementation framework. Since recursion may not work properly in the presence of memory faults (the recursion stack may indeed get corrupted), we relied on a bottom up iterative implementation of mergesort, which makes $\lceil \log_2 n \rceil$ passes over the array, where the i-th pass merges sorted subarrays of length 2^{i-1} into sorted subarrays of length 2^i. For efficiency issues we applied the standard technique of alternating the merging process from one array to the other in order to avoid unnecessary data copying. We also took care of using only $O(1)$ reliable memory words to maintain array indices, counters, and memory addresses. With respect to algorithm OPT, again for efficiency reasons, we avoided to allocate/deallocate its auxiliary buffers at each call of the merging subroutine. In spite of this, in a first set of experiments the buffer management overhead slowed down the execution of OPT considerably for some choices of the parameters. Hence, we implemented and engineered a new version of OPT with the same asymptotic running time but which avoids completely the use of buffers: throughout this paper, we will refer to this implementation as OPT-NB (i.e., OPT with No Buffering). The merging subroutine used by OPT-NB benefits from the same approach used by algorithm FAST, i.e., it avoids copying data to/from the auxiliary buffers by maintaining a constant number of suitable array indices and by working directly on the input sequences.

3 Experimental Framework

In this section we describe our experimental framework, discussing fault injection strategies, performance indicators, and additional implementation details.

Fault Injection: a Simulation Testbed. In order to study the impact of memory faults on the correctness and running time of sorting algorithms, we implemented a software testbed that simulates different fault injection strategies. Our simulation testbed is based on two threads: the *sorting thread* runs the sorting algorithm, while the *corrupting thread* is responsible of injecting memory faults. In the following we discuss where, when, and how many faults can be generated by the corrupting thread.

Fault location. In order to simulate the appearance of memory faults, we implemented an *ad-hoc* memory manager: faults are injected in memory locations that are dynamically allocated through our manager. The location to be altered by a fault is chosen uniformly at random. As previously observed, the algorithms'

implementation is such that, at any time during the execution, only a constant number of reliable memory words is in use. To maximize the damage produced by a fault, the new value of the corrupted memory location is chosen (at random) so as to be always larger than the old value.

Fault injection models. The number of faults to be injected can be specified according to two different models. The *upper bound model* requires an upper bound δ on the total number of memory faults, and the actual number α of faults that should happen during the execution of an algorithm: it must be $\alpha \leq \delta$. The fault injection strategy ensures that exactly α memory faults will occur during the execution of an algorithm, independently from the algorithm's running time. This assumption, however, may not be true in a more realistic scenario, where algorithms with larger execution times are likely to incur in a larger number of faults. The *fault-per-second* model overcomes this limitation (not addressed by the theoretical model of [5]) by using faults generated on a periodic time basis and independently from the algorithm's running time. In particular, this model requires to specify the number σ of faults that must be injected per each second of execution. We note that the algorithms under investigation were not designed for the fault-per-second model, as they make explicit use of δ in their implementation. Hence, to stress those algorithms on a different terrain, we start from a number σ of faults per second and must generate a suitable value of δ: for each σ and instance size n, we experimented with different values of δ, paying attention to use only values that guarantee the algorithms to be correct (if any).

Fault generation time. In the upper bound model, before running the algorithm the corrupting thread precomputes α breakpoints, at which the sorting thread will be interrupted and one fault will be injected. The breakpoints can be spread over the entire algorithm execution, or concentrated in a given temporal interval (e.g., at the beginning or at the end of the execution). In both cases the corrupting thread needs an accurate estimate of the algorithm's running time in order to guarantee that $\Theta(\alpha)$ faults will be generated and evenly spread as required: an automatic tuning mechanism takes care of this estimate. In the fault-per-second model, faults are simply generated at regular time intervals.

Experimental Setup. Our experiments have been carried out on a workstation equipped with two Opteron processors with 2 GHz clock rate and 64 bit address space, 2 GB RAM, 1 MB L2 cache, and 64 KB L1 data/instruction cache. The workstation runs Linux Kernel 2.6.11. All programs have been compiled through the GNU `gcc` compiler version 3.3.5 with optimization level `O3`. The full package, including algorithm implementations and memory manager, is publicly available at the URL `http://www.dsi.uniroma1.it/~finocchi/experim/faultySort/`. Unless stated otherwise, in our experiments we average each data point on ten different instances, and, for each instance, on five runs using different fault sequences on randomly chosen memory locations. Random values are produced by the `rand()` pseudo-random source of numbers provided by the ANSI C standard library. The running time of each experiment is measured by means of the standard system call `getrusage()`. The sorting and the corrupting threads run as two different parallel processes on our biprocessor architecture and operating system:

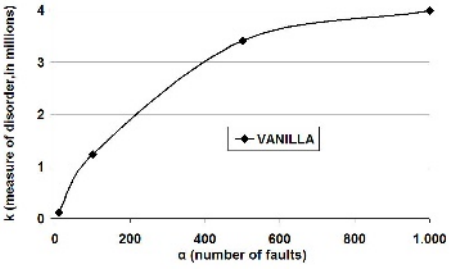


Fig. 1. Disorder produced by random memory faults in the sequence output by VANILLA mergesort. In this experiment $n = 5 \cdot 10^6$ and $\alpha = \delta$ increases up to 1000.

concurrent accesses to shared memory locations (e.g., the corruption of a key in unreliable memory) are solved at the hardware level by spending only a few CPU cycles. This allows us to get a confident measure of the algorithms' running time, without taking into account also the time spent for injecting faults.

4 Experimental Results

In this section we summarize our main experimental findings. We performed experiments using a wide variety of parameter settings and instance families. For lack of space, in this extended abstract we only report the results of our experiments with uniformly distributed integer keys: to ensure robustness of our analysis, we also experimented with skewed inputs such as almost sorted data and data with few distinct key values, and obtained similar results. Similarly, most of the experiments that we describe here are carried out in the upper bound fault injection model: the same relative performances of the algorithms have been observed in the fault-per-second model, where the differences in the running times are even more remarked.

The Price of Non-resiliency: Correctness. Our first aim was to measure the impact of memory faults on the correctness of the classical (non-resilient) mergesort, which we refer to as VANILLA mergesort. In the worst case, when merging two n-length sequences, a single memory fault may be responsible for a large disorder in the output sequence: namely, it may be necessary to remove as many as $\Theta(n)$ elements in order to obtain a 0-unordered subsequence. Since memory faults affecting large and inexpensive memories are not typically generated according to an adversarial pattern, a natural question to ask is whether the output can be completely out of order even when few faults hit memory locations at random. In order to characterize the non-resiliency of VANILLA mergesort in the presence of random memory faults, we ran this algorithm on several input sequences with a fixed number of elements while injecting an increasing number of faults spread over the entire algorithm's execution time. The correctness of the output has been measured using the k-unordered metric, which is a classical measure of disorder (see Section 2). The outcome of the experiment, exemplified

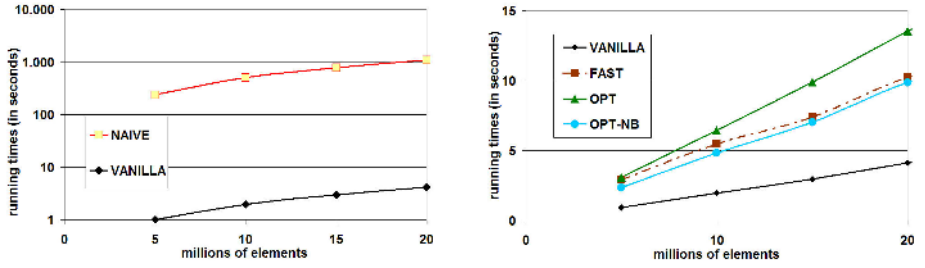


Fig. 2. Running times of (a) NAIVE and VANILLA (b) FAST, OPT, OPT-NB, and VANILLA on random input sequences of increasing length. In this test $\alpha = \delta = 500$.

in Figure 1, shows a deep vulnerability of VANILLA mergesort even in the presence of very few random faults. As it can be seen from Figure 1, when sorting 5 million elements, it is enough to have only 1000 random faults (i.e., roughly only 0.02% of the input size) to get a k-unordered output sequence for $k \approx 4 \cdot 10^6$: in other words, only 0.02% faults in the input are able to produce errors in approximately 80% of the output.

The Price of Resiliency: Running Time Overhead. In order to operate correctly, resilient algorithms must cope with memory faults and be prepared to pay some overhead in their running times. In the following experiments we will try to evaluate this overhead by comparing the sorting algorithms of Table 1 with respect to the non-resilient VANILLA mergesort.

Overhead of NAIVE. Once the need for resilient algorithms is clear, even in the presence of very few non-pathological faults, another natural question to ask is whether we really need sophisticated algorithms for this. Put in other words, one might wonder whether a simple-minded approach to resiliency (such as the one used by algorithm NAIVE) would be enough to yield a reasonable performance in practice. To answer this question, we measured the overhead of NAIVE on random input sequences. Figure 2(a) illustrates one such experiment, where we measured the running times of VANILLA and NAIVE on random input sequences of increasing length by keeping fixed the number of faults injected during the sorting process ($\alpha = \delta = 500$). As suggested by the theoretical analysis, the $\Theta(\delta)$ multiplicative factor in the running times of NAIVE makes this algorithm even hundreds of time slower than its non-resilient counterpart, and thus largely impractical. For this reason, we will not consider NAIVE any further in the rest of the paper.

Overhead of FAST, OPT, *and* OPT-NB. According to the theoretical analysis, algorithms FAST, OPT, and OPT-NB are expected to be much faster than NAIVE. All our experiments confirmed this prediction. The chart in Figure 2(b), for instance, has been obtained on the same data sets used for the experiment reported in Figure 2(a), and shows that FAST, OPT, and OPT-NB perform very well for this choice of the parameters: indeed, they exhibit a running time which approxi-

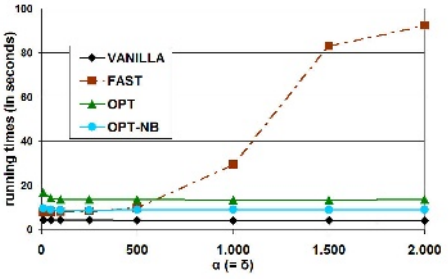


Fig. 3. Running times of FAST, OPT, OPT-NB, and VANILLA on random input sequences of length $n = 20 \cdot 10^6$ and increasing number of injected faults.

mately ranges from 2.5 times to 3 times the running time of VANILLA mergesort, while NAIVE appears to be more than two orders of magnitude slower. Note that, despite the theoretical bounds, FAST seems to have a better performance than OPT on this data set. This suggests that OPT may have larger implementation constants (probably due to the buffer management overhead) and that there are situations where FAST is able to perform better than OPT, at least in the presence of few faults. Our efforts in engineering OPT so as to avoid the use of buffers seem to pay off and confirm this intuition: indeed, in this experiment OPT-NB performs always better than both OPT and FAST.

Impact of faults on the running time. According to the asymptotic analysis, we would also expect that the performance of FAST, OPT, and OPT-NB degrade substantially as the number of faults becomes larger. In order to check this, we designed experiments in which the length of the input sequence is fixed (e.g., $n = 20 \cdot 10^6$) but the number $\alpha = \delta$ of injected faults increases. One of those experiments is illustrated in Figure 3, and shows that only the running time of FAST seems heavily influenced by the number of faults. OPT and OPT-NB, instead, appear to be quite robust as their running times tend to remain almost constant for all the values of δ considered in the experiment. To get a deeper understanding of this phenomenon, we profiled all the algorithms: in particular, for FAST we observed that when δ is large, most of the time is spent in Step 2, where the disordered fail sequence F returned by `Purify` is sorted by means of algorithm NAIVE. This suggests that either the number of calls to NAIVE or the number of elements to be sorted in each call tends to be bigger in FAST than in OPT. Computing the average and the maximum length of the fail sequences F throughout the algorithm execution confirmed the second hypothesis. As shown in Figure 4, the fail sequences in FAST can be even $10,000$ times larger than in OPT, thus suggesting that the theoretical upper bounds on $|F|$ (i.e., $O(\alpha\,\delta)$ and $O(\alpha)$ for FAST and OPT, respectively) are likely to be pretty tight also in the case of random memory faults.

Early versus late faults. In all the experiments presented so far, we have considered random faults uniformly spread in time over the entire execution of the algorithm. The time interval in which faults happen, however, may significantly influence the running time of the algorithm. Indeed all of the resilient algorithms

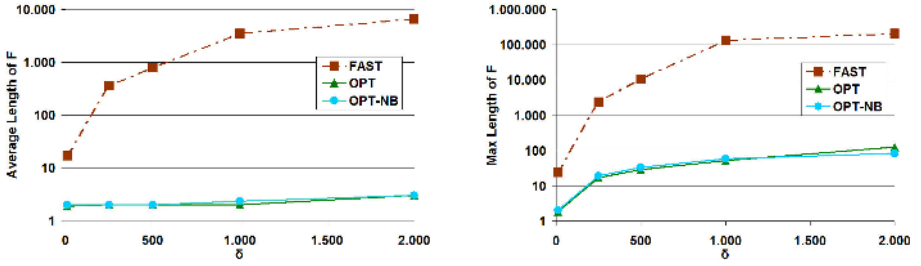


Fig. 4. Average and maximum length of the fail sequence F to be sorted by NAIVE

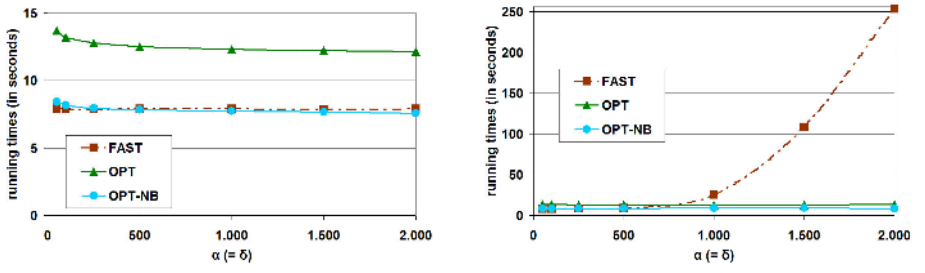


Fig. 5. Increasing number of faults concentrated in the first 20% and in the last 20% of the algorithms' running times ($n = 20 \cdot 10^6$ in this experiment)

considered tend to process many sequences of smaller size in the initial stage of their execution, and few sequences of larger size at the end of their execution. As a consequence, faults occurring in the initial phase of the execution will likely produce short fail sequences, while faults occurring during the ending phase may produce longer fail sequences. Since sorting fail sequences appears to be a bottleneck in the resilient algorithms, faults appearing at the beginning or at the end of the execution may produce quite different running times. To check this, we performed experiments in which we injected faults only in the initial and in the final 20% of the running time. The outcome of one of those experiments, presented in Figure 5, confirms that faults occurring early during the execution of FAST are processed more quickly than faults occurring late. The effect is instead negligible for algorithms OPT and OPT-NB: this seems due to the fact that the fail sequences of these algorithms are always short, independently of the fault generation time (see also Figure 4).

Sensitivity to δ. All the resilient algorithms considered in our experiments need explicitly an upper bound δ on the number of faults that may happen throughout their execution. This is not a problem when δ is known in advance. However, if the rate of faults is unknown, the algorithms need at least an estimate on δ to work properly: on the one side, rounding up δ may lead to much slower running times; on the other side, rounding it down may compromise the whole correctness of the resilient sorting algorithm. In all the experiments described up

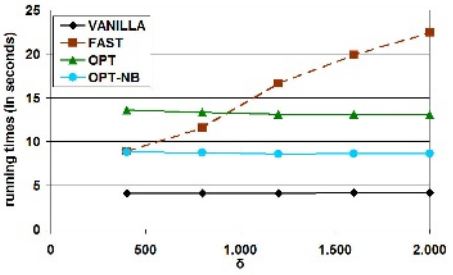


Fig. 6. Sensitivity to δ: in this experiment $n = 20 \cdot 10^6$, $\alpha = 400$, and $\delta \leq 2000$

to now we used $\delta = \alpha$. We now discuss issues related to finding a good estimate for δ in the upper bound and in the fault-per-second models, respectively.

Upper bound model. In the experiment illustrated in Figure 6 we analyzed the sensitivity of FAST, OPT, and OPT-NB with respect to variations of δ: we run the algorithms keeping the actual number of faults fixed ($\alpha = 400$) and increasing δ from 400 to 2000. When $\delta = 400$, this simulates a good estimate of δ; as δ gets much larger than α, this tend to simulate bad estimates of δ. Note that, while the performances of OPT and OPT-NB are substantially unaffected by the increase on the value of δ, the running time of FAST seems to grow linearly with δ: once again, this depends on the fact that the length of the fail sequences in FAST is proportional to δ, differently from what happens in the case of both OPT and OPT-NB. As a result, we can argue that OPT and OPT-NB appear to be much less vulnerable than FAST to potential bad estimates on the value of δ.

Fault-per-second model. The fault-per-second injection model introduces a major novelty with respect to the upper bound model, as the actual number of faults that will be generated throughout the execution of an algorithm is not bounded *a priori*. Thus, the quest for a good estimate of δ here is even more crucial. Note that in this model rounding up δ not only pushes additional overhead on the resilient sorting algorithm, but also increases the number of faults that will actually occur throughout the execution, because the running time becomes larger. In more details, let $t(n, \delta)$ denote the running time (in seconds), and let σ be the number of faults generated per each second of execution. Then $\sigma \cdot t(n, \delta)$ is the actual number of faults that will occur, and the algorithm is guaranteed to be correct only if $\delta \geq \sigma \cdot t(n, \delta)$. Since the running time t is an increasing function of δ itself, if the fault injection rate is too fast (i.e., σ is too large) it may be even possible that no value of δ satisfies the above inequality. To investigate this issue, we tried to determine, given the number σ of faults per seconds, the smallest value of δ that is larger than the actual number $\sigma \cdot t(n, \delta)$ of faults: we will refer to this value of δ as the *correctness threshold*. Table 2 reports the correctness thresholds for the three algorithms FAST, OPT, and OPT-NB corresponding to six different values of σ and to $n = 20 \cdot 10^6$. As one may expect, such thresholds increase with σ: this is because larger values of σ yield larger total numbers of injected faults. The experiment also confirms our intuition that a correctness

Table 2. Correctness thresholds for $n = 20 \cdot 10^6$ and $\sigma \in [10, 60]$

Algorithm	$\sigma = 10$	$\sigma = 20$	$\sigma = 30$	$\sigma = 40$	$\sigma = 50$	$\sigma = 60$
FAST	100	180	290	410	-	-
OPT-NB	100	210	300	400	500	590
OPT	160	310	445	620	770	880

threshold may not always exist, limiting the possibility of using the algorithms in the fault-per-second model only when the fault injection rate is small enough. In particular, it is remarkable that FAST cannot tolerate more than 50 faults per second, while OPT and OPT-NB appear to be more robust and can tolerate much higher fault injection rates. This is in line with the previous experiments, where we observed that the running time of FAST grows much more quickly than the running times of OPT and OPT-NB as the number of faults increases. We used the correctness thresholds, if any, to compare FAST, OPT, and OPT-NB in the fault-per-second model. Our experiments confirmed the relative performances observed in the upper bound model, and exhibit even more remarked differences in the running times of these algorithms. For lack of space, we defer to the full paper a detailed description of our experiments in this setting.

References

1. S. Assaf and E. Upfal. Fault-tolerant sorting networks. *SIAM J. Discrete Math.*, 4(4), 472–480, 1991.
2. R. S. Borgstrom and S. Rao Kosaraju. Comparison based search in the presence of errors. *Proc. 25th STOC*, 130–136, 1993.
3. C. R. Cook and D. J. Kim. Best sorting algorithms for nearly sorted lists. *Comm. of the ACM*, 23, 620–624, 1980.
4. U. Feige, P. Raghavan, D. Peleg, and E. Upfal. Computing with noisy information. *SIAM J. on Comput.*, 23, 1001–1018, 1994.
5. I. Finocchi and G. F. Italiano. Sorting and searching in the presence of memory faults (without redundancy). *Proc. 36th STOC*, 101–110, 2004.
6. I. Finocchi, F. Grandoni, and G. F. Italiano. Optimal sorting and searching in the presence of memory faults. To appear in *Proc. 33rd ICALP*, 2006.
7. D. J. Kleitman, A. R. Meyer, R. L. Rivest, J. Spencer, and K. Winklmann. Coping with errors in binary search procedures. *J. Comp. Syst. Sci.*, 20:396–404, 1980.
8. T. Leighton and Y. Ma. Tight bounds on the size of fault-tolerant merging and sorting networks with destructive faults. *SIAM J. on Comput.*, 29(1):258–273, 1999.
9. A. Pelc. Searching games with errors: Fifty years of coping with liars. *Theoret. Comp. Sci.*, 270, 71–109, 2002.
10. A. J. Van de Goor. *Testing semiconductor memories: Theory and practice*, ComTex Publishing, Gouda, The Netherlands, 1998.

How Branch Mispredictions Affect Quicksort

Kanela Kaligosi[1] and Peter Sanders[2]

[1] Max Planck Institut für Informatik
Saarbrücken, Germany
kaligosi@mpi-sb.mpg.de
[2] Universität Karlsruhe, Germany
sanders@ira.uka.de

Abstract. We explain the counterintuitive observation that finding "good" pivots (close to the median of the array to be partitioned) may not improve performance of quicksort. Indeed, an intentionally *skewed* pivot *improves* performance. The reason is that while the instruction count decreases with the quality of the pivot, the likelihood that the direction of a branch is mispredicted also goes up. We analyze the effect of simple branch prediction schemes and measure the effects on real hardware.

1 Introduction

Sorting is one of the most important algorithmic problems both practically and theoretically. Quicksort [1] is perhaps the most frequently used sorting algorithm since it is very fast in practice, needs almost no additional memory, and makes no assumptions on the distribution of the input. Hence, quicksort, its analysis and efficient implementation is discussed in most basic courses on algorithms. When we take a random pivot, the expected number of comparisons is $2n \ln n \approx 1.4n \lg n$. One of the most well known optimizations is that taking the median of three elements reduces the expected number of comparisons to $\frac{12}{7} n \ln n \approx 1.2n \lg n$ [2]. Indeed, by using the median of a larger random sample, the expected number of comparisons can be made as close to $n \lg n$ as we want [3]. For sufficiently large inputs, the increased overhead for pivot selection is negligible. At first glance, counting comparisons makes a lot of practical sense since in quicksort, the number of executed instructions and cache faults grow proportionally with this figure.

However, in comparison based sorting algorithms like quicksort or mergesort, neither the executed instructions nor the cache faults dominate execution time. Comparisons are much more important, but only indirectly since they cause the direction of branch instructions depending on them to be mispredicted. In modern processors with long execution pipelines and superscalar execution, dozens of subsequent instructions are executed in parallel to achieve a high peak throughput. When a branch is mispredicted, much of the work already done on the instructions following the predicted branch direction turns out to be wasted. Therefore, ingenious and very successful schemes have been devised to accurately *predict* the direction a branch takes. Unfortunately, we are facing a

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 780–791, 2006.

dilemma here. Information theory tells us that the optimal number of $\approx n \lg n$ element comparisons for sorting can only be achieved if each element comparison yields one bit of information, i.e., there is a 50 % chance for the branch to take either direction. In this situation, even the most clever branch prediction algorithm is helpless. A painfully large number of branch mispredictions seems to be unavoidable.

Related Work: This dilemma can be circumvented by devising sorting algorithms where comparisons are decoupled from branches [4]. However, the algorithm proposed in [4] is not in-place and requires compiler optimizations that are not universally available yet. Hence it remains interesting to see what can be done about branch mispredictions in quicksort. [5] based on a discussion between Sanders and Moruz in 2004 observes that a reduced number of branch mispredictions improves the running time of quicksort when inputs are almost sorted. In [6], a variant of multiway mergesort is proposed that reduces branch mispredictions by sequentially searching for the next element to be merged. This algorithm is analyzed for the case of static branch prediction. Compared to this, the innovation of the present paper is that it gives experimental results and concerns a classical, in-place algorithm. Moreover, for quicksort also dynamic branch prediction is interesting. Martinez and Roura [3] note that nonmedian pivots can be beneficial if swaps are much more expensive than comparisons. However, it seems that this situation would correspond to a nonoptimal use of quicksort because then it would be more efficient to sort references to the elements first, followed by a permutation of the original input.

Overview: In Section 2, we review quicksort and basic branch prediction mechanisms. Section 3 outlines our main theoretical contributions — an analysis of quicksort in the context of branch mispredictions. For simplicity we assume that the elements are distinct. We look at two variants of quicksort: random and skewed pivot, and three branch prediction methods: static, 1-bit predictor and 2-bit predictor. To the best of our knowledge this represents the first analysis of the interactions of a nontrivial algorithm with dynamic branch prediction methods. Note that static branch prediction is not useful for analyzing quicksort variants like random pivot that try to approximate the median. The theoretical results are complemented by experiments in Section 4. In particular, there we also look at the classical median-of-three pivot selection. It turns out that this frequently used improvement only gives a negligible advantage over random pivot. Its advantages wrt. instruction count basically cancel with its disadvantages wrt. branch prediction. Somewhat surprisingly, taking a pivot with rank around $n/10$ can lead to a better performance.

2 Preliminaries

In this section we give a more detailed description of quicksort and then give an overview of several branch prediction schemes.

2.1 Quicksort

A simple pseudocode of quicksort sufficient for our purposes can be seen in Algorithm 1. In the rest of the paper, it will be clear from the context, whether n denotes the input size or the currently relevant subproblem size. The algorithm can be instantiated with different subroutines for determining the pivot. We distinguish between three basic schemes: *random pivot*, *median-of-three* random elements, and α-*skewed pivot*, i.e., a median of rank αn. Note that the latter scheme is an idealization because in practice only approximations of this value can be obtained efficiently (using random sampling [3]). However, for sufficiently big inputs, one could get very good approximations at negligible cost for all but the lowest levels of recursion.

Algorithm 1. Sort array part $a[\ell..r]$

```
Procedure quicksort(ℓ, r : integer);
  if r > ℓ then
    i = ℓ; j = r; x = pivot();
    repeat
      while a[i] < x do i++ ; endwhile {Loop I}
      while a[j] > x do j-- ; endwhile {Loop J}
      if i ≤ j then swap(a[i], a[j]);
    until j ≤ i
    quicksort(ℓ, i − 1);
    quicksort(i + 1, r);
  end if
```

2.2 Branch Prediction Schemes

In *static branch prediction* the compiler once and for all labels a branch instruction as predict-taken or as predict-not-taken. This scheme does not take into account the dynamic behavior of the program. Static prediction is useful together with α-skewed pivot selection. For $\alpha < 1/2$, the compiler should statically predict that Loop I is not executed and that Loop J is executed.[1]

In the standard versions of pivot selection that attempt to approximate the median, static prediction does not help. Here *dynamic branch prediction* mechanisms provided by the hardware may do better.

The simplest dynamic scheme is a *1-bit predictor*. The hardware always predicts a branch instruction to take the same direction it took the last time it was executed.

A refined version working better in practice is the *2-bit predictor*. In order to further improve the prediction accuracy, 2-bit prediction schemes were introduced. In these schemes the prediction must be wrong twice before it is changed. See for example [7]. In Fig. 1 we can see the behavior of a 2-bit predictor scheme.

In fact we can have the general case of a k-bit counter. As in the 2-bit case, the counter is incremented if the branch is taken and decremented if the branch

[1] Modern compilers do that using profiling information.

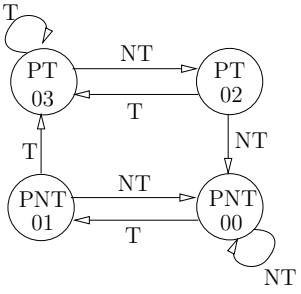


Fig. 1. 2-bit prediction scheme: There are four states, where PT means Predict Taken and PNT means Predict Not Taken. The arrows show how the states are changed when a branch is taken T or not taken NT.

is not taken. The branch is predicted taken when the counter is greater than or equal to half of its maximum value and not taken otherwise. The k-bit prediction schemes are not widely used since studies have shown that the 2-bit prediction scheme is good enough for all practical purposes.

Furthermore, there are branch prediction schemes which not only consider the history of the particular branch to be predicted but also that of other branches which may be related to the current branch and affect its outcome. In this way the prediction accuracy is further improved. See [7] for more details. It looks difficult to analyze quicksort for the most general schemes. It also seems that simple local prediction is adequate in the case of quicksort since the past behavior of a branch instruction is likely to yield information whether the pivot is larger or smaller than the median.

3 Analysis

In this section we analyze the behavior of quicksort in terms of the number of branch mispredictions it incurs. We give the analysis of the branch mispredictions occurring in the two inner and consecutive while loops of quicksort that perform the partitioning step. Note that the remaining branch instructions are much less frequently executed or easy to predict.

In the next three subsections we outline the proof of the following theorem.

Theorem 1. *Let* $H(\alpha) = -(\alpha \lg(\alpha) + (1-\alpha)\lg(1-\alpha))$ *be the binary entropy function. The number of branch mispredictions that occur during the execution of the partitioning step of quicksort are as described in Table 1. The entries for the 1-bit and the 2-bit predictor give the expected number of branch mispredictions given the assumption that there is a probability* α *of an element being smaller when compared with the pivot that has rank* αn.[2] *For the entry random pivot*

[2] This assumption means that our analysis is "heuristic" since the knowledge that the pivot has rank αn introduces slight dependencies between the comparisons. It is an interesting question whether there is an easy argument proving the same bounds for the standard average case model.

Table 1. Number of branch mispredictions

	random pivot	α-skewed pivot
static predictor	$\frac{\ln 2}{2} n \lg n + \mathcal{O}(n)$, $\frac{\ln 2}{2} \approx 0.3466$	$\frac{\alpha}{H(\alpha)} n \lg n + \mathcal{O}(n)$, $\alpha < 1/2$ $\frac{1-\alpha}{H(\alpha)} n \lg n + \mathcal{O}(n)$, $\alpha \geq 1/2$
1-bit predictor	$\frac{2 \ln 2}{3} n \lg n + \mathcal{O}(n)$, $\frac{2 \ln 2}{3} \approx 0.4621$	$\frac{2\alpha(1-\alpha)}{H(\alpha)} n \lg n + \mathcal{O}(n)$
2-bit predictor	$\frac{28 \ln 2}{45} n \lg n + \mathcal{O}(n)$, $\frac{28 \ln 2}{45} \approx 0.4313$	$\frac{2\alpha^4 - 4\alpha^3 + \alpha^2 + \alpha}{(1-\alpha(1-\alpha))H(\alpha)} n \lg n + \mathcal{O}(n)$

with static predictor there is no such assumption and for the entry α-skewed with static predictor we give a worst case analysis.

In Fig. 2 we see the α-dependent coefficients of $n \lg n$ for the case of the α-skewed pivot. As expected they are maximized for $\alpha = 0.5$ and their value decreases as we move towards smaller or larger α's. Moreover, the best curve is the one for the static predictor, followed by the one for the 2-bit predictor and then the one for the 1-bit predictor.

3.1 Static Prediction Scheme

Next we analyze the number of branch mispredictions quicksort could achieve with static branch prediction if somebody would tell the predictor whether the pivot is smaller or larger than the median. We can judge dynamic branch prediction by comparing its performance with this "best possible" prediction. We consider the random pivot and the α-skewed pivot case. For the former we give

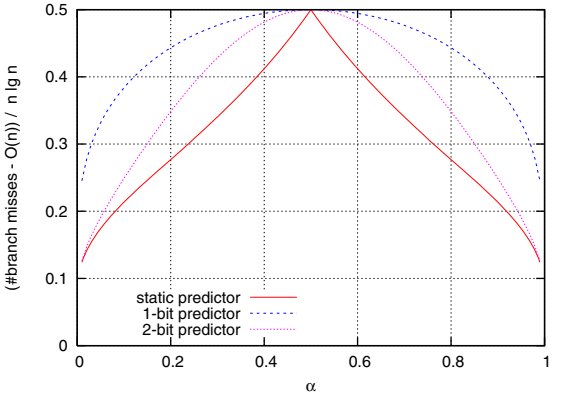


Fig. 2. The α-dependent coefficients of $n \lg n$ for varying α

an expected case analysis that holds for every input, namely we make no assumptions for the distribution of the input. For the latter we give a worst case analysis.

Let $B^{\text{stat}}(n)$ denote the expected number of branch mispredictions occurring in the partitioning step of quicksort with random pivot when a static predictor is used. Consider one execution of the partitioning step. Let x be the pivot element and αn its rank for some $0 < \alpha \leq 1$. The rank αn of the pivot can take each of the values $1, \ldots, n$ with equal probability and after the partitioning we are left with subproblems of size $\alpha n - 1$ and $(1-\alpha)n$. If $\alpha < 1/2$ then each element smaller than the pivot causes a branch misprediction, since Loop I is predicted not to be executed and Loop J is predicted to be executed. Therefore, we have at most αn branch mispredictions. If $\alpha \geq 1/2$ the prediction of the loop is the other way around and each element larger than the pivot causes a branch misprediction and therefore we have $(1-\alpha)n$ mispredictions. So, we set up the following recurrence for $n \geq 1$, with $B^{\text{stat}}(0) = 1$.

$$B^{\text{stat}}(n) \leq \frac{1}{n}\Big(\sum_{\alpha n=1}^{\lfloor n/2 \rfloor} B^{\text{stat}}(\alpha n - 1) + B^{\text{stat}}((1-\alpha)n) + \alpha n) + \sum_{\alpha n=\lfloor n/2 \rfloor+1}^{n} (B^{\text{stat}}(\alpha n - 1) + B^{\text{stat}}((1-\alpha)n) + (1-\alpha)n\Big).$$

We solve the recurrence using for example the technique in [8] and we obtain $B^{\text{stat}}(n) \leq \frac{\ln 2}{2} n \lg n + \mathcal{O}(n)$.

Now let $A^{\text{stat}}(n)$ be the number of branch mispredictions of quicksort with α-skewed pivot when the static predictor is used. Similarly to above if $\alpha < 1/2$ each element smaller than the pivot causes a branch misprediction and if $\alpha \geq 1/2$ each element larger than the pivot causes a branch misprediction. So, we set up the following recurrence for $n \geq 1$, with $A^{\text{stat}}(0) = 1$.
$A^{\text{stat}}(n) \leq \alpha n + A^{\text{stat}}(\alpha n - 1) + A^{\text{stat}}((1-\alpha)n)$, if $\alpha < 1/2$ and
$A^{\text{stat}}(n) \leq (1-\alpha)n + A^{\text{stat}}(\alpha n - 1) + A^{\text{stat}}((1-\alpha)n)$, if $\alpha \geq 1/2$. We can prove by induction that $A^{stat}(n) \leq \frac{\alpha}{H(\alpha)} n \lg n + \mathcal{O}(n)$, if $\alpha < 1/2$ and
$A^{stat}(n) \leq \frac{1-\alpha}{H(\alpha)} n \lg n + \mathcal{O}(n)$, if $\alpha \geq 1/2$.

3.2 1-Bit Prediction Scheme

We now analyse quicksort when a 1-bit prediction scheme is used. In the 1-bit prediction scheme we predict that a branch instruction will go in the same direction as the last time it was executed. Let X_i be the indicator random variable which is 1 if the i-th element in Loop I causes a branch misprediction and 0 otherwise. Correspondingly we define Y_j for Loop J. We have that $X_i = 1$ if $a[i] \geq x$ and $a[i-1] < x$ or if $a[i] < x$ and $a[i-1] \geq x$. Using our assumption that $P[a[i] < x] = \alpha$, we get $P[X_i = 1] = 2\alpha(1-\alpha)$. Similarly $P[Y_j = 1] = 2\alpha(1-\alpha)$. Let $X = \sum_{i=1}^{k} X_i + \sum_{j=k+1}^{n} Y_j$ denote the number of mispredictions. Then
$E[X] = E[\sum_{i=1}^{k} X_i + \sum_{j=k+1}^{n} Y_j] = \sum_{i=1}^{n} E[X_i] = nP[X_1 = 1] = 2\alpha(1-\alpha)n$.

Let $B^{\text{1-bit}}(n)$ denote the expected number of branch mispredictions when random pivot is used. Then we obtain the recurrence
$B^{\text{1-bit}}(n) \leq \frac{1}{n}\sum_{\alpha n=1}^{n}\Big(B^{\text{1-bit}}(\alpha n-1)+B^{\text{1-bit}}((1-\alpha)n)+2\alpha(1-\alpha)n\Big)$.
This solves to $B^{\text{1-bit}}(n)=\frac{2\ln 2}{3}n\lg n+\mathcal{O}(n)$.

Now let $A^{\text{1-bit}}(n)$ denote the expected number of branch mispredictions when an α-skewed pivot is used. Then
$A^{\text{1-bit}}(n) \leq 2\alpha(1-\alpha)n + A^{\text{1-bit}}(\alpha n-1)+A^{\text{1-bit}}((1-\alpha)n)$.
It can be shown by induction that it solves to $A^{\text{1-bit}}(n)=\frac{2\alpha(1-\alpha)}{H(\alpha)}+\mathcal{O}(n)$.

3.3 2-Bit Prediction Scheme

We now consider the 2-bit prediction scheme. As stated earlier we assume that an element is smaller than the pivot with probability α independently of the other comparisons. With this simplification, the branch predictor can be modeled as a Markov chain. First consider the predictor of Loop I. Its corresponding Markov chain has four states, each one corresponding to a state of the predictors automaton, see Fig. 1. The transition table where entry P_{kl} represents the probability of going to state l given that we are in state k is as follows.

$$\mathbf{P}=\begin{bmatrix}\alpha & 1-\alpha & 0 & 0\\ \alpha & 0 & 0 & 1-\alpha\\ \alpha & 0 & 0 & 1-\alpha\\ 0 & 0 & \alpha & 1-\alpha\end{bmatrix}$$

Let π_0,π_1,π_2,π_3 denote the stationary probabilities of the Markov chain, i.e., they are the solution to the system $\overrightarrow{\pi}\cdot\mathbf{P}=\overrightarrow{\pi}$ and $\sum_{k=0}^{3}\pi_i=1$. Then $\pi_0=\frac{\alpha^2}{1-\alpha(1-\alpha)}$, $\pi_1=\frac{\alpha^2(1-\alpha)}{1-\alpha(1-\alpha)}$, $\pi_2=\frac{\alpha(1-\alpha)^2}{1-\alpha(1-\alpha)}$ and $\pi_3=\frac{(1-\alpha)^2}{1-\alpha(1-\alpha)}$. One can easily verify this by substitution. Similarly, to the above we obtain the Markov chain corresponding to Loop J. Now, let X_i be the indicator random variable which is 1 if the i-th element of the Loop I causes a branch misprediction and 0 otherwise. Correspondingly we define Y_j for Loop J.

The ith element causes a branch misprediction in the following cases. After having considered element $a[i-1]$ the Markov chain is in state 0 and $a[i]\geq x$, or it is in state 1 and $a[i]\geq x$, or it is in state 2 and $a[i]<x$ or it is in state 3 and $a[i]<x$. Therefore,

$$P[X_i=1]=\pi_0\cdot P[a[i]\geq x]+\pi_1\cdot P[a[i]\geq x]+\pi_2\cdot P[a[i]<x]+\pi_3\cdot P[a[i]<x].$$

By substituting $P[a[i]\geq x]=1-\alpha$ and $P[a[i]<x]=\alpha$ and the values for $\pi_1,\ldots,\pi_3$ we obtain that $P[X_i=1]=\frac{2\alpha^4-4\alpha^3+\alpha^2+\alpha}{1-\alpha(1-\alpha)}$. The same holds for $P[Y_i=1]$. Now let $X=\sum_{i=1}^{k}X_i+\sum_{j=k+1}^{n}Y_j$ be the number of branch mispredictions. Then $E[X]=E[\sum_{i=1}^{k}X_i+\sum_{j=k+1}^{n}Y_j]=\sum_{i=1}^{n}E[X_i]=nP[X_1=1]=\frac{2\alpha^4-4\alpha^3+\alpha^2+\alpha}{1-\alpha(1-\alpha)}n$. Let $B^{\text{2-bit}}(n)$ denote the expected number of branch mispredictions of quicksort with random pivot. Then

$B^{2\text{-bit}}(n) \leq \frac{1}{n}\sum_{\alpha n=1}^{n}\Big(B^{2\text{-bit}}(\alpha n-1)+B^{2\text{-bit}}((1-\alpha)n)+\frac{2\alpha^4-4\alpha^3+\alpha^2+\alpha}{1-\alpha(1-\alpha)}n\Big)$. This solves to $B^{2\text{-bit}}(n) = \frac{28\ln 2}{45} n\lg n + \mathcal{O}(n)$.

Now let $A^{2\text{-bit}}(n)$ be the expected number of branch mispredictions of quicksort with α-skewed pivot. Then
$A^{2\text{-bit}}(n) \leq \frac{2\alpha^4-4\alpha^3+\alpha^2+\alpha}{1-\alpha(1-\alpha)}n + A^{2\text{-bit}}(\alpha n-1) + A^{2\text{-bit}}((1-\alpha)n)$, which solves to
$A^{2\text{-bit}}(n) = \frac{2\alpha^4-4\alpha^3+\alpha^2+\alpha}{(1-\alpha(1-\alpha))H(\alpha)} n\lg n + \mathcal{O}(n)$.

4 Experiments

For our experiments we use one of the fastest quicksort implementations std::sort from the STL library included in GCC v3.3. This implementation uses the median of 3 elements as the pivot. We added an implementation of the random pivot and the idealized α-skewed pivot mechanisms. Our inputs are random permutation of the integers in the range $[1,\ldots,n]$. We average over $\max\{100, \lceil 10^7/n \rceil\}$ inputs. Note that with a simple calculation we can obtain the element of rank αn, for a given α. Observe that this makes the cost of finding the pivot element negligible. If the time taken by quicksort is too large, the STL implementation switches to an algorithm of $\mathcal{O}(n\lg n)$ worst case performance. Since we are only interested in quicksort we have removed this switch. In order to be able to use a larger number of α's for the skewed pivot mechanism we changed the threshold of breaking the recursion from 16 to 20 elements. This does not have any significant effects. The STL implementation uses insertion sort for sorting the small instances. The measures in our figures include the cost of the final insertion sort. This changes the cost of all algorithms by the same amount and therefore does not affect our conclusions.

We used the PAPI tool which provides an interface that allows to count several CPU events including the number of branch mispredictions and the number of instructions executed. When not otherwise stated, the experiments are on a 3GHz Pentium 4 Prescott.

Figs. 3, 4 and 5 show a comparison of the random pivot, the median of 3, the exact median or 1/2-skewed and the 1/10-skewed pivoting mechanisms in terms of the execution time, the number of occurring branch mispredictions and the number of instructions executed for different values of n. In Fig. 3 we see that the random pivot algorithm is most of the times a little bit worse than the others. On the other hand the difference is very small and in particular we observe that the curves for the random pivot, the median of 3 and the exact median are very close to each other, in contrast to the common concept that the exact median and the median of 3 should significantly outperform the random pivot. Furthermore, we see that the 1/10-skewed algorithm has a better performance.

In Fig. 4 we see that the random pivot has for most n a smaller number of branch mispredictions compared to the median of 3 and the exact median. The measured prediction quality is better than the quality we would expect for a 1-bit predictor (see Table 1) but not quite as good as to be expected for a 2-bit predictor. The 1/10-skewed pivot algorithm has of course the smallest number

of branch mispredictions. In Fig. 5 we see the number of instructions that are executed. These are proportional to the number of comparisons and therefore we see that the exact median is the best, followed by the median of 3, then the random pivot and finally the 1/10-skewed pivot. Observe that the curves in this figure are very flat and smooth in contrast to the curves in Fig. 3. Therefore, it is not only the number of executed instructions that plays a major role in the running time. The fluctuations in Fig. 3 indicate architectural effects. Observe that for $n = 2^{16}$ the number of branch mispredictions of random pivot drop and for this n we also see a significant drop in its running time. Having a closer look at the curves we see that the curves of time and those of the branch mispredictions have the same shape, in the sense that when the branch mispredictions drop, the running time drops too and when the branch mispredictions increase the running time increases too. Note that the branch mispredictions only slowly approach

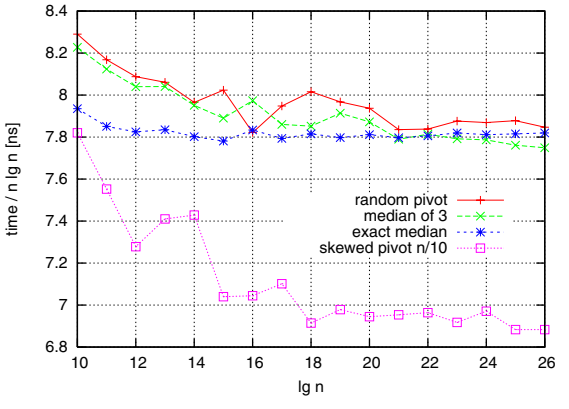


Fig. 3. Time / $n \lg n$ for random pivot, median of 3, exact median, 1/10-skewed pivot

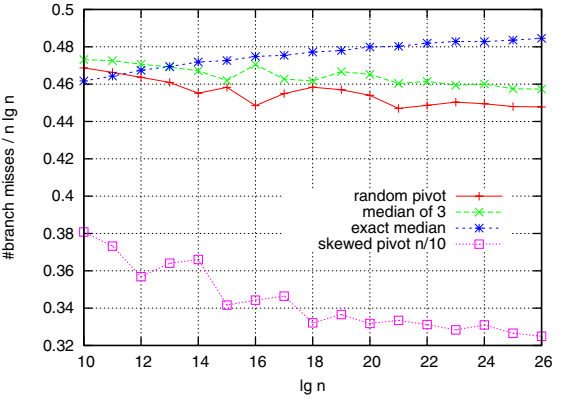


Fig. 4. Number of branch mispredictions / $n \lg n$ for random pivot, median of 3, exact median, 1/10-skewed pivot

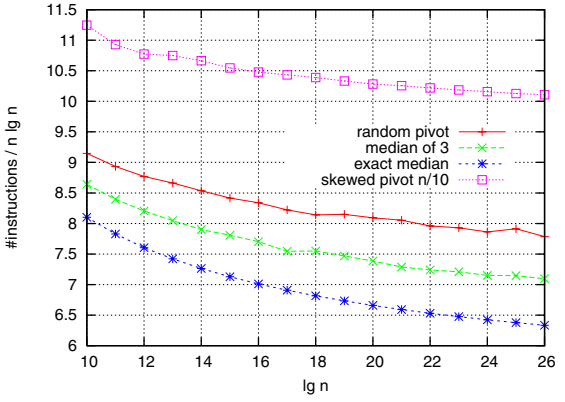


Fig. 5. Number of instructions / $n \lg n$ for random pivot, median of 3, exact median, 1/10-skewed pivot

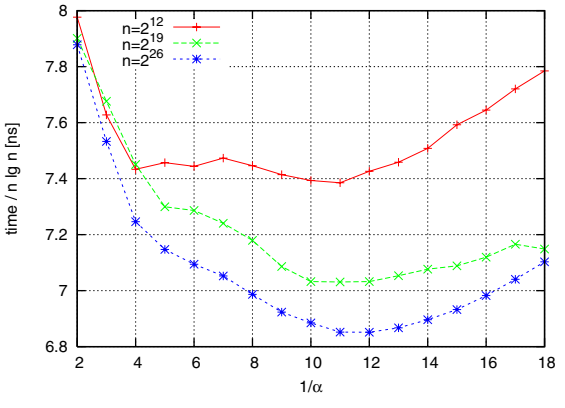


Fig. 6. Time / $n \lg n$ for different values of α

$0.5n \lg n$ for the exact median algorithm. The main reason is that the insertion sort used for small subproblems incurs only $\mathcal{O}(n)$ branch mispredictions (each iteration of the inner loop of insertion sort incurs just one branch misprediction).

Figs. 6, 7 and 8 show the performance of the α-skewed pivot when we vary α. We tried three different values of n, i.e. $2^{12}, 2^{19}$ and 2^{26}. In Fig. 6, where the running time is measured, we see that we have a parabola like figure and for $\alpha = 1/11$ we get the best running time. Moreover, the exact median which is for $\alpha = 1/2$ is a lot worse. Figs. 7 and 8 indicate why we have such a shape in Fig. 6. As α increases, the number of branch mispredictions decreases and the number of instructions increases. Therefore, we see that $\alpha = 1/11$ is the place of compromise.

In order to see the effects of different architectures we reran the experiments on an Athlon, an Opteron and a Sun machine (Figures will be in the full paper). We see for large inputs, pivots close to the median *are* an advantage. Our inter-

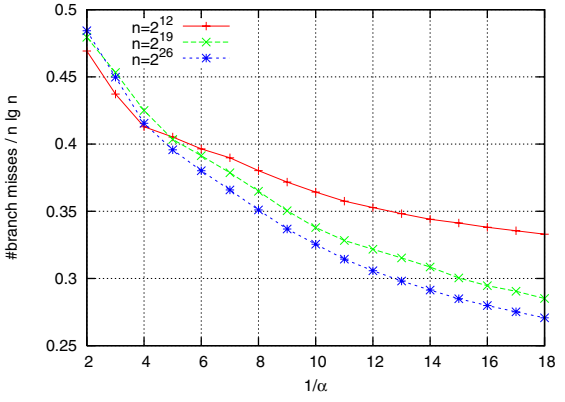


Fig. 7. Number of branch mispredictions / $n \lg n$ for different values of α

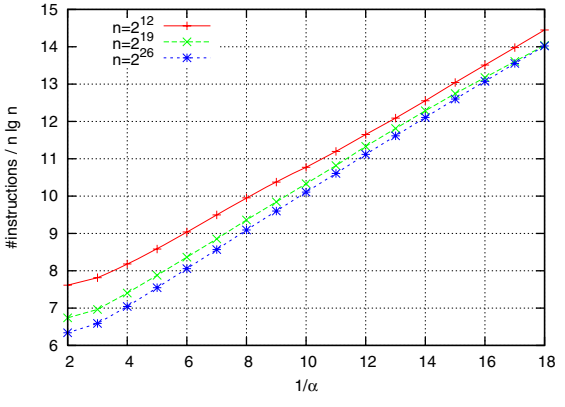


Fig. 8. Number of instructions / $n \lg n$ for different values of α

pretation is that on the Opteron, memory bandwidth is more of an issue than on the Pentium 4 architecture (perhaps its long pipelines make branch misprediction more predominant). Hence, for a skewed pivot algorithm one might want to pick α close to 1/2 for large subproblems but use a smaller value when a subproblem fits in cache. A similar strategy might be useful on a Pentium 4, when we sort larger objects. Since our goal is understanding branch mispredictions rather than designing an efficient algorithm, we do not dwell on this issue.

5 Conclusions

Somewhat astonishingly, generally accepted "improvements" of quicksort such as median-of-three pivot selection bring no significant benefits in practice (at least for sorting small objects) because they increase the number of branch mispredictions. For teaching this means that we should either stop after random

pivots or give the full story of what happens for more sophisticated pivot selection strategies. By actively choosing a skewed pivot, we can slightly improve the performance of quicksort. Since this increases the instruction count, the better approach seems to be to avoid branch mispredictions altogether, e.g. using the techniques described in [4]. However, an in-place sorting algorithm that is better than quicksort with skewed pivots is an open problem.

Acknowledgments. We would like to thank Roman Dementiev, Dimitrios Michail and Johannes Singler for crucial assistance with the experiments.

References

1. Hoare, C.A.R.: Algorithm 64: Quicksort. Commun. ACM **4**(7) (1961) 321
2. Knuth, D.E.: The Art of Computer Programming—Sorting and Searching. Volume 3. Addison Wesley (1973)
3. Martínez, C., Roura, S.: Optimal sampling strategies in Quicksort and Quickselect. SIAM Journal on Computing **31**(3) (2002) 683–705
4. Sanders, P., Winkel, S.: Super scalar sample sort. In: 12th European Symposium on Algorithms (ESA). Volume 3221 of LNCS., Springer (2004) 784–796
5. Brodal, G.S., Fagerberg, R., Moruz, G.: On the adaptiveness of quicksort. In: Workshop on Algorithm Engineering & Experiments, SIAM (2005) 130–149
6. Brodal, G.S., Moruz, G.: Tradeoffs between branch mispredictions and comparisons for sorting algorithms. In: Proc. 9th International Workshop on Algorithms and Data Structures. Volume 3608 of Lecture Notes in Computer Science., Springer Verlag, Berlin (2005) 385–395
7. Patterson, D.A., Hennessy, J.L.: Computer Architecture: A Quantitative Approach 3rd. ed. Morgan Kaufmann (2003)
8. Sedgewick, R.: Algorithms (Second Edition). Addison-Wesley Longman Publishing Co. (1988)

Robust, Generic and Efficient Construction of Envelopes of Surfaces in Three-Dimensional Spaces*

Michal Meyerovitch**

School of Computer Science
Tel Aviv University
gorgymic@post.tau.ac.il

Abstract. Lower envelopes are fundamental structures in computational geometry, which have many applications, such as computing general Voronoi diagrams and performing hidden surface removal in computer graphics. We present a generic, robust and efficient implementation of the divide-and-conquer algorithm for computing the envelopes of surfaces in $\mathbb{R}^3$. To the best of our knowledge, this is the first exact implementation that computes envelopes in three-dimensional space. Our implementation is based on CGAL (the Computational Geometry Algorithms Library) and is designated as a CGAL package. The separation of topology and geometry in our solution allows for the reuse of the algorithm with different families of surfaces, provided that a small set of geometric objects and operations on them is supplied. We used our algorithm to compute the lower and upper envelope for several types of surfaces. Exact arithmetic is typically slower than floating-point arithmetic, especially when higher order surfaces are involved. Since our implementation follows the exact geometric computation paradigm, we minimize the number of geometric operations, and by that significantly improve the performance of the algorithm in practice. Our experiments show interesting phenomena in the behavior of the divide-and-conquer algorithm and the combinatorics of lower envelopes of random surfaces.

1 Introduction

Lower envelopes are fundamental structures in computational geometry, which have many applications. Let $\mathcal{S} = \{s_1, \ldots, s_n\}$ be a collection of n (hyper)surface patches in $\mathbb{R}^d$. Let $x_1, \ldots, x_d$ denote the axes of $\mathbb{R}^d$, and assume that each s_i is monotone in $x_1, \ldots, x_{d-1}$, namely every line parallel to the x_d-axis intersects s_i in at most one point. Regard each surface patch s_i in $\mathcal{S}$ as the graph of a

* This work has been supported in part by the IST Programme of the EU as Shared-cost RTD (FET Open) Project under Contract No IST-006413 (ACS - Algorithms for Complex Shapes), and by the Hermann Minkowski–Minerva Center for Geometry at Tel Aviv University.

** This work is part of the author's M.Sc. thesis under the guidance of Prof. Dan Halperin.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 792–803, 2006.

partially defined $(d-1)$-variate function $s_i(\bar{x})$. The *lower envelope* $\mathcal{E}_{\mathcal{S}}$ of $\mathcal{S}$ is the pointwise minimum of these functions: $\mathcal{E}_{\mathcal{S}}(\bar{x}) = \min\ s_i(\bar{x}),\ \bar{x} \in \mathbb{R}^{d-1}$, where the minimum is taken over all functions defined at $\bar{x}$. Similarly, the *upper envelope* of $\mathcal{S}$ is the pointwise maximum of these functions.

The *minimization diagram* $\mathcal{M}_{\mathcal{S}}$ of $\mathcal{S}$ is the subdivision of $\mathbb{R}^{d-1}$ into maximal connected cells such that $\mathcal{E}_{\mathcal{S}}$ is attained by a fixed subset of functions over the interior of each cell. In the same manner, the *maximization diagram* of $\mathcal{S}$ is the subdivision of $\mathbb{R}^{d-1}$ induced by the upper envelope of $\mathcal{S}$.

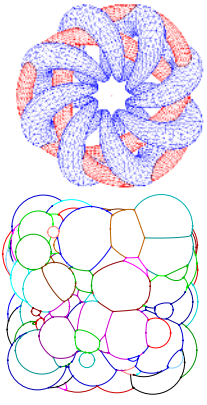
The complexity of the lower envelope of a set of surfaces is defined as the complexity of its minimization diagram. The maximum combinatorial complexity of the lower envelope of n x-monotone Jordan arcs in the plane such that each pair intersects in at most s points, for some fixed constant s, is near linear [22]. The combinatorial complexity of the lower envelope of a set $\mathcal{S}$ of n *well-behaved* (see e.g., [3]) surface patches in $\mathbb{R}^d$ is $O(n^{d-1+\varepsilon})$, for any $\varepsilon > 0$, where the constant of proportionality depends on ε, d and some surface-specific constants [12, 21].

In this paper we deal only with envelopes in $\mathbb{R}^3$. Many algorithms for computing envelopes in three-space exist. Agarwal et al. [2] presented a divide-and-conquer algorithm with running time $O(n^{2+\varepsilon})$, for any $\varepsilon > 0$. This time bound is based on their result that the combinatorial complexity of the overlay of two minimization diagrams of two collections of a total of n surface patches is also $O(n^{2+\varepsilon})$, for any $\varepsilon > 0$. Boissonnat and Dobrindt presented a randomized incremental algorithm with the same running-time bound [7]. Another randomized incremental approach [18] leads to a "quasi output sensitive" algorithm whose expected running time is a sum of weights associated with all intersections of projected objects edges, where the weight of an intersection is inversely proportional to the number of objects "hiding" that intersection from the viewing point. Obtaining output sensitive algorithms that compute envelopes is a major challenge. Such algorithms exist for special cases only. de Berg et al. [9] presented an output-sensitive algorithm for polyhedral objects. Using the data structure of Agarwal and Matoušek [1] its running time is $O(n^{2/3+\varepsilon}k^{2/3})$ for any $\varepsilon > 0$, where k is the output size and n is the number of faces of the polyhedra. Katz et al. [14] presented an output-sensitive algorithm which runs in time $O((U(n)+k)\log^2 n)$, where k is the complexity of the output map and $U(n)$ is a super-additive bound on the maximal complexity of the union of the projections on the viewing plane of any n objects. The method assumes that the surfaces can be ordered by depth from the viewing point and its efficiency shows up when $U(n)$ is small. For a comprehensive summary of results see [8].

Transforming a geometric algorithm into a computer program is not a simple task. An algorithm implemented from a textbook is susceptible to robustness issues, and thus may yield incorrect results, enter infinite loops or crash (see e.g., [15, 20]). This is mainly due to two assumptions that are often made in the theoretical study of geometric algorithms, which are not realistic in practice. First, the general position assumption excludes all degenerate inputs. Secondly, the real RAM model is assumed, which allows for infinite precision arithmetic operations on real numbers. Moreover, every operation on a constant number of

simple geometric objects is assumed to take constant time. This is of course not true in computer programs that use finite precision numbers.

CGAL, the Computational Geometry Algorithms Library,[1] is a product of a collaborative effort of several sites in Europe and Israel, aiming to provide a robust, generic and efficient implementation of geometric data structures and algorithms. It is a software library written in C++ following the generic programming paradigm. The *arrangement* of a set $\mathcal{C}$ of planar curves is the subdivision of the plane induced by the curves in $\mathcal{C}$ into maximally connected cells. CGAL provides a robust implementation for constructing planar arrangements of arbitrary bounded curves and supporting operations on them [10, 23]. Robustness is achieved both by handling all degenerate cases, and by using exact number types. The CGAL arrangement package contains a class-template, which represents a planar arrangement, and is parameterized by a traits class that encapsulates the geometry of the family of curves it handles. Robustness is guaranteed as long as the traits class uses exact number types for the computations it performs. Among the number-type libraries that are used are GMP[2] for rational numbers, and CORE[3] [13] and LEDA[4] [16, Chapter 4] for algebraic numbers.



We devised an exact and generic implementation of the divide-and-conquer algorithm for constructing the envelope of surface patches in $\mathbb{R}^3$. Our solution is complete in the sense that it handles all degenerate cases, and at the same time it is efficient. To the best of our knowledge, this is the first implementation of this kind. Our implementation is based on the CGAL library and is designated as a CGAL package. The problem of computing the envelope is somewhat two-and-a-half dimensional, since the input is three-dimensional, but the output is naturally represented as a two-dimensional object, the minimization diagram. We use the CGAL two-dimensional arrangement package for the representation of the minimization diagram. The separation of topology and geometry in our solution allows for the reuse of the algorithm with different families of surfaces, provided that a small set of geometric objects and operations on them is supplied. We used our algorithm to compute the lower or upper envelope of sets of triangles, of sets of spheres and of sets of quadratic surfaces [6]. The figures above show examples of minimization diagrams of a set of triangles and of a set of spheres as computed by our program. Our implementation follows the exact geometric computation paradigm. Exact arithmetic is typically slower than floating-point arithmetic, especially when higher order surfaces are involved. One of the main contributions of our work is minimizing the number of geometric operations, and by that significantly improving the performance of

[1] See the CGAL project homepage: `http://www.cgal.org/`.
[2] Gnu's multi-precision library `http://www.swox.com/gmp/`.
[3] `http://www.cs.nyu.edu/exact/core_pages/intro.html`.
[4] `http://www.algorithmic-solutions.com/enleda.htm`.

the algorithm in practice. Our experiments show interesting phenomena in the behavior of the divide-and-conquer algorithm and the combinatorics of lower envelopes of random surfaces. In particular, they show that on some input sets the algorithm performs better than the worst-case bound, and the combinatorial size of the envelope is typically asymptotic much smaller than the worst-case bound.

The rest of this paper is organized as follows. In Sect. 2 we briefly review the divide-and-conquer algorithm. In Sect. 3 we explain how we separate the geometry and topology, and provide the details on the geometric part of our implementation. Section 4 lists the methods we use to reduce the amount of geometric computation and improve the performance of the algorithm. In Sect. 5 we present experimental results and discuss the practical performance of the algorithm. More details on the work can be found in [17]. In the following sections, to simplify the exposition, we refer to *lower* envelopes. However, our code is generic and capable of computing envelopes in any direction, including upper envelopes.

2 The Divide-and-Conquer Algorithm

We are given as input a set $\mathcal{F}$ of n surface patches in $\mathbb{R}^3$. The first step is to extract all the xy-monotone portions of these surfaces that are relevant to the envelope. We denote this set by $\mathcal{G}$. Henceforth, for convenience, we only work on these xy-monotone surfaces in $\mathcal{G}$. (Some of our methods described in Sect. 4 are valid only under the assumption that the surfaces are xy-monotone. Yet, this assumption does not contradict the generality of the algorithm since $\mathcal{E}_{\mathcal{F}} = \mathcal{E}_{\mathcal{G}}$.) The output of our program is a minimization diagram, represented as a planar arrangement where each arrangement feature (vertex, edge or face) is labelled with the set of xy-monotone surfaces that attain the minimum over that feature. The label can contain a single surface, several surfaces, or no surface at all, in which case we call it the no surface label.

When $\mathcal{G}$ consists of a single xy-monotone surface, we construct its minimization diagram using the projection of its boundary. When $\mathcal{G}$ contains more than one xy-monotone surface, we split $\mathcal{G}$ into two sets $\mathcal{G}_1$ and $\mathcal{G}_2$ of (roughly) equal size, recursively construct the minimization diagrams $\mathcal{M}_1$ and $\mathcal{M}_2$ of these sets respectively, and finally merge these two diagrams into the final minimization diagram $\mathcal{M}$.

The merge step is carried out as follows. First, we overlay the two planar arrangements underlying the minimization diagrams $\mathcal{M}_1$ and $\mathcal{M}_2$ to obtain the arrangement $\mathcal{O}$, where each feature is a maximal connected portion of the intersection of one feature f_1 of $\mathcal{M}_1$ and one feature f_2 of $\mathcal{M}_2$. For each feature in $\mathcal{O}$ we keep two pointers to f_1 and f_2.

Next, we determine the structure of the minimization diagram over each feature in $\mathcal{O}$, to obtain the arrangement $\mathcal{O}'$, which is a refinement of the arrangement underlying $\mathcal{M}$. We then label each feature of $\mathcal{O}'$ with the correct envelope surfaces. We should consider here the two relevant features in $\mathcal{M}_1$ and $\mathcal{M}_2$ and their labels l_1 and l_2, respectively. The non-trivial case is when both labels rep-

resent non-empty sets of xy-monotone surfaces. These surfaces are defined over the entire current feature f, and their envelope over f is the envelope of $\mathcal{G}_1 \cup \mathcal{G}_2$ there. Since all the xy-monotone surfaces of one label l_i overlap over the current feature f, it is possible to take only one representative surface s_i from each label and find the shape of their minimization diagram over f. Here we should consider the intersection between the representative surfaces s_1 and s_2, or more precisely, the projection $\mathcal{C}$ (which is, in general, a set of curves) onto the xy-plane of this intersection, which may split the current feature, if it is an edge or a face. We then label all the features of the arrangement of $\mathcal{C}$ restricted to f (where f is considered relatively open) with the correct label, which might be either one of the labels l_1 and l_2, or $l_1 \cup l_2$ in case of an overlap.[5] A face of the overlay handled in this step, and hence the arrangement restricted to the face, can be very complicated, with arbitrary topology (including holes, isolated points and "antennas") and unbounded complexity.

Finally, we apply a cleanup step in order to remove redundant features (edges or vertices) of $\mathcal{O}'$ and obtain the minimization diagram $\mathcal{M}$ of $\mathcal{G}$.

The algorithm that we implemented (and described in detail below) is similar to the one presented by [2] with one main difference — it handles the complex faces directly instead of performing a vertical decomposition to get simple constant-size faces.

3 The Geometric Traits Class

Our algorithm is parameterized with a *traits* class [5, 19], which serves as the geometric interface to the algorithm. The traits class encapsulates the geometric objects the algorithm operates on, and the predicates and constructions on these objects used by the algorithm. In this manner the algorithm is made generic and independent of the specific geometry needed to handle a special type of surfaces.

The set of requirements from a traits class forms a *concept* [5]. The geometric-traits concept for the envelope algorithm refines (extends) the concept for building planar arrangements of general bounded curves. The latter concept, which is described in detail in [23], defines three object types: planar points, x-monotone curves and general curves, and operations on them. The envelope concept adds to these requirements two more object types: three-dimensional xy-monotone surfaces and general surfaces, and the following operations on them:

1. Given a general surface, extract maximal continuous xy-monotone patches of the surface which contribute to its envelope.
2. Construct all the planar curves that form the boundary of the vertical projection of a given xy-monotone surface onto the xy-plane.
3. Construct all the planar curves and points, which compose the projection (onto the xy-plane) of the intersection between two xy-monotone surfaces s_1 and s_2, or return an empty set in case these surfaces do not intersect. If

[5] With a slight abuse of notation we use the label l_i to denote the corresponding set of surfaces.

possible, indicate, for each projected intersection curve, whether the *envelope order* of s_1 and s_2 changes when crossing that curve, or not. The envelope order of s_1 and s_2 indicates whether s_1 is below/coincides with/is above s_2. This information (referred to as the intersection type information) is optional — when provided, it is used to improve performance of the algorithm, as is explained in Sect. 4.

4. Given two xy-monotone surfaces s_1 and s_2, and a planar point p, which lies in their xy definition range, determine the envelope order of s_1 and s_2 at the xy-coordinates of p. This operation is only used in degenerate cases. A situation where this operation is used is illustrated in Fig. 1(a).
5. Given two xy-monotone surfaces s_1 and s_2, and a planar x-monotone curve c, which is a part of their projected intersection, determine the envelope order of s_1 and s_2 immediately above (similarly below) the curve c (in the plane). Note that c is a curve in the plane, and we refer to the region above/below c in the *plane*, here and in the description of Operation 6.
6. Given two xy-monotone surfaces s_1 and s_2, and a planar x-monotone curve c, which lies fully in their common xy range, such that s_1 and s_2 do not intersect over the interior of c, determine the envelope order of s_1 and s_2 in the interior of c. Figure 1(b) illustrates a situation where this operation is used.

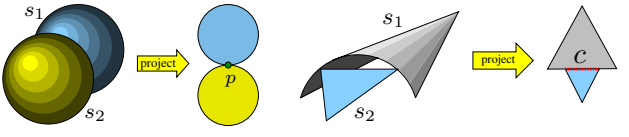


Fig. 1. (a) Two tangent spheres s_1 and s_2 will be compared over the only point p in their common xy-range. (b) The surfaces s_1 and s_2 will be compared over the interior of the line segment c.

Making the traits-class concept as tight as possible, by identifying the minimal number of required methods, is crucial. It can make the whole difference between being able to implement a traits class for a specific type of surfaces or not, and may have a major effect on the efficiency of the algorithm, especially for non-linear objects. This observation has guided us in our design. Our algorithm does not require the traits class to define any three-dimensional types except the surface types. The interface with the traits class contains only the necessary types for the input and the output of the algorithm, and a small set of operations defined on these types.

Our implementation of the divide-and-conquer algorithm is completely independent of the direction in which the envelope is to be computed. The traits is responsible for controlling this direction. All our traits classes support the computation of a lower and an upper envelope. It is also possible to use our algorithm to compute the envelope seen from a direction that is not parallel to the z-axis, provided that the traits-class correctly implements all operations with respect to that direction.

4 Reducing the Number of Algebraic Operations

We invested considerable effort in trying to minimize the amount of algebraic computation whenever possible, as such computation is usually very costly. This actually means substituting calls to the different traits-class methods by the propagation of information (in the form of labels) between incident features.

In the rest of this section we use the following notation. We merge two minimization diagrams $\mathcal{M}_1$ and $\mathcal{M}_2$ (representing the lower envelopes $\mathcal{E}_1$ and $\mathcal{E}_2$ respectively) of two sets of xy-monotone surfaces $\mathcal{G}_1$ and $\mathcal{G}_2$ respectively. The result of the merge is a minimization diagram $\mathcal{M}$ (representing the lower envelope $\mathcal{E}$) of the set $\mathcal{G}_1 \cup \mathcal{G}_2$.

We use two types of information caching to avoid recomputing some geometric information, (for more details on these caches we refer the reader to [17]).

- Cache for projected intersections of pairs of xy-monotone surfaces (since the projected intersection of the same pair of surfaces may arise in the algorithm more than once).
- Cache for comparison results of pairs of disjoint xy-monotone surfaces, where the projection onto the xy-plane of each of the surfaces is convex.

Let $\mathcal{O}'$ be the arrangement, which is created by overlaying the arrangements underlying $\mathcal{M}_1$ and $\mathcal{M}_2$, and determining the shape of the minimization diagram over each feature of the overlay. $\mathcal{O}'$ is a refinement of the arrangement underlying $\mathcal{M}$. For each feature f of $\mathcal{O}'$ the envelope $\mathcal{E}$ is attained by the same set of surfaces over all points of f. $\mathcal{O}'$ is the input to the labelling step, which determines the correct label of all the features of $\mathcal{O}'$. For each feature f of $\mathcal{O}'$ we have to decide between two labels from the two minimization diagrams currently being merged; we say that f is associated with a *decision*. A *decision* can be one of three values: first, when the feature should be labelled with all the surfaces of the first label, second, when the feature should be labelled with all the surfaces of the second label and both, when the surfaces of both labels overlap over the feature. We work with decisions in the labelling step, and after the cleanup step, we translate the decisions into the relevant labels.

Obviously, in order to make a decision for a feature, we can use one of the three types of comparison operations described in Sect. 3. However, we can use the following observations to significantly reduce the number of such operations. These savings are demonstrated in the experiments reported in Sect. 5.

- No need to compare xy-monotone surfaces over their projected intersection, since they are equal there.
- Information on intersection type can be used when available, to avoid the comparison of two xy-monotone surfaces on one side of a curve (which is a part of their projected intersection) if their envelope order on the other side of that curve is known. Recall that such information is (optionally) given by the traits operation which constructs the projected intersection of two xy-monotone surfaces. Similar information is extremely helpful in constructing two-dimensional arrangements of curves [11].

– Information on the continuity or discontinuity of the two envelopes currently being merged can be used in order to conclude the decision for a feature from a decision on an incident feature. We regard the following as incident features: (i) a face and an edge on its boundary, (ii) a face and a vertex on its boundary, and (iii) an edge and its endpoint vertices.

Using continuity or discontinuity information, as we explain in the rest of this section, it is possible to carry a decision over from a face to a boundary edge and vice versa. To best exploit this property, we traverse the faces in a breadth-first order, moving from a face to its neighboring faces.

Using Continuity or Discontinuity Information

We consider the xy-monotone surfaces as graphs of partially defined bivariate continuous functions, and their envelope as a function defined over the entire xy-plane. In addition, we consider the boundary of an xy-monotone surface to be part of this surface. For lack of space, we omit proofs here; the interested reader can find them in [17].

Definition 1. *Let $\mathcal{E}$ be an envelope and $\mathcal{M}$ its minimization diagram. Let f and e be two incident features of $\mathcal{M}$, such that e lies on the boundary of f. We say that $\mathcal{E}$ meets f and e continuously if $\mathcal{E}$ restricted to $f \cup e$ is continuous over e.*

Observation 1. *Let $\mathcal{E}$ be an envelope of a set $\mathcal{S}$ of surfaces and $\mathcal{M}$ its minimization diagram. Let f and e be two incident features of $\mathcal{M}$, such that e lies on the boundary of f. $\mathcal{E}$ meets f and e continuously if and only if there exists an xy-monotone surface $s \in \mathcal{S}$ which appears over f and e on the envelope $\mathcal{E}$.*

We use this observation in our implementation to decide where an envelope meets two incident features continuously. This information in then used to reduce the number of geometric comparisons according to the following lemma.

Lemma 1. *Let f be a face of $\mathcal{O}'$ and e be an edge on its boundary. Suppose that: (i) a decision over the face f is known, and (ii) both envelopes $\mathcal{E}_1$ and $\mathcal{E}_2$ meet f and e continuously. Let s_1 and s_2 be the xy-monotone surfaces that appear over f and e on these envelopes respectively. Suppose further that e is not part of the projected intersection of the surfaces s_1 and s_2. Then the decision made on f is valid also on e. Similar statements can be used for other types of incident features.*

Sometimes, we can use also the *discontinuity* information to deduce a decision for a feature without comparing the relevant surfaces.

Observation 2. *Let $\mathcal{E}$ be a lower envelope and $\mathcal{M}$ its minimization diagram. Let f and e be two incident features of $\mathcal{M}$, such that e lies on the boundary of f. Let s_f and s_e be representative xy-monotone surfaces of $\mathcal{E}$ over f and e respectively. $\mathcal{E}$ does not meet f and e continuously if and only if s_e lies below s_f over e (note that s_f is defined over e).*

Applying Observation 2 to the labelling step we get:

Lemma 2. *Let f be a face of $\mathcal{O}'$ and e be an edge on its boundary. Assume that the lower envelope $\mathcal{E}_1$ meets f and e continuously, but the lower envelope $\mathcal{E}_2$ does not. (i) If $\mathcal{E}_2$ is below $\mathcal{E}_1$ over f, then $\mathcal{E}_2$ is below $\mathcal{E}_1$ also over e. (ii) If $\mathcal{E}_1$ is below $\mathcal{E}_2$ over e, then $\mathcal{E}_1$ is below $\mathcal{E}_2$ also over f. Similar arguments apply to all other incidence relationships.*

Figure 2 illustrates how the observations above are used in the labelling step. Table 2 in Sect. 5 demonstrates the significant savings in geometric operations by the techniques presented above. For example, for a set of 1000 random triangles the total number of comparison operations reduces from 265,211 to 13,620, and for a set of 1000 random spheres the total number of such operations reduces from 48,842 to 2,457, which is roughly a saving of 95% in either case.

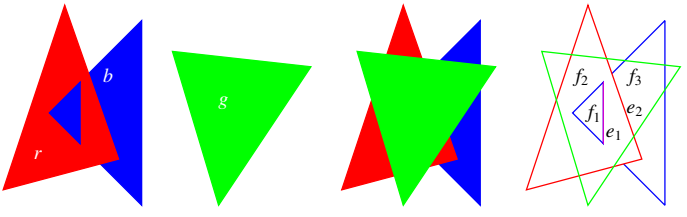


Fig. 2. Applying continuity or discontinuity arguments to carry on a decision between incident features: (a) the envelope of two triangles r and b, (b) the envelope of one triangle g, (c) the envelope of r, b and g, (d) the overlaid arrangement before the labelling step. To label face f_1 with g we compare triangles b and g. Using Lemma 1, we label all the features on the boundary of f_1 with g. To label face f_2, where triangles r and g should be compared, we use Lemma 1 and the edge e_1 to conclude that g appears on the envelope, without actually comparing r and g. Using Lemma 1, we label all the features on the boundary of f_2 with g. Finally, we use Lemma 2 and the edge e_2 to set the label of face f_3 to g. To summarize, we need only compare triangles b and g once, and all the other decisions follow.

5 Experimental Results

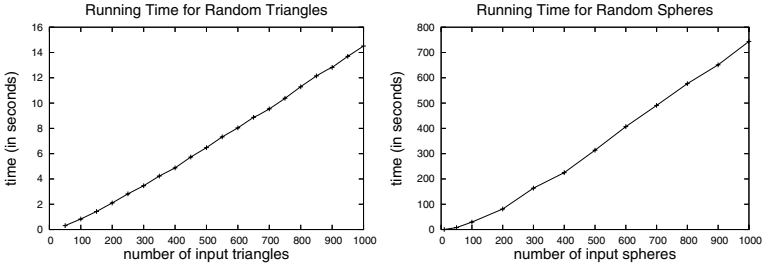
In this section we present experimental results that demonstrate the performance of our algorithm and show its behavior on various input sets. Many more experimental results are available in [17]. The running times reported in this section were obtained on a single 3 GHz PC with 2 Gb of RAM, running under Linux.

Our experiments were conducted on the following input sets:

- `rnd_triangles_`n n triangles, each of which was generated by choosing the coordinates of its three vertices uniformly at random as integers in the cube $[0, 10000]^3$.
- `rnd_small_`p`_triangles_`n n triangles, each of which was generated by first choosing the coordinates of one corner of the triangle uniformly at randomin

the cube $[0, 10000]^3$, then choosing two random points in the sphere with radius $p * 10000$ around this point. All the vertex coordinates are integers.
- `rnd_small_spheres_`n n spheres, where the centers were chosen with integer coordinates uniformly at random in the range $[-1000, 1000]^3$, and the integer radii were chosen uniformly at random in the range $[1, 250]$.
- `rnd_spheres_`n n spheres, where the centers were chosen with integer coordinates uniformly at random in the range $[-1000, 1000]^3$, and the integer radii were chosen uniformly at random in the range $[1, 500]$.

We measured the running time of our algorithm on various types of examples, to investigate the behavior of the algorithm in practice. Some of the results are shown in Fig. 3. Our results show that on some inputs sets the algorithm performs better than the worst-case estimate.



(a) (b)

Fig. 3. The running time of computing the envelope for different input sizes: (a) `rnd_triangles_`n, (b) `rnd_spheres_`n

We investigated the behavior of the size of the lower envelope of specific input sets; results graphs are available in [17]. For the `rnd_triangles_n` input sets, the results show that the size of the minimization diagram is roughly[6] $\Theta(n^{2/3})$. For other input sets the minimization diagrams are sub-linear in the input size as well.

Table 1 shows the actual running time for different input sets, each consisting of 1000 input surfaces. In the last three columns we give statistics of the whole process, which can give an idea about the amount of work that is carried out during the whole execution. It can be seen that the algorithm is much slower when run on non-linear input than on linear input; this is expected when using exact arithmetic, since with linear input, rational arithmetic suffices, whereas with non-linear input, algebraic numbers should be used. For lack of space we omit the statistics on the size of the minimization diagrams, which demonstrate the huge variance in output size for the same (combinatorial) input size; they can be found in [17].

Table 2 shows the algorithm running time and the number of calls to the three types of comparison methods made by the algorithm in the labelling process,

[6] This bound of roughly $\Theta(n^{2/3})$ is inspired by recent results of Alon et al. [4] for envelopes of segments in the plane. It seems that some of their results extend to 3D implying this bound.

Table 1. Results for different input sets. *Intermediate* is the total sum of the combinatorial size of all the minimization diagrams computed during the recursion. *Intersections* is the number of intersections between pairs of surfaces that were found by the algorithm. *2d-Intersections* is the number of two-dimensional intersections between projected x-monotone curves that were found during the entire run of the algorithm.

Input File (1000 surfaces)	Time (seconds)	Process details:		
		Intermediate	Intersections	2d-Intersections
rnd_triangles	14.073	190,942	12,007	52,990
rnd_small_0.1_triangles	2.369	94,906	117	11,134
rnd_small_0.5_triangles	6.532	144,383	2,676	29,093
rnd_small_spheres	249.111	60,465	842	7,472
rnd_spheres	654.044	53,188	1,565	8,547

comparing between the naïve approach and our approach. The naïve approach means comparing surfaces over all features, except over projected intersections. Our approach is described in Sect. 4, and uses the intersection type and continuity/discontinuity information together with a breadth-first traversal of the faces. Both approaches use the caching information described in Sect. 4. It can be seen that the reduction in the number of operations is highly significant for all the input sets. We remark that the number of comparison operations over a two-dimensional point reduces to zero in our approach since these examples do not contain degeneracies in which this operation is invoked.

Table 2. Comparing the number of comparison operations used by the algorithm with and without our means for reducing the number of algebraic operations. The *Pt., Cv. and Cv.-side* columns represent the number of calls made to the appropriate version of the comparison method: comparison over a planar point, comparison over a planar x-monotone curve and comparison above/below a planar x-monotone curve respectively. The construction time is given in seconds.

Input File (1000 surfaces)	Naïve solution				Using our improvements			
	Pt.	Cv.	Cv.-side	Time	Pt.	Cv.	Cv.-side	Time
rnd_triangles	85,091	166,405	13,715	25.028	0	10,333	3,287	14.073
rnd_small_0.3_triangles	42,090	76,763	1,934	6.861	0	8,244	791	5.263
rnd_small_0.5_triangles	51,598	96,662	3,703	9.593	0	8,851	1,325	6.532
rnd_small_spheres	14,901	25,853	1297	399.327	0	2,466	450	249.111
rnd_spheres	14,965	25,630	2247	1116.430	0	1,840	617	654.044

References

1. P. K. Agarwal and J. Matoušek. Ray shooting and parametric search. *SIAM J. Comput.*, 22(4):794–806, 1993.
2. P. K. Agarwal, O. Schwarzkopf, and M. Sharir. The overlay of lower envelopes and its applications. *Discrete Comput. Geom.*, 15:1–13, 1996.

3. P. K. Agarwal and M. Sharir. Arrangements and their applications. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 49–119. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 2000.
4. N. Alon, D. Halperin, O. Nechushtan, and M. Sharir. The complexity of the outer face in arrangements of random segments. Manuscript, 2006.
5. M. H. Austern. *Generic Programming and the STL*. Addison-Wesley, 1999.
6. E. Berberich and M. Meyerovitch. Computing envelopes of quadrics. In preparation.
7. J.-D. Boissonnat and K. T. G. Dobrindt. On-line construction of the upper envelope of triangles and surface patches in three dimensions. *Comput. Geom. Theory Appl.*, 5(6):303–320, 1996.
8. M. de Berg. *Ray Shooting, Depth Orders and Hidden Surface Removal*, volume 703 of *LNCS*. Springer-Verlag, 1993.
9. M. de Berg, D. Halperin, M. Overmars, J. Snoeyink, and M. van Kreveld. Efficient ray shooting and hidden surface removal. *Algorithmica*, 12:30–53, 1994.
10. E. Flato, D. Halperin, I. Hanniel, O. Nechushtan, and E. Ezra. The design and implementation of planar maps in CGAL. *J. of Experim. Alg.*, 5:1–23, 2000.
11. E. Fogel et al. An empirical comparison of software for constructing arrangements of curved arcs. Technical Report ECG-TR-361200-01, Tel-Aviv Univ., 2004.
12. D. Halperin and M. Sharir. New bounds for lower envelopes in three dimensions, with applications to visibility in terrains. *Discrete Comput. Geom.*, 12:313–326, 1994.
13. V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A core library for robust numeric and geometric computation. In *Proc. Symp. on Computational Geometry 1999*, pages 351–359, 1999.
14. M. J. Katz, M. H. Overmars, and M. Sharir. Efficient hidden surface removal for objects with small union size. *Comput. Geom. Theory Appl.*, 2:223–234, 1992.
15. L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C. Yap. Classroom examples of robustness problems in geometric computations. In *Proc. European symp. on Algorithms 2004*, volume 3221 of *LNCS*, pages 702–713, 2004.
16. K. Mehlhorn and S. Näher. LEDA*: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, UK, 2000.
17. M. Meyerovitch. Robust, generic and efficient construction of envelopes of surfaces in three-dimensional space. M.Sc. thesis, School of Computer Science, Tel Aviv University, Tel Aviv, Israel, July 2006.
18. K. Mulmuley. An efficient algorithm for hidden surface removal, II. *J. Comput. Syst. Sci.*, 49(3):427–453, 1994.
19. N. Myers. "Traits": A new and useful template technique. In S. B. Lippman, editor, *C++ Gems*, volume **5** of *SIGS Reference Library*, pages 451–458. 1997.
20. S. Schirra. Robustness and precision issues in geometric computation. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 597–632. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 1999.
21. M. Sharir. Almost tight upper bounds for lower envelopes in higher dimensions. *Discrete Comput. Geom.*, 12:327–345, 1994.
22. M. Sharir and P. K. Agarwal. *Davenport-Schinzel Sequences and Their Geometric Applications*. Cambridge University Press, Cambridge-New York-Melbourne, 1995.
23. R. Wein, E. Fogel, B. Zukerman, and D. Halperin. Advanced programming techniques applied to CGAL's arrangement package. In *Proc. Library-Centric Software Design 2005*, 2005.

Engineering Highway Hierarchies*

Peter Sanders and Dominik Schultes

Universität Karlsruhe (TH), 76128 Karlsruhe, Germany
{sanders, schultes}@ira.uka.de

Abstract. In [1], we presented a shortest path algorithm that allows fast point-to-point queries in graphs using preprocessed data. Here, we give an extensive revision of our method. It allows faster query and preprocessing times, it reduces the size of the data obtained during the preprocessing and it deals with directed graphs. Some important concepts like the *neighbourhood radii* and the *contraction of a network* have been generalised and are now more flexible. The query algorithm has been simplified: it differs only by a few lines from the bidirectional version of DIJKSTRA's algorithm. We can prove that our algorithm is correct even if the graph contains several paths of the same length.

Experiments with real-world road networks confirm the effectiveness of our approach. Preprocessing the network of Western Europe, which consists of about 18 million nodes, takes 15 minutes and yields 68 bytes of additional data per node. Then, random queries take 0.76 ms on average. If we are willing to accept slower query times (1.38 ms), the memory usage can be decreased to 17 bytes per node. For the European and the US road networks, we can guarantee that at most 0.05% of all nodes are visited during any query.

1 Introduction

Computing fastest routes in road networks is one of the showpieces of real-world applications of algorithmics. In principle we could use DIJKSTRA's algorithm. But for large road networks this would be far too slow. Therefore, there is considerable interest in speedup techniques for route planning. Commercial systems use information on road categories to speed up search. "Sufficiently far away" from source and target, only "important" roads are used. This requires manual tuning of the data and a delicate tradeoff between computation speed and suboptimality of the computed routes. In a previous paper [1] we introduced the idea to *automatically* compute *highway hierarchies* that yield *optimal* routes *uncompromisingly quickly*. This was the first speedup technique that was able to preprocess the road network of a continent in realistic time and obtain large speedups (several thousands) over DIJKSTRA's algorithm. Since this was a prototype, we made several simplifying assumptions. Our system was limited to undirected graphs, we only had a proof for a simplified version of the query algorithm, practitioners criticised the considerable constant factor in space consumption, and the query algorithm was fairly complicated.

* Partially supported by DFG grant SA 933/1-3.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 804–816, 2006.

In this paper we tackle all these issues. We originally thought that this would be a more or less routine case study in algorithm engineering. However, we arrived at some algorithmically interesting new and more general concepts and we obtained results we did not expect. In particular, our system became at the same time considerably simpler, more space efficient, and faster with respect to both preprocessing and query time.

Our Contributions. Perhaps the most crucial definition for highway hierarchies is a specification of the concept of *local search*. Section 3 allows directed graphs and an individual neighbourhood radius for each node. The highway network—a set of edges that suffice for all shortest paths outside of local neighbourhoods—can then still be computed using methods analogous to [1]. Since the highway network is very sparse, it is then important to *contract* it by removing nodes of small degree. In [1] this was done using specialised routines for *attached trees* and *lines* of nodes with degree two. First experiments indicated that a straight forward adaptation of these concepts to directed graphs leads to deteriorating performance. In Section 4 we describe a simpler, more general method that leads to *better* performance even for undirected graphs: A node v can be *bypassed* by replacing all edge pairs of the form $(u, v), (v, w)$ with $u \neq w$ by a *shortcut* (u, w). For a tuning parameter c, if the number of introduced shortcuts is smaller than c times the number of removed edges adjacent to v, the node is actually bypassed.

Section 5 describes a simple query algorithm for directed highway hierarchies. Its pseudocode is only four lines longer than code for ordinary bidirectional Dijkstra. Since highway hierarchies are additionally similar to the heuristic hierarchies used in industry, we are very optimistic that they are easy to use in products. Moreover, the simplified algorithm also allows us to give a complete correctness proof.

Section 6 deals with the abort criterion that can be applied when forward and backward search have met. In contrast to the undirected prototype from [1], we *drop* optimisations intended for pruning the search space. While this inflates the search space by about 50%, our measurements indicate that the net effect on the running time is an improvement by more than 50%. Furthermore, we describe how an additional acceleration can be obtained by computing a complete distance table for the topmost level of the highway hierarchy.

The experiments in Section 7 give a strong indication that directed highway hierarchies are currently the most efficient technique for route planning. The tuning parameters turn out to work uniformly well for all the inputs or at least suboptimal values only lead to small performance degradations. Highway hierarchies also allow per-instance worst case performance guarantees, i.e., we can give a good approximation of the complete query time distribution including the worst case for all n^2 possible query pairs without actually executing this astronomic number of queries.

Related Work. For a detailed review of practical and theoretical speedup techniques we refer to [2, 3, 4]. Here we restrict ourselves to the latest news and the concepts needed to understand the problems at hand. A classical technique is *bidirectional search*, which simultaneously searches forward from s and back-

wards from t until the search frontiers meet. For the remaining techniques, we distinguish between two basic speedup effects. Some techniques direct the search towards the target node (and backward search towards the source node), other approaches exploit the *hierarchy* inherent in road networks, and some incorporate both effects by storing information about nodes reached by shortest paths via some edge. Besides highway hierarchies, the most effective hierarchy based technique is *reach based routing* [5] which was considerably strengthened in [6]. Interestingly, the methods used to efficiently compute highway hierarchies in [1] also turned out to be crucial for computing reaches. Reach based routing combined with the strong sense of goal direction from the *landmark method* (the REAL algorithm) beats [1] with respect to query time whereas it needs significantly more preprocessing time. Our new results achieve better query times, a factor ≥ 26 smaller preprocessing times, and need less space.

2 Preliminaries

Graphs and Paths. We expect a *directed* graph $G = (V, E)$ with n nodes and m edges (u, v) with *nonnegative* weights $w(u, v)$ as input. We assume w.l.o.g. that there are no self-loops, parallel edges, and 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. $P^* = \langle s, \ldots, t\rangle$ 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 $P = \langle s, \ldots, s', u_1, u_2, \ldots, u_k, t', \ldots, t\rangle$ is a path from s to t, then $P|_{s'\to t'} = \langle s', u_1, u_2, \ldots, u_k, t'\rangle$ denotes the *subpath* of P from s' to t'.

Dijkstra*'s Algorithm.* Dijkstra's algorithm can be used to solve the *single source shortest path (SSSP) problem*, i.e., to compute the shortest paths from a single source node s to all other nodes in a given graph. It is covered by virtually any textbook on algorithms, so that we confine ourselves to introducing our terminology: Starting with the source node s as root, Dijkstra's algorithm grows a *shortest path tree* that contains shortest paths from s to all other nodes. During this process, each node of the graph is either *unreached*, *reached*, or *settled*. A node that already belongs to the tree is *settled*. If a node u is settled, a shortest path P^* from s to u has been found and the distance $d(s, u) = w(P^*)$ is known. A node that is adjacent to a settled node is *reached*. Note that a settled node is also reached. If a node u is reached, a path P from s to u, which might not be the shortest one, has been found and a *tentative distance* $\delta(u) = w(P)$ is known. Nodes that are not reached are *unreached*.

A *bidirectional* version of Dijkstra's algorithm can be used to find a shortest path from a given node s to a given node t. Two Dijkstra searches are executed in parallel: one searches from the source node s in the original graph $G = (V, E)$, also called *forward graph* and denoted as $\overrightarrow{G} = (V, \overrightarrow{E})$; another searches from the target node t backwards, i.e., it searches in the *reverse graph* $\overleftarrow{G} = (V, \overleftarrow{E})$, $\overleftarrow{E} := \{(v, u) \mid (u, v) \in E\}$. The reverse graph $\overleftarrow{G}$ is also called *backward graph*. When both search scopes meet, a shortest path from s to t has been found.

3 Highway Hierarchy

A *highway hierarchy* of a graph G consists of several levels $G_0, G_1, G_2, \ldots, G_L$, where the number of levels $L+1$ is given. We provide an inductive definition:

- Base case (G'_0, G_0): level 0 ($G_0 = (V_0, E_0)$) corresponds to the original graph G; furthermore, we define $G'_0 := G_0$.
- First step ($G'_\ell \to G_{\ell+1}, 0 \le \ell < L$): for given *neighbourhood radii*, we will define the *highway network* $G_{\ell+1}$ of a graph G'_ℓ.
- Second step ($G_\ell \to G'_\ell, 1 \le \ell \le L$): for a given set $B_\ell \subseteq V_\ell$ of *bypassable* nodes, we will define the *core* G'_ℓ of level ℓ.

*First step (*highway network*).* For each node u, we choose a nonnegative *neighbourhood radius* $r^{\rightarrow}_\ell(u)$ for the forward graph and a radius $r^{\leftarrow}_\ell(u) \ge 0$ for the backward graph. To avoid some case distinctions, for any direction $\leftrightharpoons \in \{\rightarrow, \leftarrow\}$, we set the neighbourhood radius $r^{\leftrightharpoons}_\ell(u)$ to infinity for $u \notin V'_\ell$ and for $\ell = L$.

The level-ℓ *neighbourhood* of a node $u \in V'_\ell$ is $\mathcal{N}^{\rightarrow}_\ell(u) := \{v \in V'_\ell \mid d_\ell(u,v) \le r^{\rightarrow}_\ell(u)\}$ with respect to the forward graph and, analogously, $\mathcal{N}^{\leftarrow}_\ell(u) := \{v \in V'_\ell \mid d^{\leftarrow}_\ell(u,v) \le r^{\leftarrow}_\ell(u)\}$ with respect to the backward graph, where $d_\ell(u,v)$ denotes the distance from u to v in the forward graph G_ℓ and $d^{\leftarrow}_\ell(u,v) := d_\ell(v,u)$ in the backward graph $\overleftarrow{G_\ell}$.

The *highway network* $G_{\ell+1} = (V_{\ell+1}, E_{\ell+1})$ of a graph G'_ℓ is the subgraph of G'_ℓ induced by the edge set $E_{\ell+1}$: an edge $(u,v) \in E'_\ell$ belongs to $E_{\ell+1}$ iff there are nodes $s, t \in V'_\ell$ such that the edge (u,v) appears in the canonical shortest path[1] $\langle s, \ldots, u, v, \ldots, t\rangle$ from s to t in G'_ℓ with the property that $v \notin \mathcal{N}^{\rightarrow}_\ell(s)$ and $u \notin \mathcal{N}^{\leftarrow}_\ell(t)$.

*Second step (*core*).* For a given set $B_\ell \subseteq V_\ell$ of *bypassable* nodes, we define the set S_ℓ of *shortcut edges* that bypass the nodes in B_ℓ: for each path $P = \langle u, b_1, b_2, \ldots, b_k, v\rangle$ with $u, v \in V_\ell \setminus B_\ell$ and $b_i \in B_\ell, 1 \le i \le k$, the set S_ℓ contains an edge (u,v) with $w(u,v) = w(P)$. The *core* $G'_\ell = (V'_\ell, E'_\ell)$ of level ℓ is defined in the following way: $V'_\ell := V_\ell \setminus B_\ell$ and $E'_\ell := (E_\ell \cap (V'_\ell \times V'_\ell)) \cup S_\ell$.

4 Construction

Neighbourhood Radii. We suggest the following strategy to set the neighbourhood radii. For this paragraph, we interpret the graph G'_ℓ as an undirected graph, i.e., a directed edge (u,v) is interpreted as an undirected edge $\{u,v\}$ even if the edge (v,u) does not exist in the directed graph. Let us fix any rule that decides which element DIJKSTRA's algorithm removes from the priority queue in the case that there is more than one queued element with the smallest key. Then, during a DIJKSTRA search from a given node u in the undirected graph, all nodes are settled in a fixed order. The *Dijkstra rank* $\mathrm{rk}_u(v)$ of a node v is the rank of v w.r.t. this order. u has DIJKSTRA rank $\mathrm{rk}_u(u) = 0$, the closest neighbour v_1 of u

[1] For each connected node pair (s,t), we select a unique *canonical shortest path* in such a way that each subpath of a canonical shortest path is canonical as well. For details, we refer to [1].

has DIJKSTRA rank $\text{rk}_u(v_1) = 1$, and so on. For a given parameter H_ℓ, for any node $u \in V'_\ell$, we set $r^{\rightarrow}_\ell(u) := r^{\leftarrow}_\ell(u) := d^{\leftrightarrow}_\ell(u, v)$, where v is the node whose DIJKSTRA rank $\text{rk}_u(v)$ is H_ℓ; $d^{\leftrightarrow}_\ell(u, v)$ denotes the distance between u and v in the undirected graph. Applying this strategy to the forward and backward graph one after the other in order to define individual forward and backward radii yields a similar good performance, but needs twice the memory.

Fast Construction of a Highway Network. The fast construction method introduced in [1] has been modified in order to deal with directed graphs and the new, more general neighbourhood definition. For details, we refer to the full paper.

Contraction of a Graph. In order to obtain the core of a highway network, we contract it, which yields several advantages. The search space during the queries gets smaller since bypassed nodes are not touched and the construction process gets faster since the next iteration only deals with the nodes that have not been bypassed. Furthermore, a more effective contraction allows us to use smaller neighbourhood sizes without compromising the shrinking of the highway networks. This improves both construction and query times. However, bypassing nodes involves the creation of shortcuts, i.e., edges that represent the bypasses. Due to these shortcuts, the average degree of the graph is increased and the memory consumption grows. In particular, more edges have to be relaxed during the queries. Therefore, we have to carefully select nodes so that the benefits of bypassing them outweigh the drawbacks.

We give an iterative algorithm that combines the selection of the bypassable nodes B_ℓ with the creation of the corresponding shortcuts. We manage a stack that contains all nodes that have to be considered, initially all nodes from V_ℓ. As long as the stack is not empty, we deal with the topmost node u. We check the *bypassability criterion* $\#\text{shortcuts} \leq c \cdot (\deg_{\text{in}}(u) + \deg_{\text{out}}(u))$, which compares the number of shortcuts that would be created when u was bypassed with the sum of the in- and outdegree of u. The magnitude of the contraction is determined by the parameter c. If the criterion is fulfilled, the node is bypassed, i.e., it is added to B_ℓ and the appropriate shortcuts are created. Note that the creation of the shortcuts alters the degree of the corresponding endpoints so that bypassing one node can influence the bypassability criterion of another node. Therefore, all adjacent nodes that have been removed from the stack earlier, have not been bypassed, yet, and are bypassable now are pushed on the stack once again. It happens that shortcuts that were created once are discarded later when one of its endpoints is bypassed. Note that we will get a contraction that is similar to our trees-and-lines method [1] if we set $c = 0.5$.

5 Query

Our *highway query algorithm* is a modification of the bidirectional version of DIJKSTRA's algorithm. For now, we assume that the search is *not* aborted when both search scopes meet. This matter is dealt with in Section 6. We only describe the modifications of the forward search since forward and backward search are

symmetric. In addition to the *distance* from the source, the key of each node includes the search *level* and the *gap* to the next applicable neighbourhood border. The search starts at the source node s in level 0. First, a local search in the neighbourhood of s is performed, i.e., the gap to the next border is set to the neighbourhood radius of s in level 0. When a node v is settled, it adopts the gap of its parent u minus the length of the edge (u, v). As long as we stay inside the current neighbourhood, everything works as usual. However, if an edge (u, v) crosses the neighbourhood border (i.e., the length of the edge is greater than the gap), we switch to a higher search level ℓ. The node u becomes an *entrance point* to the higher level. If the level of the edge (u, v) is less than the new search level ℓ, the edge is *not* relaxed—this is one of the two restrictions that cause the speedup in comparison to DIJKSTRA's algorithm (Restriction 1). Otherwise, the edge is relaxed[2]. If the relaxation is successful, v adopts the new search level ℓ and the gap to the border of the neighbourhood of u in level ℓ since u is the corresponding entrance point to level ℓ.

We have to deal with the special case that an entrance point to level ℓ does not belong to the core of level ℓ. In this case, as soon as the level-ℓ core is entered, i.e., a node $u \in V'_\ell$ is settled, u is assigned the gap to the border of the level-ℓ neighbourhood of u. Note that before the core is entered, the gap has been infinity. To increase the speedup, we introduce another restriction (Restriction 2): when a node $u \in V'_\ell$ is settled, all edges (u, v) that lead to a bypassed node $v \in B_\ell$ in search level ℓ are *not* relaxed.

Despite of Restriction 1, we always find the optimal path since the construction of the highway hierarchy guarantees that the levels of the edges that belong to the optimal path are sufficiently high so that these edges are not skipped. Restriction 2 does not invalidate the correctness of the algorithm since we have introduced shortcuts that bypass the nodes that do not belong to the core. Hence, we can use these shortcuts instead of the original paths.

The Algorithm. We use two priority queues $\overrightarrow{Q}$ and $\overleftarrow{Q}$, one for the forward search and one for the backward search. The key of a node u is a triple $(\delta(u), \ell(u), \text{gap}(u))$, the (tentative) distance $\delta(u)$ from s (or t) to u, the search level $\ell(u)$, and the gap $\text{gap}(u)$ to the next applicable neighbourhood border. A key $(\delta, \ell, \text{gap})$ is less than another key $(\delta', \ell', \text{gap}')$ iff $\delta < \delta'$ or $\delta = \delta' \wedge \ell > \ell'$ or $\delta = \delta' \wedge \ell = \ell' \wedge \text{gap} < \text{gap}'$. Figure 1 contains the pseudo-code of the highway query algorithm. The proof of correctness is included in the full paper.

6 Optimisations

Abort on Success. In the bidirectional version of DIJKSTRA's algorithm, we can abort the search as soon as both search scopes meet. Unfortunately, this would be incorrect for our highway query algorithm. We therefore use a more conservative criterion: after a tentative shortest path P' has been encountered (i.e., after both search scopes have met), the forward (backward) search is not continued if the

[2] To *relax* an edge means to execute Line 11 in Fig. 1.

```
input: source node s and target node t

1  Q→.insert(s, (0, 0, r0→(s))); Q←.insert(t, (0, 0, r0←(t)));
2  while (Q→ ∪ Q← ≠ ∅) do {
3      ⇋ ∈ {→, ←};                                          // select direction
4      u := Q⇋.deleteMin();
5      if gap(u) ≠ ∞ then gap' := gap(u) else gap' := r⇋ℓ(u)(u);
6      foreach e = (u, v) ∈ E⇋ do {
7          for (ℓ := ℓ(u), gap := gap'; w(e) > gap; ℓ++)
               gap := r⇋ℓ+1(u);                             // go "upwards"
8          if ℓ(e) < ℓ then continue;                       // Restriction 1
9          if u ∈ V'ℓ ∧ v ∈ Bℓ then continue;               // Restriction 2
10         k := (δ(u) + w(e), ℓ, gap − w(e));
11         if v ∈ Q⇋ then Q⇋.decreaseKey(v, k); else Q⇋.insert(v, k);
12     }
13 }
```

Fig. 1. The highway query algorithm. Differences to the bidirectional version of DIJKSTRA's algorithm are marked: additional and modified lines have a framed line number; in modified lines, the modifications are underlined.

minimum element u of the forward (backward) queue has a key $\delta(u) \geq w(P')$. More sophisticated rules used in [1] turned out to be too expensive in terms of query time.

Speeding Up the Search in the Topmost Level. Let us assume that we have a distance table that contains for any node pair $s, t \in V'_L$ the optimal distance $d_L(s,t)$. Such a table can be precomputed during the preprocessing phase by $|V'_L|$ SSSP searches in V'_L. Using the distance table, we do not have to search in level L. Instead, when we arrive at a node $u \in V'_L$ that 'leads' to level L, we add u to a set $\overrightarrow{I}$ or $\overleftarrow{I}$ depending on the search direction; we do not relax the edge that leads to level L. After the sets $\overrightarrow{I}$ and $\overleftarrow{I}$ have been determined, we consider all pairs $(u,v), u \in \overrightarrow{I}, v \in \overleftarrow{I}$, and compute the minimum path length $D := d_0(s,u) + d_L(u,v) + d_0(v,t)$. Then, the length of the shortest s-t-path is the minimum of D and the length of the tentative shortest path found so far (in case that the search scopes have already met in a level $< L$). This optimisation can be included in the highway query algorithm (Fig. 1) by adding two lines:

between Lines 5 and 6:
5a **if** $\text{gap}' \neq \infty \wedge \ell(u) = L$ **then** $\{\overleftrightarrow{I} := \overleftrightarrow{I} \cup \{u\}$; **continue**;$\}$

between Lines 9 and 10:
9a **if** $\text{gap} \neq \infty \wedge \ell = L \wedge \ell > \ell(u)$ **then** $\{\overleftrightarrow{I} := \overleftrightarrow{I} \cup \{u\}$; **continue**;$\}$

7 Experiments

Environment and Instances. The experiments were done on one core of an AMD Opteron Processor 270 clocked at 2.0 GHz with 4 GB main memory and 2 × 1 MB L2 cache, running SuSE Linux 10.0 (kernel 2.6.13). The program was compiled by the GNU C++ compiler 4.0.2 using optimisation level 3. We use 32 bits to store edge weights and path lengths.

We deal with the road networks of Western Europe[3] and of the USA (without Hawaii) and Canada. Both networks have been made available for scientific use by the company PTV AG. 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. In order to compare ourselves with [1, 6], we also perform experiments on another version of the US road network (without Alaska and Hawaii) that was obtained from the TIGER/Line Files [7]. However, in contrast to the PTV data, the TIGER graph is undirected, planarised and distinguishes only between four road categories.

As in [1, 6], we report only the times needed to compute the shortest path distance between two nodes without outputting the actual route. In order to obtain the corresponding subpaths in the original graph, we are able to extract the used shortcuts without using any extra data. However, if a fast output routine is required, we might want to spend some additional space to accelerate the unpacking process. For details, we refer to the full paper. Table 1 summarises the properties of the used road networks and key results of the experiments.

Parameters. Unless otherwise stated, the following default settings apply. We use the contraction rate $c = 1.5$ and the neighbourhood sizes H as stated in Tab. 1—the same neighbourhood size is used for all levels of a hierarchy. First, we contract the original graph. Then, we perform four iterations of our construction procedure, which determines a highway network and its core. Finally, we compute the distance table between all level-4 core nodes.

In one test series (Fig. 2), we used all the default settings except for the neighbourhood size H, which we varied from 40 to 90. On the one hand, if H is too small, the shrinking of the highway networks is less effective so that the level-4 core is still quite big. Hence, we need much time and space to precompute and store the distance table. On the other hand, if H gets bigger, the time needed to preprocess the lower levels increases because the area covered by the local searches depends on the neighbourhood size. Furthermore, during a query, it takes longer to leave the lower levels in order to get to the topmost level where the distance table can be used. Thus, the query time increases as well. We observe that we get good space-time tradeoffs for neighbourhood sizes around 60. In particular, we find that a good choice of the parameter H does not vary very much from graph to graph.

In another test series (Tab. 2a), we did not use a distance table, but repeated the construction process until the topmost level was empty or the hierarchy

[3] 14 countries: at, be, ch, de, dk, es, fr, it, lu, nl, no, pt, se, uk.

Table 1. Overview of the used road networks and key results. '∅overhead/node' accounts for the additional memory that is needed by our highway hierarchy approach (divided by the number of nodes). The amount of memory needed to store the original graph is not included. Query times are average values based on 10 000 random *s*-*t*-queries. 'Speedup' refers to a comparison with DIJKSTRA's algorithm (unidirectional). Worst case is an upper bound for *any* possible query in the respective graph.

		Europe	USA/CAN	USA (Tiger)
INPUT	#nodes	18 029 721	18 741 705	24 278 285
	#directed edges	42 199 587	47 244 849	58 213 192
	#road categories	13	13	4
PARAM.	average speeds [km/h]	10–130	16–112	40–100
	H	50	60	60
PREPROC.	CPU time [min]	15	20	18
	∅overhead/node [byte]	68	69	50
QUERY	CPU time [ms]	0.76	0.90	0.88
	#settled nodes	884	951	1 076
	#relaxed edges	3 182	3 630	4 638
	speedup (CPU time)	8 320	7 232	7 642
	speedup (#settled nodes)	10 196	9 840	11 080
	worst case (#settled nodes)	8 543	3 561	5 141

consisted of 15 levels. We varied the contraction rate c from 0.5 to 2. In case of $c = 0.5$ (and $H = 50$), the shrinking of the highway networks does not work properly so that the topmost level is still very big. This yields huge query times. Note that in [1] we used a larger neighbourhood size to cope with this problem. Choosing larger contraction rates reduces the preprocessing and query times since the cores and search spaces get smaller. However, the memory usage and the average degree are increased since more shortcuts are introduced. Adding too many shortcuts ($c = 2$) further reduces the search space, but the number of relaxed edges increases so that the query times get worse.

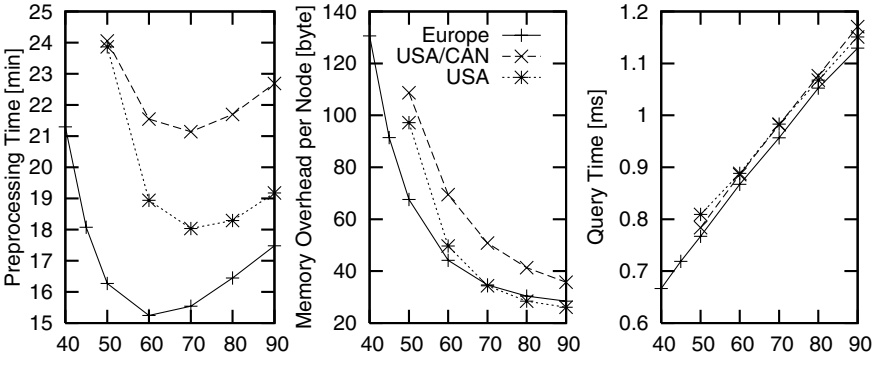


Fig. 2. Preprocessing and query performance depending on the neighbourhood size H

Table 2. Preprocessing and query performance for the European road network depending on the contraction rate c (a) and the number of levels (b). 'overhead' denotes the average memory overhead per node in bytes.

contr. rate c	PREPROCESSING time [min]	over-head	∅deg.	QUERY time [ms]	#settled nodes	#relaxed edges
0.5	89	27	3.2	176.05	242 156	505 086
1	16	27	3.7	1.97	2 321	8 931
1.5	13	27	3.8	1.58	1 704	7 935
2	13	28	3.9	1.70	1 681	8 607

(a)

# levels	PREPROC. time [min]	over-head	QUERY time [ms]	#settled nodes
5	16	68	0.77	884
7	13	28	1.19	1 290
9	13	27	1.51	1 574
11	13	27	1.62	1 694

(b)

In a third test series (Tab. 2b), we used the default settings except for the number of levels, which we varied from 5 to 11. In each test case, a distance table was used in the topmost level. The construction of the higher levels of the hierarchy is very fast and has no significant effect on the preprocessing times. In contrast, using only five levels yields a rather large distance table, which somewhat slows down the preprocessing and increases the memory usage. However, in terms of query times, '5 levels' is the optimal choice since using the distance table is faster than continuing the search in higher levels.

Space Saving. If we omit the first contraction step and use a smaller contraction rate ($\rightsquigarrow$ less shortcuts), use a bigger neighbourhood size ($\rightsquigarrow$ higher levels get smaller), and construct more levels before the distance table is used ($\rightsquigarrow$ smaller distance table), the memory usage can be reduced considerably. In case of Europe, using seven levels, $H = 100$, and $c = 1$ yields an average overhead per node of 17 bytes. The construction and query times increase to 55 min and 1.38 ms, respectively.

Worst Case Upper Bounds. By executing a query from each node of a given graph to an added isolated dummy node and a query from the dummy node to each actual node in the backward graph, we obtain a distribution of the search spaces of the forward and backward search, respectively. We can combine both distributions to get an upper bound for the distribution of the search spaces of bidirectional queries: when $\mathcal{F}_{\rightarrow}(x)$ ($\mathcal{F}_{\leftarrow}(x)$) denotes the number of source (target) nodes whose search space consists of x nodes in a forward (backward) search, we define $\mathcal{F}_{\leftrightarrow}(z) := \sum_{x+y=z} \mathcal{F}_{\rightarrow}(x) \cdot \mathcal{F}_{\leftarrow}(y)$, i.e., $\mathcal{F}_{\leftrightarrow}(z)$ is the number of s-t-pairs such that the upper bound of the search space size of a query from s to t is z. In particular, we obtain the upper bound $\max\{z \mid \mathcal{F}_{\leftrightarrow}(z) > 0\}$ for the worst case without performing all n^2 possible queries. Figure 3 visualises the distribution $\mathcal{F}_{\leftrightarrow}(z)$ as a histogram.

For the European road network, we observe several outliers between 7 800 and 8 600. The investigation of some samples indicates that these outliers are situated on some islands next to the Norwegian coast. Since Norway is sparsely populated and the road network is very sparse as well, we know that the neighbourhoods in

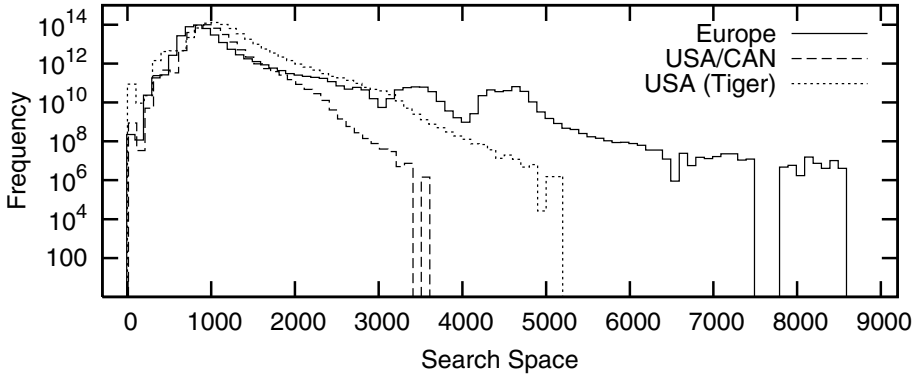


Fig. 3. Histogram of upper bounds for the search spaces of s-t-queries. To increase readability, only the outline of the histogram is plotted instead of the complete boxes.

low levels, which are defined by a fixed number of road junctions, cover a large geographic area. Hence, the search spreads very far before entering a reasonably high search level. When several densely populated areas are encountered while the search level is still quite low, the total search space size gets comparatively large. To improve the worst case, it might be a good idea to introduce adaptive neighbourhood sizes instead of fixed ones so that the above mentioned effect can be avoided.

In a similar way, we obtained a distribution of the number of entries in the distance table that have to be accessed during an s-t-query. While the average values are reasonably small (2 874 in case of Europe), the worst case can get quite large (62 250). For example, accessing 62 250 entries in a table of size 13 465 $\times$ 13 465 takes about 1 ms, where 13 465 is the size of the level-4 core of the European highway hierarchy. Hence, in some cases the time needed to determine the optimal entry in the distance table might dominate the query time. We could try to improve the worst case by introducing a case distinction that checks whether the number of entries that have to be considered exceeds a certain threshold. If so, we would not use the distance table, but continue with the normal search process. However, this measure would have only little effect on the average performance.

Comparisons. In Tab. 3, we compare several variants of our HH algorithm with the REAL algorithm, which is the method from [6] that features the best query times. Experimental results for the USA (Tiger) graph are taken from [6]. Results for the European graph have been provided by Andrew Goldberg.[4]

[4] These latter results have to viewed as tentative since the networks of Europe and North America are different (long distance ferries,...) and this was the very first attempt to run REAL on the European network. It is likely that future versions of REAL will yield better results.

Table 3. Comparison between the REAL algorithm [6] and our highway hierarchies. In addition to the current version of the highway hierarchies with the default settings (HH), we provide results that have been obtained using settings that reduce the memory usage (HH mem). Furthermore, we give old values (HH old) from [1]. Note that the *disk space* includes the memory that is needed to store the original graph.

	Europe				USA (Tiger)			
	PREPROCESSING		QUERY		PREPROCESSING		QUERY	
method	time [min]	disk space [MB]	time [ms]	#settled nodes	time [min]	disk space [MB]	time [ms]	#settled nodes
HH old	161	892	7.4	4 065	255	1 171	7.04	3 912
REAL	1 625	1 746	2.8	1 867	459	2 392	1.84	891
HH	15	1 570	0.8	884	18	1 686	0.88	1 076
HH mem	55	692	1.4	1 976	65	919	1.60	2 217

Note that the CPU times cannot be compared directly since the implementation of the REAL algorithm was executed on an AMD Opteron running at 2.4 GHz, while our machine only runs at 2.0 GHz. We also have to be careful when we compare the memory usage. In [6] a translation table is created that can be used to unpack shortcuts. We have subtracted the size of the translation tables from the disk spaces used by the REAL algorithm in order to account for the fact that our numbers do not include space for such a table.

Compared to the old highway hierarchies we see big improvements. An order of magnitude reduction in both preprocessing and query time. The main difference between HH and REAL is the dramatically smaller preprocessing time of HH. We see a factor 26 for the USA and a factor 108 for Europe. HH queries are also significantly faster than REAL. Only for the Tiger graph, REAL has a smaller search space. All variants of HH need less space than REAL. The main reason is the overhead for storing distances to landmarks.

8 Discussion

Highway hierarchies are a simple, robust and space efficient concept that allows very efficient fastest path queries even in huge realistic road networks. No other technique has reported such short query times although highway hierarchies have not yet been combined with goal directed search and although none of the previous techniques is competitive w.r.t. preprocessing time. Real-world applications suggest a number of additional challenging problems: How to handle mobile devices with limited fast memory? How to update or patch the hierarchy when edge weights change, e.g. due to traffic jams? What about multiple objective functions or time dependent edge weights? We are optimistic that several of these problems can be solved, in particular because plain highway hierarchies are so fast that even a 100 fold slow-down w.r.t. query time or preprocessing time would be acceptable in some situations.

References

1. Sanders, P., Schultes, D.: Highway hierarchies hasten exact shortest path queries. In: 13th European Symposium on Algorithms (ESA). Volume 3669 of LNCS., Springer (2005) 568–579
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, Universität Karlsruhe (TH), Fakultät für Informatik (2005)
4. Schultes, D.: Fast and exact shortest path queries using highway hierarchies. Master's thesis, Universität des Saarlandes (2005)
5. Gutman, R.: Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In: 6th Workshop on Algorithm Engineering and Experiments. (2004)
6. Goldberg, A., Kaplan, H., Werneck, R.: Reach for A^*: Efficient point-to-point shortest path algorithms. In: Workshop on Algorithm Engineering & Experiments. (2006)
7. U.S. Census Bureau, Washington, DC: UA Census 2000 TIGER/Line Files. `http://www.census.gov/geo/www/tiger/tigerua/ua_tgr2k.html` (2002)

Univariate Polynomial Real Root Isolation: Continued Fractions Revisited

Elias P. Tsigaridas and Ioannis Z. Emiris

Department of Informatics and Telecommunications,
National Kapodistrian University of Athens, HELLAS
{et, emiris}@di.uoa.gr

Abstract. We present algorithmic, complexity and implementation results concerning real root isolation of integer univariate polynomials using the continued fraction expansion of real numbers. We improve the previously known bound by a factor of $d\tau$, where d is the polynomial degree and τ bounds the coefficient bitsize, thus matching the current record complexity for real root isolation by exact methods. Namely, the complexity bound is $\widetilde{\mathcal{O}}_B(d^4\tau^2)$ using a standard bound on the expected bitsize of the integers in the continued fraction expansion. We show how to compute the multiplicities within the same complexity and extend the algorithm to non square-free polynomials. Finally, we present an efficient open-source C++ implementation in the algebraic library SYNAPS, and illustrate its efficiency as compared to other available software. We use polynomials with coefficient bitsize up to 8000 and degree up to 1000.

1 Introduction

In this paper we deal with real root isolation of univariate integer polynomials, a fundamental problem in computer algebra as well as in many applications ranging from computational geometry to quantifier elimination. The problem consists in computing intervals with rational endpoints which contain exactly one real root of the polynomial. We use the continued fraction expansion of *real algebraic numbers*. Recall that such a number is a real root of an integer polynomial.

One motivation is to explain the method's good performance in implementations, despite the higher complexity bound which was known until now. Indeed, we show that continued fractions lead to (expected) asymptotic bit complexity bounds that match those recently proven for other exact methods, such as Sturm sequences and Descartes' subdivision.

Notation: In what follows $\mathcal{O}_B$ means bit complexity and the $\widetilde{\mathcal{O}}_B$-notation means that we are ignoring logarithmic factors. For $A = \sum_{i=0}^{d} a_i X^i \in \mathbb{Z}[X]$, $\mathtt{deg}(A)$ denotes its degree. We consider square-free polynomials except if explicitly stated otherwise. By $\mathcal{L}(A)$ we denote an upper bound on the bit size of the coefficients of A (including a bit for the sign). For $\mathsf{a} \in \mathbb{Q}$, $\mathcal{L}(\mathsf{a}) \geq 1$ is the maximum bit size of the numerator and the denominator. Let $\mathsf{M}(\tau)$ denote the bit complexity of multiplying two integers of bit size at most τ. Using FFT, $\mathsf{M}(\tau) = \mathcal{O}_B(\tau \lg^c \tau)$ for a suitable constant c. $Var(A)$ denotes the sign variations

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 817–828, 2006.

in the coefficient list of A ignoring zero terms and Δ the separation bound of A, that is the smallest distance between two (complex) roots of A.

Previous work and our results: Real root isolation of univariate integer polynomials is a well known problem with a huge bibliography and we only scratch the surface of it. We encourage the reader to refer to the references.

Exact subdivision based algorithms for real root isolation are based either on Descartes' rule of sign or on Sturm sequences. Roughly speaking, the idea behind both approaches is to subdivide a given interval that initially contains all the real roots until it is certified that none or one root is contained. Quite recently it was proven (cf [12, 13] and references therein) that both approaches (Descartes and Sturm), achieve the same bit complexity bound, namely $\widetilde{\mathcal{O}}_B(d^4\tau^2)$, where d is the polynomial degree and τ bounds the coefficient bitsize, or $\widetilde{\mathcal{O}}_B(N^6)$, where $N = \max\{d, \tau\}$. Moreover using Sturm sequences in a pre-processing and a post-processing step [14, 15] the bound holds for the non square-free case and the multiplicities of the roots can also be computed.

The continued fraction algorithm (from now on called CF) differs from the subdivision algorithms in that instead of bisecting a given initial interval it computes the continued fraction expansions of the real roots of the polynomial. The first formulation of CF is due to Vincent [32], see also [2] for historical references, based on his theorem (Th. 3 without the terminating condition) where it was stated that repeated transformations of the polynomial will eventually yield a polynomial with zero (or one) sign variation, thus Descartes' rule implies the transformed polynomial has zero (resp. one) real root in $(0, \infty)$. Unfortunately Vincent's algorithm is exponential [9].

Uspensky [29] extended Vincent's theorem by computing an upper bound on the number of transformations so as to isolate the real roots, but failed to deal with its exponential behavior. Using Vincent's theorem, Collins and Akritas [9] derived a polynomial subdivision-based algorithm using Descartes' rule of sign. Akritas [5, 1] dealt with the exponential behavior of CF, by computing the partial quotients as positive lower bounds of the positive real roots, via Cauchy's bound (for details, see Sec. 4), and obtained a complexity of $\widetilde{\mathcal{O}}_B(d^5\tau^3)$ or $\widetilde{\mathcal{O}}_B(N^8)$, without using fast Taylor shifts [33]. However, it is not clear how this approach accounts for the increased coefficient size in the transformed polynomial after applying $X \mapsto b + X$. Another issue is to bound the size of the partial quotients. Refer to Eq. (1) which indicates that the *magnitude* of the partial quotients is unbounded. CF is the standard real root isolation algorithm in MATHEMATICA [3]. For some experiments against subdivision-based algorithms, in MATHEMATICA, the reader may refer to [4].

Another class of univariate solvers are numerical solvers, e.g. [24, 6] that compute an approximation of all the roots of a polynomial up to a desired accuracy. The complexity of these algorithms is $\widetilde{\mathcal{O}}_B(d^3\,\tau)$ or $\widetilde{\mathcal{O}}_B(N^4)$.

The contributions of this paper are the following: First, we improve the bound of the number of steps (transformations) that the algorithm performs. Second, we bound the bitsize of the partial quotients and thus the growth of the transformed polynomials which appear during the algorithm. We revisit the proof of [5, 1] so

as to improve the overall bit complexity bound of the algorithm to $\widetilde{\mathcal{O}}_B(N^6)$, thus matching the current record complexity for real root isolation. The extension to the non square-free case uses the techniques from [14, 15]. Third, we present our efficient open-source C++ implementation in SYNAPS[1] [23], and illustrate it on various data sets, including polynomials of degree up to 1000 and coefficients of 8000 bits. We performed experiments against RS[2], which seems to be one of the fastest available software for exact real root isolation and against ABERTH [6], a numerical solver available through SYNAPS. Our implementation seems to have the best performance in practice. We believe that our software contributes towards reducing the gap between rational and numeric computation, the latter being usually perceived as faster.

The rest of the paper is structured as follows. The next section sketches the theory behind continued fractions. Sec. 3 presents the CF algorithm and Sec. 4 its analysis. We conclude with experiments using our implementation, along with comparisons against other available software for univariate equation solving.

2 Continued Fractions

We present a short introduction to continued fractions, following [30] which although is far from complete suffices for our purposes. The reader may refer to e.g [5, 34, 7, 30]. In general a *simple (regular) continued fraction* is a (possibly infinite) expression of the form

$$c_0 + \cfrac{1}{c_1 + \cfrac{1}{c_2 + \dots}} = [c_0, c_1, c_2, \dots]$$

where the numbers c_i are called *partial quotients*, $c_i \in \mathbb{Z}$ and $c_i \geq 1$ for $i > 0$. Notice that c_0 may have any sign. By considering the recurrent relations

$$P_{-1} = 1,\ P_0 = c_0,\ P_{n+1} = c_{n+1}\,P_n + P_{n-1}$$
$$Q_{-1} = 0,\ Q_0 = 1,\ Q_{n+1} = c_{n+1}\,Q_n + Q_{n-1}$$

it can be shown by induction that $R_n = \frac{P_n}{Q_n} = [c_0, c_1, \dots, c_n]$, for $n = 0, 1, 2, \dots$

If $\gamma = [c_0, c_1, \dots]$ then $\gamma = c_0 + \frac{1}{Q_0 Q_1} - \frac{1}{Q_1 Q_2} + \dots = c_0 + \sum_{n=1}^{\infty} \frac{(-1)^{n-1}}{Q_{n-1} Q_n}$ and since this is a series of decreasing alternating terms it converges to some real number γ. A finite section $R_n = \frac{P_n}{Q_n} = [c_0, c_1, \dots, c_n]$ is called the $n-$th *convergent* (or *approximant*) of γ and the tails $\gamma_{n+1} = [c_{n+1}, c_{n+2}, \dots]$ are known as its *complete quotients*. That is $\gamma = [c_0, c_1, \dots, c_n, \gamma_{n+1}]$ for $n = 0, 1, 2, \dots$. There is a one to one correspondence between the real numbers and the continued fractions, where the finite continued fractions correspond to rational numbers. It is known that $Q_n \geq F_{n+1}$ and that $F_{n+1} < \phi^n < F_{n+2}$, where F_n is the $n-$th Fibonacci number and $\phi = \frac{1+\sqrt{5}}{2}$ is the *golden ratio*. Continued fractions are the best (for a given denominator size), approximations, i.e

[1] www-sop.inria.fr/galaad/logiciels/synaps/
[2] fgbrs.lip6.fr/salsa/Software/index.php

$$\frac{1}{Q_n(Q_{n+1}+Q_n)} \leq \left|\gamma - \frac{P_n}{Q_n}\right| \leq \frac{1}{Q_nQ_{n+1}} \leq \frac{1}{Q_n^2} < \phi^{-2n}$$

Let $\gamma = [c_0, c_1, \dots]$ be the continued fraction expansion of a real number. The Gauss-Kuzmin distribution [7, 25] states that for almost all real numbers γ (the set of exceptions has Lebesgue measure zero) the probability for a positive integer δ to appear as an element in the continued fraction expansion of γ is

$$Prob[c_i = \delta] = \lg \frac{(\delta+1)^2}{\delta(\delta+2)}, \quad i > 0 \tag{1}$$

The Gauss-Kuzmin law induces that we can not bound the mean value of the partial quotients, i.e $E[c_i] = \sum_{\delta=1}^{\infty} \delta\, Prob[c_i = \delta] = \infty$, $i > 0$. However the geometric (and the harmonic) mean is not only asymptotically bounded, but is bounded by a constant. For the geometric mean this is the famous Khintchine's constant [17], i.e. $\lim_{n\to\infty} \sqrt[n]{\prod_{i=1}^{n} c_i} = \mathcal{K} = 2.685452001...$ which is not known if it is an irrational number, let alone transcendental. The expected value of the bitsize of the partial quotients is a constant for almost all real numbers, when $n \to \infty$ or n sufficiently big [17, 25]. Following closely [25], we have: $E[\ln c_i] = \frac{1}{n}\sum_{i=1}^{n} \ln c_i = \ln \mathcal{K} = 0.98785...$, as $n \to \infty$, $\forall i > 0$. Let $\mathcal{L}(c_i) \triangleq b_i$, then

$$E[b_i] = \mathcal{O}(1) \tag{2}$$

A real number has an (eventually) periodic continued fraction expansion if and only if it is a root of an irreducible quadratic polynomial. "There is no reason to believe that the continued fraction expansions of non-quadratic algebraic irrationals generally do anything other than faithfully follow Khintchine's law"[8], and also various experimental results [7, 25, 26] suggest so.

3 The CF Algorithm

Theorem 1 (Budan). *[20, 5] Let $A \in \mathbb{R}[X]$ and $a < b$, where $a, b \in \mathbb{R}$. Let A_a (A_b) be the polynomial produced after we apply the map $X \mapsto X+a$ ($X \mapsto X+b$) to A. Then the followings hold: (i) $Var(A_a) \geq Var(A_b)$, (ii) $\#\{\gamma \in (a,b) | A(\gamma) = 0\} \leq Var(A_a) - Var(A_b)$ and (iii) $\#\{\gamma \in (a,b) | A(\gamma) = 0\} \equiv Var(A_a) - Var(A_b)$* mod 2.

Theorem 2 (Descartes' rule of sign). *The number R of real roots of $A(X)$ in $(0, \infty)$ is bounded by $Var(A)$ and we have $R \equiv Var(A)$* mod 2.

In general Descartes' rule of sign obtains an overestimation of the number of the positive real roots. However if we know that A is *hyperbolic*, i.e has only real roots or when the number of sign variations is 0 or 1 then it counts exactly.

The CF algorithm depends on the following theorem, which dates back to Vincent's theorem in 1836 [32]. It is a very interesting question whether the one and two circle theorems (cf [19] and references therein), employed in the analysis of the subdivision-based real-root isolation algorithm [9], can also be applied and possibly improve the complexity of CF.

Theorem 3. *[5, 29] Let $A \in \mathbb{Z}[X]$, with $\mathtt{deg}(A) = d$ and let Δ be the separation bound. Let n be the smallest index such that $F_{n-1}\Delta > 2$ and $F_{n-1}F_n\Delta > 1 + \frac{1}{\epsilon_d}$, where F_n is the n-th Fibonnaci number and $\epsilon_d = (1 + \frac{1}{d})^{\frac{1}{d-1}} - 1$. Then the map $X \mapsto [c_0, c_1, \ldots, c_n, X]$, where $c_0, c_1, \ldots, c_n$ is an arbitrary sequence of positive integers, transforms $A(X)$ to $A_n(X)$, which has no more than one sign variation.*

Remark 1. Since $\frac{3}{4d^2} < \epsilon_d < \frac{4}{d^2}$ [10] we conclude that $\frac{1}{\epsilon_d} + 1 < 2d^2$ for $d \geq 2$. Thus, if $d \geq 2$ we can replace the two conditions of Th. 3 by $F_{n-1}\Delta \geq 2d^2$, since $F_n \geq F_{n-1} \geq 1$ and $F_{n-1}F_n\Delta \geq F_{n-1}\Delta \geq 2d^2 > 2$.

Th. 3 can be used to isolate the positive real roots of a square-free polynomial A. In order to isolate the negative roots we perform the transformation $X \mapsto -X$, so in what follows we will consider only the positive real roots of A. Vincent's variant of CF goes as follows: A polynomial A is transformed to A_1 by the transformation $X \mapsto 1 + X$ and if $Var(A_1) = 0$ or $Var(A_1) = 1$ then A has 0, resp. 1, real root greater than 1 (Th. 2). If $Var(A_1) < Var(A)$ then (possibly) there are real roots of A in $(0, 1)$, due to Budan's theorem (Th. 1). A_2 is produced by applying the transformation $X \mapsto 1/(1+X)$ to A, if $Var(A_2) = 0$ or $Var(A_2) = 1$ then A has 0, resp. 1, real root less than 1 (Th. 2). Uspensky's [29] variant of the algorithm (see also [26]) at every step produces both polynomials A_1 and A_2, probably, as Akritas states [2], because he was unaware of Budan's theorem (Th. 1). In both variants, if the transformed polynomial has more than one sign variations, we repeat the process.

We may consider the process of CF as an infinite binary tree in which the root corresponds to the initial polynomial A. The branch from a node to a right (left) child corresponds to the map $X \mapsto X+1$ ($X \mapsto \frac{1}{1+X}$). Vincent's algorithm (and Uspensky's) results to a sequence of transformations as in Th. 3, and so the leaves of the tree hold (transformed) polynomials that have no more than one sign variations, if Th. 3 holds. Akritas [1, 5] replaced a series of $X \mapsto X + 1$ transformations by $X \mapsto X + b$, where b is the positive lower bound (`PLB`) on the positive roots of the tested polynomial, using Cauchy's bound [5, 34]. This way, the number of steps is polynomial and the complexity is in $\widetilde{\mathcal{O}}_B(d^5\tau^3)$. However, it is not clear whether or how the analysis takes into account that the coefficient bitsize increases after a shift operation. Another issue is to bound the size of b.

For these polynomials that have one sign variation we still have to find the interval where the real root of the initial polynomial A lies. Consider a polynomial A_n that corresponds to a leaf of the binary tree that has one sign variation. Notice that A_n is produced after a transformation as in Th. 3, using positive integers $c_0, c_1, \ldots, c_n$. Using the convergents, this transformation becomes

$$M : X \mapsto \frac{P_n X + P_{n-1}}{Q_n X + Q_{n-1}} \tag{3}$$

where $\frac{P_{n-1}}{Q_{n-1}}$ and $\frac{P_n}{Q_n}$ are consecutive convergents of the continued fraction $[c_0, c_1, \ldots, c_n]$. Notice that (3) is a Möbius transformation, see [5, 34] for more details. Since A_n has one sign variation it has one and only one real root in $(0, \infty)$, so in order to obtain the isolating interval for the corresponding real

Algorithm 1. $\texttt{CF}(A, M)$

Input: $A \in \mathbb{Z}[X], M(X) = \frac{kX+l}{mX+n}$, $k, l, m, n \in \mathbb{Z}$

```
1  if A(0) = 0 then
2  |  OUTPUT Interval( M(0), M(0)) ;
3  |  A ← A(X)/X;
4  |  CF(A, M);
5  if Var(A) = 0 then RETURN ;
6  if Var(A) = 1 then OUTPUT Interval( M(0), M(∞)), RETURN ;
7  b ← PLB(A) // PLB ≡ PositiveLowerBound ;
8  if b > 1 then A ← A(b + X), M ← M(b + X) ;
9  A1 ← A(1 + X), M1 ← M(1 + X) ;
10 CF(A1, M1) // Looking for real roots in (1, +∞);
11 A2 ← A(1/(1+X)), M2 ← M(1/(1+X)) ;
12 CF(A2, M2) // Looking for real roots in (0, 1) ;
```

root of A we evaluate the right part of Eq. (3) once over 0 and once over ∞. The (unordered) endpoints of the isolating interval are $\frac{P_{n-1}}{Q_{n-1}}$ and $\frac{P_n}{Q_n}$.

The pseudo-code of CF is presented in Alg. 1. The `Interval` function orders the endpoints of the computed isolating interval and $\texttt{PLB}(A)$ computes a lower bound on the positive roots of A. The input of the algorithm is a polynomial $A(X)$ and the trivial transformation $M(X) = X$. Notice that Lines 11 and 12 are to be executed only when $Var(A_1) < Var(A_2)$, but in order to simplify the analysis we omit this, since it only doubles the complexity.

4 The Complexity of the CF Algorithm

Let $\texttt{disc}(A)$ be the discriminant and $\texttt{lead}(A)$ the leading coefficient of A. Mahler's measure of a polynomial A is $\mathcal{M}(A) = |\,\texttt{lead}(A)\,| \prod_{i=1}^{d} \max\{1, |\gamma_i|\}$, where γ_i are all the (complex) roots of A [34, 20, 21]. We prove the following theorem, which is based on a theorem by Mignotte [20], thus extending [11, 13].

Theorem 4. *Let $A \in \mathbb{Z}[X]$, with $\deg(A) = d$ and $\mathcal{L}(A) = \tau$. Let Ω be any set of k pairs of indices (i, j) such that $1 \le i < j \le d$ and let the non-zero (complex) roots of A be $0 < |\gamma_1| \le |\gamma_2| \le \cdots \le |\gamma_d|$. Then*

$$2^k \mathcal{M}(A)^k \ge \prod_{(i,j)\in\Omega} |\gamma_i - \gamma_j| \ge 2^{k - \frac{d(d-1)}{2}}\, \mathcal{M}(A)^{1-d-k} \sqrt{\texttt{disc}(A)}$$

Proof. Consider the multiset $\overline{\Omega} = \{j | (i, j) \in \Omega\}$, $|\overline{\Omega}| = k$. We use the inequality

$$\forall\, a, b \in \mathbb{C} \quad |a - b| \le 2 \max\{|a|, |b|\} \tag{4}$$

and the fact [20, 21] that for any root of A, $\frac{1}{\mathcal{M}(A)} \le |\gamma_i| \le \mathcal{M}(A)$. In order to prove the left inequality

$$\prod_{(i,j)\in\Omega} |\gamma_i - \gamma_j| \le 2^k \prod_{j\in\overline{\Omega}} |\gamma_j| \le 2^k \max_{j\in\overline{\Omega}} |\gamma_j|^k \le 2^k \mathcal{M}(A)^k.$$

Recall [34, 20] that $\mathtt{disc}(A) = \mathtt{lead}(A)^{2d-2} \prod_{i<j} (\gamma_i - \gamma_j)^2$. For the right inequality we consider the absolute value of the discriminant of A:

$$\begin{array}{rl} |\,\mathtt{disc}(A)| & = |\,\mathtt{lead}(A)\,|^{2d-2} \prod_{i<j} |\gamma_i - \gamma_j|^2 \\ & = |\,\mathtt{lead}(A)\,|^{2d-2} \prod_{(i,j)\in\Omega} |\gamma_i - \gamma_j|^2 \prod_{(i,j)\notin\Omega} |\gamma_i - \gamma_j|^2 \Leftrightarrow \\ \sqrt{|\,\mathtt{disc}(A)|} & = |\,\mathtt{lead}(A)\,|^{d-1} \prod_{(i,j)\in\Omega} |\gamma_i - \gamma_j| \prod_{(i,j)\notin\Omega} |\gamma_i - \gamma_j| \end{array}$$

We consider the product $\prod_{(i,j)\notin\Omega} |\gamma_i - \gamma_j|$ and we apply $\frac{d(d-1)}{2} - k$ times inequality (4), thus

$$\begin{array}{l} \prod_{(i,j)\notin\Omega} |\gamma_i - \gamma_j| \le 2^{\frac{d(d-1)}{2}-k} |\gamma_1|^0 |\gamma_2|^1 \cdots |\gamma_d|^{d-1} \left(\prod_{j\in\overline{\Omega}} |\gamma_j|\right)^{-1} \\ \qquad\qquad \le 2^{\frac{d(d-1)}{2}-k} \mathcal{M}(A)^{d-1} |\,\mathtt{lead}(A)\,|^{1-d} \mathcal{M}(A)^k \end{array} \tag{5}$$

where we used the inequality $|\gamma_1|^0 |\gamma_2|^1 \cdots |\gamma_d|^{d-1} \le |\mathcal{M}(A)/\,\mathtt{lead}(A)\,|^{d-1}$, and the fact [20] that, since $\forall i$, $|\gamma_i| \ge \mathcal{M}(A)^{-1}$, we have $\prod_{j\in\overline{\Omega}} |\gamma_j| \ge |\gamma_1|^k \ge \mathcal{M}(A)^{-k}$. Thus $\prod_{(i,j)\in\Omega} |\gamma_i - \gamma_j| \ge 2^{k-\frac{d(d-1)}{2}} \mathcal{M}(A)^{1-d-k} \sqrt{|\,\mathtt{disc}(A)|}$. □

A similar theorem but with more strict hypotheses on the roots first appeared in [11] and the conditions were generalized in [13]. Th. 4 has a factor 2^{d^2} instead of d^d in [11, 13], which plays no role when $d = \mathcal{O}(\tau)$ or when notation with N is used. Possibly a more involved proof of Th. 4 may eliminate this factor [22].

Remark 2. There is a simple however crucial observation about Th. 3. When the transformed polynomial has one (zero) sign variation, then the interval with endpoints $\frac{P_{n-1}}{Q_{n-1}} = [c_0, \ldots, c_{n-1}]$ and $\frac{P_n}{Q_n} = [c_0, \ldots, c_n]$ isolates a positive real root (a complex root with positive real part) of A, say γ_i. Then, in order for Th. 3 to hold, it suffices to consider, instead of the separation bound Δ, the quantity $|\gamma_i - \gamma_{c_i}|$, where γ_{c_i} is the (complex) root of A closest to γ_i.

Theorem 5. *The CF algorithm performs at most $\mathcal{O}(d^2 + d\tau)$ steps.*

Proof. Let $0 < |\gamma_1| \le \cdots \le |\gamma_k|$, $k \le d$ be the (complex) roots of A with positive real part and let γ_{c_i} denote the root of A that is closest to γ_i. We consider the binary tree T generated during the execution of CF. The number of steps of CF corresponds to the number of nodes in T, which we denote by $\#(T)$. We use some arguments and the notation from [13] in order to prune T.

With each node v of T we associate a Möbius transformation $M_v : X \mapsto \frac{kX+l}{mX+n}$, a polynomial A_v and implicitly an interval I_v whose unordered endpoints can be found if we evaluate M_v on 0 and on ∞. Recall that A_v is produced after M_v is applied to A. The root of T is associated with A, $M(X) = X$ (i.e $k = n = 1, l = m = 0$) and implicitly with the interval $(0, \infty)$.

Let a leaf u of T be **type-i** if its interval I_u contains $i \ge 0$ real roots. Since the algorithm terminates the leaves are type-0 or type-1. We will prune certain leaves of T so as to obtain a certain subtree T'. We remove every leaf that has a sibling that is not a leaf. Now we consider the leaves that have a sibling that is also a leaf. If both leaves are type-1, we arbitrary prune one of them. If one of them is type-1 then we prune the other. If both leaves are type-0, this means that the polynomial on the parent node has at least two sign variations and thus that we

are trying to isolate the (positive) real part of some complex root. We keep the leaf that contains the (positive) real part of this root. And so $\#(T) < 2\,\#(T')$.

Now we consider the leaves of T'. All are type-0 or type-1. In both cases they hold the positive real part of a root of A, the associated interval is $|I_v| \geq |\gamma_i - \gamma_{c_i}|$ (Rem. 2) and the number of nodes from a leaf to the root is n_i, which is such that the condition of Rem. 1 is satisfied. Since n_i is the smallest index such that the condition of Rem. 1 holds, if we reduce n_i by one then the inequality does not hold. Thus

$$F_{n_i-2}|\gamma_i - \gamma_{c_i}| \leq 2d^2 \Rightarrow \phi^{n_i-3}|\gamma_i - \gamma_{c_i}| < 2d^2 \Rightarrow n_i < 4 + 2\lg d - \lg|\gamma_i - \gamma_{c_i}|$$

We sum over all n_i to bound the nodes of T', thus

$$\#(T') \leq \sum_{i=1}^{k} n_i \leq 2k(2+\lg d) - \sum_{i=1}^{k} \log|\gamma_i - \gamma_{c_i}| \leq 2k(2+\lg d) - \log\prod_{i=1}^{k}|\gamma_i - \gamma_{c_i}| \quad (6)$$

So as to use Th. 4 we should rearrange $\prod_{i=1}^{k}|\gamma_i - \gamma_{c_i}|$ so that the requirements on the indices of roots are fulfilled. This can not be achieved when symmetric products occur and the worst case is when the product consists only of symmetric products i.e $\prod_{i=1}^{k/2}|(\gamma_j - \gamma_{c_j})(\gamma_{c_j} - \gamma_j)|$. Thus we consider the square of the inequality of Th. 4 taking $\frac{k}{2}$ instead of k and $\mathtt{disc}(A) \geq 1$ (since A is square-free), thus

$$\begin{aligned} \textstyle\prod_{i=1}^{k}|\gamma_i - \gamma_{c_i}| &\geq \left(2^{\frac{k}{2}-\frac{d(d-1)}{2}}\,\mathcal{M}(A)^{1-d-\frac{k}{2}}\right)^2 \\ -\log\textstyle\prod_{i=1}^{d}|\gamma_i - \gamma_{c_i}| &\leq d^2 - d - k + (2d+k-2)\lg\mathcal{M}(A) \end{aligned} \quad (7)$$

Eq. (6) becomes $\#(T') \leq 2k(2+\lg d) + d^2 - d - k + (2d+k-2)\lg\mathcal{M}(A)$. However for Mahler's measure it is known that $\mathcal{M}(A) \leq 2^\tau\sqrt{d+1} \Rightarrow \lg\mathcal{M}(A) \leq \tau + \lg d$, for $d \geq 2$, thus $\#(T') \leq 2k(2+\lg d) + d^2 - d - k + (2d+k-2)(\tau + \lg d)$. Since $\#(T) < 2\,\#(T')$ and $k \leq d$, we conclude that $\#(T) = \mathcal{O}(d^2 + d\,\tau + d\lg d)$. □

To complete the analysis of CF we have to compute the cost of every step that the algorithm performs. In the worst case every step consists of a computation of a positive lower bound b (Line 7) and three transformations, $X \mapsto b + X$, $X \mapsto 1 + X$ and $X \mapsto \frac{1}{1+X}$ (Lines 8, 9 and 11 in Alg. 1). Since inversion can be performed in $\mathcal{O}(d)$, the complexity is dominated by the cost of the shift operation (Line 8 in Alg. 1) if a small number of calls to `PLB` is needed in order to compute a partial quotient. We will justify this in the end of the section. We also will use the following theorem:

Theorem 6 (Fast Taylor shift). *[33] Let $A \in \mathbb{Z}[X]$, with $\deg(A) = d$ and $\mathcal{L}(A) = \tau$ and let $a \in \mathbb{Z}$, such that $\mathcal{L}(a) = \sigma$. The cost of computing $B = A(a+X) \in \mathbb{Z}[X]$ is $\mathcal{O}_B(\mathsf{M}\left(d^2\lg d + d^2\sigma + d\tau\right))$. Moreover $\mathcal{L}(B) = \mathcal{O}(\tau + d\sigma)$.*

Initially A has degree d and bitsize τ. Evidently the degree does not change after a shift operation. Each shift operation by a number of bitsize b_h increases the bit size of the polynomial by an additive factor $d\,b_h$, in the worst case (Th. 6). At the $h-$th step of the algorithm the polynomial has bit size $\mathcal{O}(\tau + d\sum_{i=1}^{h} b_i)$ and we perform a shift operation by a number of bit size b_{h+1}. Th. 6 states that this can be done in $\mathcal{O}_B\left(\mathsf{M}\left(d^2\lg d + d^2 b_{h+1} + d(\tau + d\sum_{i=1}^{h} b_i)\right)\right)$.

Table 1. Experimental results

		100	200	300	400	500	600	700	800	900	1000
L	CF	0.27	2.24	9.14	25.27	55.86	110.13	214.99	407.09	774.22	1376.34
	RS	0.65	3.65	13.06	35.23	77.21	151.17	283.43	527.42	885.86	1387.45
	#roots	100	200	300	400	500	600	700	800	900	1000
C1	CF	0.11	0.85	3.16	8.61	19.67	38.23	77.75	139.18	247.11	414.51
	RS	0.21	1.36	3.80	10.02	23.15	46.02	82.01	150.01	269.35	458.67
	#roots	100	200	300	400	500	600	700	800	900	1000
C2	CF	0.11	0.77	3.14	8.20	19.28	38.58	73.59	133.52	233.48	386.61
	RS	0.23	1.48	3.80	9.84	23.28	46.34	83.58	146.04	273.00	452.77
	#roots	100	200	300	400	500	600	700	800	900	1000
W	CF	0.11	0.76	2.54	6.09	12.07	21.43	34.52	53.35	81.88	120.21
	RS	0.09	0.59	2.25	6.34	14.62	29.82	55.47	104.56	179.23	298.45
	#roots	100	200	300	400	500	600	700	800	900	1000
M1	CF	0.02	0.08	0.21	0.42	0.73	1.19	1.84	2.75	4.16	6.22
	RS	7.83	287.27	1936.48	7328.86	*	*	*	*	*	*
	ABERTH	0.01	0.04	0.07	0.11	0.12	0.26	0.43	0.37	0.47	0.90
	#roots	4	4	4	4	4	4	4	4	4	4
M2	CF	0.08	0.43	1.10	2.78	4.71	8.67	18.26	25.28	40.15	60.10
	RS	1.24	144.64	1036.785	4278.275	12743.79	*	*	*	*	*
	ABERTH	0.04	0.78	3.24	?	?	?	?	?	?	?
	#roots	8	8	8	8	8	8	8	8	8	8
R1	CF	0.001	0.04	0.07	0.33	0.06	0.37	0.66	0.76	1.03	1.77
	RS	0.026	0.09	0.11	0.68	0.22	0.89	0.95	0.69	1.55	2.09
	ABERTH	0.02	0.03	0.07	0.14	0.21	0.31	0.44	0.51	0.64	0.80
	#roots	4	4	2	6	2	4	4	2	4	4
R2	CF	0.01	0.04	0.08	0.36	0.14	0.38	0.74	0.77	1.24	1.42
	RS	0.05	0.23	0.47	1.18	0.81	1.64	2.68	3.02	4.02	4.88
	ABERTH	0.01	0.05	0.08	0.14	0.23	0.33	0.44	0.55	0.67	0.83
	#roots	4	4	4	6	4	4	6	4	6	4

In order to bound $\sum_{i=1}^{h+1} b_i$ we use Eq. (2), which bounds $E[b_i]$. By linearity of expectation it follows that $E[\sum_{i=1}^{h+1} b_i] = \mathcal{O}(h)$. Since $h \leq \#(T) = \mathcal{O}(d^2 + d\tau)$ (Th. 5), the (expected) worst case cost of step h is $\mathcal{O}_B(\mathsf{M}\left(d^2 \lg d + d\tau + d^2(d^2 + d\tau)\right))$ or $\widetilde{\mathcal{O}}_B(d^2(d^2 + d\tau))$. Finally, multiplying by the number of steps, $\#(T)$, we conclude that the overall complexity is $\widetilde{\mathcal{O}}_B(d^6 + d^5\tau + d^4\tau^2)$, or $\widetilde{\mathcal{O}}_B(d^4\tau^2)$ if $d = \mathcal{O}(\tau)$.

Now consider $A_{in} \in \mathbb{Z}[X]$, not necessarily square-free, with $\mathtt{deg}(A_{in}) = d$ and $\mathcal{L}(A_{in}) = \tau$. Following [14, 15] we compute the square-free part A of A_{in} using Sturm-Habicht sequences in $\widetilde{\mathcal{O}}_B(d^2\tau)$ and $\mathcal{L}(A) = \mathcal{O}(d+\tau)$. Using CF we isolate the positive real root of A and then, by applying the map $X \mapsto -X$, we isolate the negative real roots. Finally, using the square-free factorization of A_{in}, which can be computed in $\widetilde{\mathcal{O}}_B(d^3\tau)$, it is possible to find the multiplicities in $\widetilde{\mathcal{O}}_B(d^3\tau)$. The previous discussion leads to the following theorem:

Theorem 7. *Let $A \in \mathbb{Z}[X]$ (not necessarily square-free) such that $\mathtt{deg}(A) = d > 2$ and $\mathcal{L}(A) = \tau$. We can isolate the real roots of A and compute their multiplicities in expected time $\widetilde{\mathcal{O}}_B(d^6+d^4\tau^2)$, or $\widetilde{\mathcal{O}}_B(N^6)$, where $N = \max\{d, \tau\}$.*

Rational roots and PLB (Positive Lower Bound) realization: If $\frac{p}{q}$ is a root of A then p divides a_0 and q divides a_d, thus in the worst case $\mathcal{L}(p/q) = \mathcal{O}(\tau)$ and so the rational roots are isolated fast. Treating them as real algebraic numbers

leads to an overestimation of the number of iterations. There is one exception to this good behavior of rational roots, namely when they are very large, well separated, and we are interested in practical complexity [3], since then PLB must be applied many times. In [25], the authors performed a small number of Newton iterations in order to have a good approximation of a partial quotient. In [3, 4], this problem was solved by performing the transformation $X \mapsto bX$, where b is the computed bound, whenever $b \geq 16$. We follow the latter approach so, after Line 8 in Alg. 1, if $b = \texttt{PLB}(A) \geq 16$, we apply $X \mapsto bX$ to polynomial A.

$\texttt{PLB}(A)$ is computed as the inverse of an upper bound on the roots of $X^d A(\frac{1}{X})$. In general $\texttt{PLB}(A)$ is applied more than once in order to compute some c_i. However this number is very small [5, 1]. Eq. (1) implies that the probability that a partial quotient is ≤ 10 is ~ 0.87, thus in general the partial quotients are of small magnitude. In order to implement PLB we set $\texttt{PLB}(A) = 2 \max_{a_j<0} |\frac{a_j}{a_d}|^{1/j}$, which is nearly optimal [18]. Actually this bound "[...] is to be recommended among all"[31]. In our implementation we compute PLB only as powers of 2 so that we can take advantage of fast operations as in [27]. Notice that PLB is not a general bound on the roots, but a bound on the positive roots only, see [18, 28].

5 Implementation and Experiments

We have implemented CF in SYNAPS [23], which is a C++ library for symbolic-numeric computations. The implementation is based on GMP[3] (v. 4.1.4) and uses only transformations of the form $X \mapsto 2^\beta X$ and $X \mapsto X + 1$. We consider square-free polynomials of degree $\in \{100, 200, \ldots, 1000\}$. Following [27], the first class of experiments concerns well-known ill-conditioned polynomials: Laguerre (L), first (C1) and second (C2) kind Chebyshev, and Wilkinson (W) polynomials. We also consider Mignotte (M1) polynomials $X^d - 2(101X - 1)^2$, that have 4 real roots but two of them very close together, and products, $\left(X^d - 2(101X - 1)^2\right)\left(X^d - 2((101 + \frac{1}{101})X - 1)^2\right)$, of two such polynomials (M2). Finally, we consider polynomials with random coefficients (R1), and monic polynomials with random coefficients (R2) in the range [-1000, 1000], produced by MAPLE, using 101 as a seed for the pseudo-random number generator.

We performed experiments against RS that implements a subdivision-based algorithm using Descartes' rule of sign with several optimizations and symbolic-numeric techniques [27]. We used RS through its MAPLE interface and with default options. Timings were reported by its function `rs_time()`. We also test ABERTH [6], a numerical solver with unknown (bit) complexity but very efficient in practice, available through SYNAPS. In particular, it uses multi-precision floats and provides a floating-point approximation of all (real and complex) roots. Unfortunately, we were not always able to tune its behavior in order to produce the correct number of real roots in all the cases.

So, in Table 1, we report experiments with CF, RS and ABERTH, where the timings are in seconds. The asterisk (*) denotes that the computation did not

[3] www.swox.com/gmp/

finish after 12000s and the question-mark (?) that we were not able to tune ABERTH. The experiments were performed on a 2.6GHz PIII with 1GB RAM, using g++ 3.3 with option -O3.

For (M1) and (M2), there are rational numbers with a very simple continued fraction expansion that isolate the real roots which are close. These experiments are extremely hard for RS. On (M1), ABERTH is the fastest and correctly computes all real roots, but on (M2), which has 4 real roots close together, it is slower than CF. CF is advantageous on (W) since, as soon as a real root is found, transformations of the form $X \mapsto X+1$ rapidly produce the other real roots. We were not able to tune ABERTH on (W). For (L), (C1) and (C2), CF is comparable to RS, while we were not able to appropriately tune ABERTH to produce the correct number of real roots. The polynomials in (R1) and (R2) have few and well separated roots, thus the semi-numerical techniques of RS isolate all roots using only 63 bits of accuracy. ABERTH is even faster on these experiments. However, even in this case, CF is only a little slower than ABERTH. Finally, we tested a univariate polynomial that appears in the Voronoi diagram of ellipses [16]. The polynomial has degree 184, coefficient bitsize 903, and 8 real roots. CF solves it in 0.12s, RS in 0.3s and ABERTH in 1.7s. We have to mention, as F. Rouillier pointed out to us, that RS can be about 30% faster in (L), (C1) and (C2), if we use it with the (non-default) option `precision=0`.

Acknowledgments. Both authors acknowledge fruitful discussions with A. Akritas and B. Mourrain. The first author is also grateful to M. Mignotte, F. Rouillier and D. Stefanecu for various discussions and suggestions. Both authors acknowledge partial support by IST Programme of the EU as a Shared-cost RTD (FET Open) Project under Contract No IST-006413-2 (ACS - Algorithms for Complex Shapes).

References

1. A. Akritas. An implementation of Vincent's theorem. *Numerische Mathematik*, 36:53–62, 1980.
2. A. Akritas. There is no "Uspensky's method". Extended Abstract. In *Proc. Symp. on Symbolic and Algebraic Computation*, pp. 88–90, Waterloo, Canada, 1986.
3. A. Akritas, A. Bocharov, and A. Strzébonski. Implementation of real root isolation algorithms in Mathematica. *Abstracts of Interval'94*, pp. 23–27, Russia, 1994.
4. A. Akritas and A. Strzebonski. A comparative study of two real root isolation methods. *Nonlinear Analysis: Modelling and Control*, 10(4):297–304, 2005.
5. A.G. Akritas. *Elements of Computer Algebra with Applications*. J. Wiley & Sons, New York, 1989.
6. D. Bini and G. Fiorentino. Design, analysis, and implementation of a multiprecision polynomial rootfinder. *Numerical Algorithms*, pp. 127–173, 2000.
7. E. Bombieri and A. van der Poorten. Continued fractions of algebraic numbers. In *Computational algebra and number theory*, pp. 137–152. Kluwer, Dordrecht, 1995.
8. R. Brent, A. van der Poorten, and H. Riele. A comparative study of algorithms for computing continued fractions of algebraic numbers. In Henri Cohen, editor, *ANTS*, volume 1122 of *LNCS*, pp. 35–47. Springer, 1996.

9. G. Collins and A. Akritas. Polynomial real root isolation using Descartes' rule of signs. In *SYMSAC '76*, pp. 272–275, New York, USA, 1976. ACM Press.
10. G.E. Collins and R. Loos. Real zeros of polynomials. In B. Buchberger, G.E. Collins, and R. Loos, editors, *Computer Algebra: Symbolic and Algebraic Computation*, pp. 83–94. Springer-Verlag, Wien, 2nd edition, 1982.
11. J. H. Davenport. Cylindrical algebraic decomposition. Technical Report 88–10, School of Mathematical Sciences, University of Bath, England, 1988.
12. Z. Du, V. Sharma, and C. K. Yap. Amortized bound for root isolation via Sturm sequences. In D. Wang and L. Zhi, editors, *Int. Workshop on Symbolic Numeric Computing*, pp. 81–93, School of Science, Beihang University, Beijing, China, 2005.
13. A. Eigenwillig, V. Sharma, and C. Yap. Almost tight complexity bounds for the Descartes method. (to appear in ISSAC 2006), 2006.
14. I. Emiris and E. P. Tsigaridas. Computations with one and two algebraic numbers. Technical report, ArXiv, Dec 2005.
15. I. Z. Emiris, B. Mourrain, and E. P. Tsigaridas. Real Algebraic Numbers: Complexity Analysis and Experimentation. RR 5897, INRIA, Apr 2006.
16. I.Z. Emiris, E.P. Tsigaridas, and G.M. Tzoumas. The predicates for the Voronoi diagram of ellipses. In *Proc. 24th Annual ACM SoCG*, pp. 227–236, 2006.
17. A. Khintchine. *Continued Fractions*. University of Chicago Press, Chicago, 1964.
18. J. Kioustelidis. Bounds for the positive roots of polynomials. *Journal of Computational and Applied Mathematics*, 16:241–244, 1986.
19. W. Krandick and K. Mehlhorn. New bounds for the Descartes method. *JSC*, 41(1):49–66, Jan 2006.
20. M. Mignotte. *Mathematics for computer algebra*. Springer-Verlag, New York, 1991.
21. M. Mignotte and D. Stefanescu. *Polynomials*. Springer, 1999.
22. M. Mignotte. On the Distance Between the Roots of a Polynomial. *Appl. Algebra Eng. Commun. Comput.*, 6(6):327–332, 1995.
23. B. Mourrain, J. P. Pavone, P. Trébuchet, and E. Tsigaridas. SYNAPS, a library for symbolic-numeric computation. In *8th MEGA*, Italy, 2005. Software presentation.
24. V. Pan. Solving a polynomial equation: Some history and recent progress. *SIAM Rev.*, 39(2):187–220, 1997.
25. R. Richtmyer, M. Devaney, and N. Metropolis. Continued fraction expansions of algebraic numbers. *Numerische Mathematik*, 4:68–64, 1962.
26. D. Rosen and J. Shallit. A continued fraction algorithm for approximating all real polynomial roots. *Math. Mag*, 51:112–116, 1978.
27. F. Rouillier and Z. Zimmermann. Efficient isolation of polynomial's real roots. *J. of Computational and Applied Mathematics*, 162(1):33–50, 2004.
28. D. Stefanescu. New bounds for the positive roots of polynomials. *Journal of Universal Computer Science*, 11(12):2132–2141, 2005.
29. J. V. Uspensky. *Theory of Equations*. McGraw-Hill, 1948.
30. A. van der Poorten. An introduction to continued fractions. In *Diophantine analysis*, pp. 99–138. Cambridge University Press, 1986.
31. A. van der Sluis. Upper bounds for the roots of polynomials. *Numerische Mathematik*, 15:250–262, 1970.
32. A. J. H. Vincent. Sur la résolution des équations numériques. *J. Math. Pures Appl.*, 1:341–372, 1836.
33. J. von zur Gathen and J. Gerhard. Fast Algorithms for Taylor Shifts and Certain Difference Equations. In *ISSAC*, pp. 40–47, 1997.
34. C.K. Yap. *Fundamental Problems of Algorithmic Algebra*. Oxford University Press, New York, 2000.

Exact and Efficient Construction of Planar Minkowski Sums Using the Convolution Method*

Ron Wein

School of Computer Science,
Tel-Aviv University, Israel
wein@post.tau.ac.il

Abstract. The Minkowski sum of two sets $A, B \in \mathbb{R}^d$, denoted $A \oplus B$, is defined as $\{a + b \mid a \in A, b \in B\}$. We describe an efficient and robust implementation for the construction of Minkowski sums of polygons in $\mathbb{R}^2$ using the *convolution* of the polygon boundaries. This method allows for faster computation of the sum of non-convex polygons in comparison with the widely-used methods for Minkowski-sum computation that decompose the input polygons into convex sub-polygons and compute the union of the pairwise sums of these convex sub-polygon.

1 Introduction

Given two sets $A, B \in \mathbb{R}^d$, their *Minkowski sum*, denoted by $A \oplus B$, is the set $\{a + b \mid a \in A, b \in B\}$. Minkowski sums are used in many applications, such as motion planning and computer-aided design and manufacturing. In this paper we focus on an important sub-class of the planar Minkowski-sum computation problem: computing the sum of two simple polygons.

If P and Q are simple planar polygons having m and n vertices respectively, then $P \oplus Q$ is a subset of the arrangement of $O(mn)$ line segments, where each segment is the Minkowski sum of an edge of P with a vertex of Q, or vice-versa. The size of the sum is therefore bounded by $O(m^2n^2)$, and this bound is tight [11]. However, if both P and Q are convex, then $P \oplus Q$ is a convex polygon with at most $m+n$ vertices, and can be computed in $O(m+n)$ time (see, e.g., [3, Chap. 13]). If only P is convex, the Minkowski sum of P and Q is bounded by $O(mn)$ [12], and this bound is tight as well.

As we mentioned above, computing the Minkowski sum of two convex polygons can be performed in linear time using a simple procedure that can be easily implemented in software. The prevailing method for computing the sum of two non-convex polygons P and Q, is therefore based on *convex decomposition*: we decompose P into convex sub-polygons $P_1, \ldots, P_k$ and Q into convex sub-polygons $Q_1, \ldots, Q_\ell$, obtain the Minkowski sum of each pair of sub-polygons and compute the union of the $k\ell$ pairwise sub-sums. Namely, we calculate $P \oplus Q = \bigcup_{i,j} (P_i \oplus Q_j)$. Flato [4] (see also [1]), implemented a software

* Partially supported by the IST Programme of the EU as a Shared-cost RTD (FET Open) Project under Contract No IST-006413 (ACS - Algorithms for Complex Shapes), and by the Hermann Minkowski–Minerva Center for Geometry at Tel Aviv University.

Y. Azar and T. Erlebach (Eds.): ESA 2006, LNCS 4168, pp. 829–840, 2006.

package for computing Minkowski sums of planar polygons in an exact manner, based on the decomposition method. He implemented about a dozen different polygon-decomposition strategies and several methods for the union computation, and conducted thorough experiments to determine the optimal decomposition and union strategies. Flato's code is robust and produces exact results. It is based on CGAL Version 2.0,[1] and uses exact rational arithmetic. This is the first implementation capable of handling degenerate inputs, and the only one that correctly identifies low-dimensional elements of the Minkowski sum, such as antennas or isolated vertices (see more details in Sec. 3.2). The LEDA library [13] also contains functions for robust Minkowski-sum computation based on convex polygon decomposition that use exact rational arithmetic.[2] However, these functions are limited to performing *regularized* Minkowski-sum computations, which eliminate low-dimensional features of the output.

Another approach to computing the Minkowski sum of two polygons is calculating the *convolution* of the boundaries of P and Q [7,8]. Ramkumar [15] used this approach to devise an efficient algorithm for computing the outer boundary of the Minkowski sum of two polygons.[3] To the best of our knowledge, our code is the first implementation of software for robust and exact computation of Minkowski sums that is based on the convolution method. As our experiments show, using the convolution method we construct intermediate geometric entities that are more compact than the ones constructed using the decomposition method. Consequently, we are able to obtain faster running times.

The rest of this paper is organized as follows: In Sec. 2 we review the definition of polygon convolution and develop the notation that will be used throughout the paper. In Sec. 3 we give the details of our implementation of the Minkowski-sum algorithm. We present experimental results in Sec. 4, and give some concluding remarks in Sec. 5.

2 Preliminaries

Guibas *et al.* [7] introduced the concept of convolutions of general *planar tracings*, giving a special attention to *polygonal tracings*, which are composed of a series of interleaved *moves* (translations in a fixed direction) and *turns* (rotations at a fixed location).

Given two polygons, P with vertices $(p_0, \ldots, p_{m-1})$ and Q with vertices $(q_0, \ldots, q_{n-1})$, we make a move by traversing a polygon edge $\overrightarrow{p_i p_{i+1}}$, and make a turn be rotating on a polygon vertex p_i from the direction of $\overrightarrow{p_{i-1} p_i}$ to the of direction $\overrightarrow{p_i p_{i+1}}$.[4] Without loss of generality, we can assume that both polygons are counterclockwise oriented. The *convolution* of these two polygons, denoted

[1] See CGAL's homepage at `http://www.cgal.org`.

[2] For more details and a detailed on-line documentation, see: `http://www.algorithmic-solutions.info/leda_guide/geo_algs/minkowski.html`.

[3] The complexity of this boundary is $O(mn \cdot \alpha(n))$, where $\alpha(\cdot)$ is the functional inverse of Ackermann's function [9].

[4] Throughout this paper, when we increment or decrement an index of a vertex, we do so modulo the size of the polygon. Indeed, $\overrightarrow{p_{m-1} p_0}$ is also a valid polygon edge.

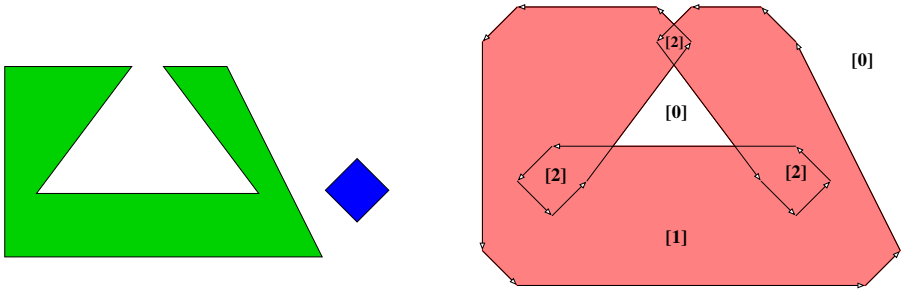


Fig. 1. Computing the convolution of a convex polygon and a non-convex polygon (left). The convolution consists of a single self-intersecting cycle, drawn as a sequence of arrows (right). The winding number associated with each face of the arrangement induced by the segments forming the cycle appears in brackets. The Minkowski sum of the two polygons is shaded.

$P * Q$, is a collection of line segments of the form $\overrightarrow{(p_i + q_j)(p_{i+1} + q_j)}$, where the vector $\overrightarrow{p_i p_{i+1}}$ lies between $\overrightarrow{q_{j-1} q_j}$ and $\overrightarrow{q_j q_{j+1}}$,[5] and — symmetrically — of segments of the form $\overrightarrow{(p_i + q_j)(p_i + q_{j+1})}$, where the vector $\overrightarrow{q_j q_{j+1}}$ lies between $\overrightarrow{p_{i-1} p_i}$ and $\overrightarrow{p_i p_{i+1}}$. We can label the convolution segment as $\langle (i, i+1), j \rangle$ in the former case or $\langle i, (j, j+1) \rangle$ in the latter case. From the definition, it is clear that $P * Q$ contains at most $O(mn)$ line segments.

The segments of the convolution form a number of closed (not necessarily simple) polygonal curves called *convolution cycles*. The Minkowski sum $P \oplus Q$ is the set of points having a non-zero winding number with respect to these cycles (see, e.g., [14, Chap. 7] for the topological definition of the *winding number* and some examples). See Fig. 1 for an illustration.

In case both input polygons P and Q are convex, their convolution is a convex polygonal tracing. If only one polygon (say P) is convex, then $P * Q$ still contains a single cycle, which may not be simple (see [15] for a proof), as illustrated in Fig. 1. If both P and Q are non-convex, the convolution may comprise several cycles, and in order to compute the Minkowski sum of the polygons one has to consider the set of points having a non-zero winding number with respect to all cycles (see Fig. 2 for an illustration).

3 Implementation Details

Given two simple polygons P and Q having m and n vertices, respectively, we compute their Minkowski sum in three steps: first we compute the cycles of the convolution $P * Q$, then we construct the planar arrangement induced by the segments that form the convolution cycles, and finally we extract the Minkowski sum from this arrangement.

[5] We say that a vector $\boldsymbol{v}$ lies between two vectors $\boldsymbol{u}$ and $\boldsymbol{w}$, if we reach $\boldsymbol{v}$ strictly before reaching $\boldsymbol{w}$ if we move all three vectors to the origin and rotate $\boldsymbol{u}$ counterclockwise. Note that this also covers the case where $\boldsymbol{u}$ has the same direction as $\boldsymbol{v}$.

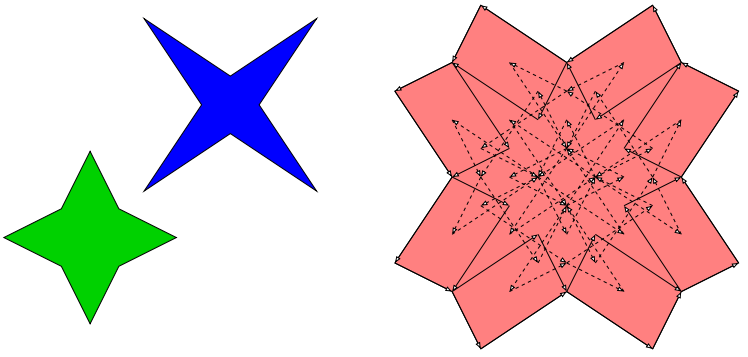


Fig. 2. Computing the convolution of two non-convex octagons (left). The convolution consists of two cycles (right), one (solid arrows) comprises 32 line segments while the other (dashed arrows) contains 48 line segments, non of which lies on the boundary of the Minkowski sum (shaded).

3.1 Computing the Convolution Cycles

Guibas and Seidel [8] show how to compute the convolution cycles of two polygons in optimal $O(m+n+K)$ time, where $K = |P * Q|$. However, they make some general-position assumptions on the polygons (e.g., an edge of P cannot have the same direction as an edge in Q) and cannot handle degenerate inputs. In this section we present a simple and robust algorithm, whose asymptotic running time is $O(m+n+\min\{m_r n, n_r m\}+K)$, where m_r and n_r are the number of reflex vertices in P and Q, respectively. As our experiments show, the running time of the construction of the convolution cycles is in practice negligible with respect to the overall Minkowski-sum computation.

We start by describing a simple procedure that constructs a single convolution cycle $\mathcal{C}$ of two polygons $P = (p_0, \ldots p_{m-1})$ and $Q = (q_0, \ldots q_{n-1})$, starting from two vertices p_{i_0} and q_{j_0}, such that $p_{i_0} + q_{j_0}$ is a vertex on the cycle $\mathcal{C}$. See the pseudo-code listing of the procedure ComputeConvolutionCycle. Observe that in the main loop of the procedure we add at least one convolution segment each iteration. However, in degenerate situations, namely when $\overrightarrow{p_i, p_{i+1}}$ and $\overrightarrow{q_j, q_{j+1}}$ have the same direction, we add two segments to the cycle in a single iteration.

We next describe how to locate the pairs of indices i_0 and j_0 needed for the procedure above. We start by locating the bottommost vertex of P (the minimal vertex with respect to a yx-lexicographical order) and the bottommost vertex of Q. Assume, without loss of generality, that these are the vertices p_0 and q_0. These vertices are not reflex, and it is clear that either $\overrightarrow{p_0 p_1}$ lies between the edges around q_0, or vice-versa. We can therefore compute a convolution cycle starting from this pair of vertices.

If either of the polygons is convex (that is, $m_r = n_r = 0$), then the convolution consists of a single cycle, and we are done. Otherwise, we traverse the reflex vertices of Q (we assume, without loss of generality that $n_r m < m_r n$), and for

ComputeConvolutionCycle $(P, i_0;\ Q, j_0)$

let $\mathcal{C} \longleftarrow \varnothing$.
let $i \longleftarrow i_0$. **let** $j \longleftarrow j_0$.
let $s \longleftarrow (p_i + q_j)$.
do:
  let *inc_P* $\longleftarrow$ IsBetweenCounterclockwise $(\overrightarrow{p_i, p_{i+1}};\ \overrightarrow{q_{j-1}q_j}, \overrightarrow{q_j q_{j+1}})$.
  let *inc_Q* $\longleftarrow$ IsBetweenCounterclockwise $(\overrightarrow{q_j, q_{j+1}};\ \overrightarrow{p_{i-1}p_i}, \overrightarrow{p_i p_{i+1}})$.
  if *inc_P* = True, **then**:
    let $t \longleftarrow (p_{i+1} + q_j)$.
    Push the segment ***st***, labelled $\langle (i, i+1), j \rangle$, into $\mathcal{C}$.
    let $s \longleftarrow t$.
    let $i \longleftarrow i + 1$ (modulo the size of P).
  if *inc_Q* = True, **then**:
    let $t \longleftarrow (p_i + q_{j+1})$.
    Push the segment ***st***, labelled $\langle i, (j, j+1) \rangle$, into $\mathcal{C}$.
    let $s \longleftarrow t$.
    let $j \longleftarrow j + 1$ (modulo the size of Q).
while $i \neq i_0$ **or** $j \neq j_0$.
return $\mathcal{C}$.

each such vertex q_{j_0} we go over all vertices of P and try to locate a vertex p_{i_0}, such that $\overrightarrow{p_{i_0} p_{i_0+1}}$ lies between $\overrightarrow{q_{j_0-1} q_{j_0}}$ and $\overrightarrow{q_{j_0} q_{j_0+1}}$. When we locate such a vertex pair and such that the label $\langle (i_0, i_0+1), j_0 \rangle$ has not been used (for this purpose we maintain an auxiliary set of *used labels*; this set can be represented using a hash table, such that each access to the set takes $O(1)$ time on average), we invoke the procedure above to compute an additional convolution cycle.

3.2 Obtaining the Minkowski Sum from the Convolution Cycles

Flato [4, App. A.1] describes a simple algorithm for computing the union of a set of simple polygons having counterclockwise orientation. He constructs the arrangement of the directed segments that correspond to the polygon edges (referred to as the *boundary segments*), namely the subdivision they induce on the plane into one unbounded face and several bounded faces. The arrangement is represented using a doubly-connected edge-list (Dcel for short; see, e.g., [3, Chap. 2]), such that it is easy to perform a breadth-first search traversal of all arrangement faces, starting from the unbounded face (whose winding number is of course 0), and compute the winding number of each face. The same arguments used for proving the correctness of the polygon-union algorithm in [4, App. A.1] also hold for the case of convolution cycles. We can therefore construct the arrangement of all directed segments constituting the convolution cycles, compute the winding numbers of the arrangement faces and output the union of the faces with a positive winding number. Consider for example Fig. 1, where we correctly identify the hole in the Minkowski sum, as it is represented by a face whose winding number is 0.

So far we have described an algorithm that computes the regularized Minkowski sum $P \hat{\oplus} Q$, namely the closure of the interior of $P \oplus Q$, as all 1-dimensional or

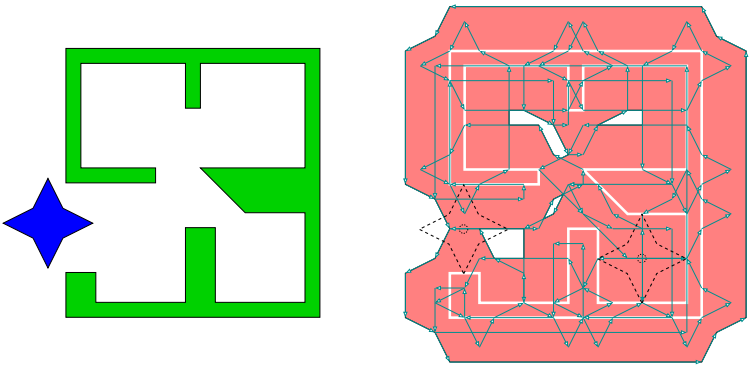


Fig. 3. A house plan and a star-shaped polygon (left). The Minkowski sum of the two polygons (right) consists of low-dimensional features. For clarity, two copies of the star are drawn using a dashed line with their center positioned on these features. The left copy is located on an *antenna* on the Minkowski-sum boundary, such that the star can move along this antenna while touching the walls of the house. The right copy is located on an *isolated vertex*, which designates a location where the star can be placed without being able to move.

0-dimensional features of the sum (*antennas* and *isolated vertices*, respectively) are disregarded. In this case, the output is given as a polygon that represents the outer boundary of the sum and an additional, possibly empty, set of polygons that represents the holes inside this polygon. This representation is sufficient for many applications, but for other applications, such as motion planning and assembly planning, low-dimensional features may play an important role as they represent *tight passages*. Consider the example depicted in Fig. 3, where we wish to move a star-shaped polygon in a house, allowing translations only. The interior of the Minkowski sum in this case consists of all forbidden placements for the star center, such that each point on the boundary of the sum corresponds to a *semi-free* placement of the star, where it touches the walls but does not penetrate them.

Our software is also capable of outputting a planar arrangement that captures all features of the Minkowski sum of the input polygons.[6] Locating the low-dimensional features incurs no asymptotic run-time penalty; see [4] for the full details of the algorithm including a proof of correctness.

4 Experimental Results

We have adapted parts of Flato's software [4] to comply with the interface of the latest version of CGAL (Version 3.2). As the original software consists of several thousands of lines of code, we did not convert all polygon-decomposition strategies; instead, we use the three convex-decomposition algorithm bundled with the Polygon Partitioning package of the CGAL public release. In addition,

[6] Unlike previous CGAL versions, Version 3.2 of CGAL's arrangement package supports isolated vertices, so we can have 0-dimensional features in the output as well.

we added the small-side angle-bisector decomposition strategy (see below). In his experiments, Flato reports this strategy as the one that yields the fastest running times for the overall Minkowski-sum computation process (see also [1]). We ran our experiments with all four strategies listed below.

Optimal (Opt). The dynamic-programming algorithm of Greene [6] for computing an optimal decomposition of a polygon into a minimal number of convex sub-polygons. The main drawback of this strategy is that it runs in $O(n^4)$ time and requires $O(n^3)$ space in the worst case, where n is the number of vertices in the input polygon.

Hertel–Mehlhorn (HM). The approximation algorithm suggested by Hertel and Mehlhorn [10], which triangulates the input polygon and proceeds by throwing away unnecessary triangulation edges. This algorithm requires $O(n)$ time and space and guarantees that the number of sub-polygons it generates is not more than four times the optimum.

Greene (Gre.). Greene's approximation algorithm [6] which computes a convex decomposition of the polygon based on its partitioning into y-monotone polygons. This algorithm runs in $O(n \log n)$ time and $O(n)$ space, and has the same approximation guarantee as Hertel and Mehlhorn's algorithm.

Small-side angle-bisector (SSAB). A heuristic improvement to the angle-bisector decomposition method suggested by Chazelle and Dobkin [2]. It starts by examining each pair of reflex vertices in the input polygon such that the entire interior of the diagonal connecting these vertices is contained in the polygon. Out of all available pairs, it selects p_i and p_j, such that the number of reflex vertices from p_i to p_j (or from p_j to p_i) is minimal. The polygon is split by the diagonal p_ip_j and the process continues recursively on both resulting sub-polygons. In case it is not possible to eliminate two reflex vertices at once any more, each reflex vertex is eliminated by an angle bisector emanating from it. The entire process takes $O(n^2)$ time.

In the original implementation, the intersections between the angle bisectors and the polygon edges induce *Steiner points* in the decomposed sub-polygons. We have slightly modified the algorithm, such that instead of eliminating a reflex vertex p_i using an angle bisector, we look for another vertex p_{j^*}, such that $p_ip_{j^*}$ is contained in the polygon, and such that the ratio between the two angles $\measuredangle p_{i-1}p_ip_{j^*}$ and $\measuredangle p_{j^*}p_ip_{i+1}$ that $p_ip_{j^*}$ induces is as close to 1 as possible. Our experiments show that this modified approach yields very good decompositions, while avoiding the introduction of Steiner points, which may lead to more complex computations.

The original Minkowski-sum software was based on the arrangement package of CGAL Version 2.0, which supported only the *incremental* construction of arrangements, inserting curves one at a time. In the current CGAL version, it is possible to construct arrangements *aggregately*, so all boundary segments are inserted together using a sweep-line algorithm. Constructing an arrangement aggregately is asymptotically more efficient for line segments that sparsely intersect, as happens in our case. This theoretical argument is also backed up by

experiments that show that constructing an arrangement of line segments aggregately is usually one order of magnitude faster than constructing it incrementally. We have therefore implemented an aggregated version of the union algorithm, as described in Sec. 3.2, and did not try the incremental union algorithm, as described in [4, Chap. 3].

We have conducted a large number of tests with various input sets. Here we report on eight representative input sets containing polygon pairs, most of them taken from [1] (see Figs 4 and 5 for illustrations). All data sets are available on-line at `http://www.cs.tau.ac.il/~wein/software/`:

Chain: A chain-shaped polygon with 82 vertices (37 of them are reflex), and a star-shaped polygon with 30 vertices (6 reflex).

Stars: Two star-shaped polygons, each containing 40 vertices (14 reflex).

Comb: A comb-shaped polygon containing 53 vertices (24 reflex) and a convex polygon with 22 vertices. This input set introduces the worst-case complexity for the sum of a convex and a non-convex polygon.

Fork: The well-known example for a Minkowski sum of size $O(m^2n^2)$ [11]. The large "fork" consists of 34 vertices (19 reflex) and the smaller one has 31 vertices (18 reflex).

Cavity: A random-looking polygon with 22 vertices (10 reflex) and a small convex octagon, chosen to fit some of the cavities in the larger polygon.

Random: Two random-looking polygons, with 40 and 20 vertices (19 and 8 reflex vertices, respectively).

Knife: A large polygon with 64 vertices (40 reflex) and a small polygon with 12 vertices (5 reflex), that can fit into the cavities on the larger polygon.

Country: The map of Israel, represented as a polygon with 50 vertices (24 reflex), and a smaller polygon with 30 vertices (8 reflex).

Table 1 summarizes the performance of the Minkowski-sum computations for the selected input sets, using the polygon-decomposition method. We give the running times, as obtained on a Pentium IV 3 GHz machine with 2 Gb of RAM, and averaged over 100 executions for each decomposition strategy. We also indicate — for the strategy that achieved the fastest running time on each input set — the numbers of sub-polygons k and ℓ in the decompositions of the two input polygons, the total number S of segments in all $k\ell$ Minkowski sums, and the size of the arrangement (number of vertices, edges and faces) induced by the boundary segments.

It should be mentioned that the running times stated in Table 1 are sometimes two orders of magnitude faster than the ones reported in [1]. This can be partly attributed to the fact that we used a faster machine in our experiments,[7] and mostly due to the improvements in the new version of CGAL's arrangement package, which now handles intersections of line segments more efficiently [5] due to numerous improvements, such as the reduction of the number of geometric operations and predicates the arrangement-construction algorithms invoke [16]. Moreover, we use the predefined CGAL kernel, which uses interval arithmetic to

[7] Flato reports using a 500 MHz Pentium III machine.

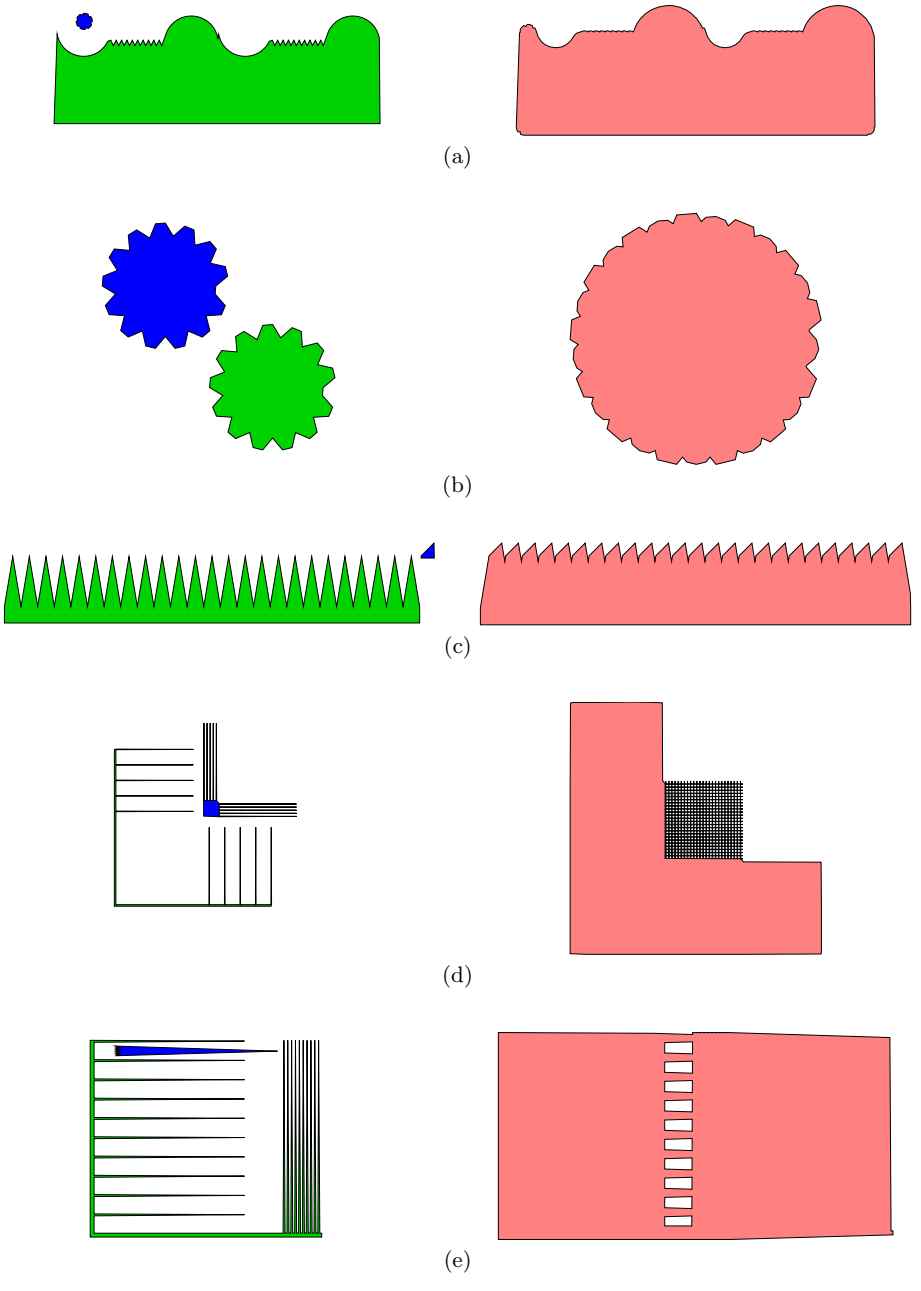


Fig. 4. Samples of input polygons (left) and their Minkowski sums (right): (a) chain; (b) stars; (c) comb; (d) fork; (e) knife

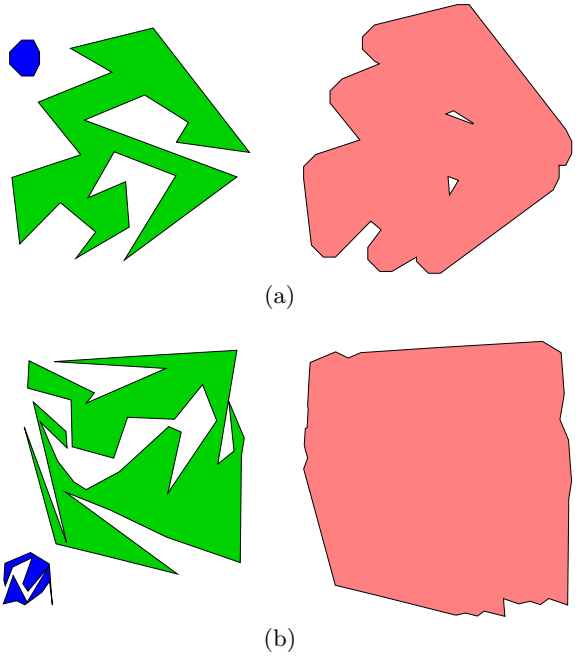


Fig. 5. Samples of input polygons (left) and their Minkowski sums (right): (a) cavity; (b) random

filter exact computations with rational numbers, and helps reducing the running times even further. The exact rational number-type we use is provided by Gnu's multi-precision library (GMP Version 4.1).

Table 2 summarizes the performance of the Minkowski-sum computations for the selected input sets using the convolution method. We indicate the numbers of convolution cycles N_c, the total number K of convolution segments and the size of the induced arrangement. We also include a column that states the running time using the fastest decomposition scheme in each case (as given in Table 1) and the running times of the Minkowski-sum function provided by LEDA (Version 4.4) on the various input sets. We used the exact rational kernel of LEDA, which also employs arithmetic filtering to speed up the computations with an exact rational number-type.

In almost all cases, the convolution methods yields faster running times than the fastest decomposition scheme (recall that LEDA also employs the polygon-decomposition method). This is attributed to the fact that the convolution method usually induces a smaller arrangement as its intermediate construct. As the total running-time is clearly dominated by the arrangement-construction time, the convolution method may achieve a speed-up of up to a factor of 5 in some cases. Moreover, the memory requirements for storing the intermediate arrangement are also considerably smaller.

The convolution method has another important advantage. As we learn from Table 1, the choice of a decomposition strategy may have drastic effects on the

Table 1. Running times (measured in milliseconds) for the Minkowski-sum computation using various decomposition strategies. The running time of the for the fastest strategy in each case is shown in bold, and its construction statistics are also given.

				Decomp.	Arrangement size			Running time			
Input set	k	ℓ	S	time	$\|V\|$	$\|E\|$	$\|F\|$	Opt.	HM	Gre.	SSAB
chain	35	7	2520	123	7239	13627	6390	**390**	424	578	444
stars	17	17	2446	13	12955	25624	12671	488	744	867	**477**
comb	26	1	650	4	671	769	100	32	58	**17**	37
fork	12	11	1048	6	6827	13377	6552	287	524	1220	**260**
cavity	14	1	160	1	167	244	79	8	7	**6**	8
random	20	10	1540	15	8209	16310	8103	**234**	349	376	320
knife	22	6	1108	9	7721	15150	7431	249	370	1630	**232**
country	16	8	1344	8	5126	9772	4648	338	798	410	**188**

Table 2. Running times (measured in milliseconds) and construction statistics for the Minkowski-sum computation using the convolution method

			Conv.	Arrangement size			Total	Best	Using
Input set	N_c	K	time	$\|V\|$	$\|E\|$	$\|F\|$	time	decomp.	LEDA
chain	1	1452	7	2077	2868	793	**69**	390	463
stars	2	1200	7	3225	5482	2259	**92**	477	332
comb	1	603	7	627	651	26	**17**	17	97
fork	1	1266	5	11203	22063	10862	**521**	260	266
cavity	1	110	1	135	161	28	**4**	6	21
random	1	580	3	2589	4698	2111	**62**	234	229
knife	1	876	5	5075	9749	4676	**184**	232	175
country	2	1050	5	1940	3064	1126	**61**	188	275

running time of the Minkowski-sum computation, and the best strategy can only be found by experimenting; when using the convolution method, no such experiments are needed.

The *fork* input set is the only example that exhibits slower running times when using the convolution method. This is due to the fact that the input polygons are decomposed very nicely by the SSAB decomposition-strategy, separating each of the fork prongs into a single thin rectangle, such that the Minkowski sums of the sub-polygons are nearly pairwise disjoint. On the other hand, as we have many pairs of parallel edges in the input polygons (note that all edges of the two forks are axes-parallel), the convolution cycle in this case contains many redundant loops that incur the greater run-time overhead.

5 Conclusions

We present an efficient implementation for computing Minkowski sums of simple polygons using the convolution method. This is the first software implementation of a robust algorithm based on the polygon-convolution method. It uses

exact computation, and can handle inputs with many degeneracies, yielding topologically correct results. As our experiments show, the convolution method is superior to the polygon-decomposition method on almost all input sets, and improves the running times by a factor of 2–5. We intend to include our software in the next public release of CGAL (the forthcoming Version 3.3).

References

1. P. K. Agarwal, E. Flato, and D. Halperin. Polygon decomposition for efficient construction of Minkowski sums. *Computational Geometry: Theory and Applications*, **21**:39–61, 2002.
2. B. Chazelle and D. P. Dobkin. Optimal convex decompositions. In G. T. Toussaint, editor, *Computational Geometry*, pages 63–133. North-Holland, Amsterdam, The Netherlands, 1985.
3. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, Germany, 2nd edition, 2000.
4. E. Flato. Robust and efficient construction of planar Minkowski sums. Master's thesis, School of Computer Science, Tel-Aviv University, 2000. `http://www.cs.tau.ac.il/CGAL/Theses/flato/thesis/`.
5. E. Fogel, R. Wein, and D. Halperin. Code flexibility and program efficiency by genericity: Improving CGAL's arrangements. In *Proc. 12th Europ. Sympos. Alg.*, volume **3221** of *LNCS*, pages 664–676, 2004.
6. D. H. Greene. The decomposition of polygons into convex parts. In F. P. Preparata, editor, *Computational Geometry*, volume **1** of *Adv. Comput. Res.*, pages 235–259. JAI Press, Greenwich, Conn., 1983.
7. L. J. Guibas, L. Ramshaw, and J. Stolfi. A kinetic framework for computational geometry. In *Proc. 24th Sympos. Found. Comput. Sci.*, pages 100–111, 1983.
8. L. J. Guibas and R. Seidel. Computing convolutions by reciprocal search. *Discrete and Computational Geometry*, **2**:175–193, 1987.
9. S. Har-Peled, T. M. Chan, B. Aronov, D. Halperin, and J. Snoeyink. The complexity of a single face of a Minkowski sum. In *Proc. 7th Canad. Conf. Comput. Geom.*, pages 91–96, 1995.
10. S. Hertel and K. Mehlhorn. Fast triangulation of simple polygons. In *Proc. 4th Internat. Conf. Found. Comput. Theory*, volume **158** of *LNCS*, pages 207–218. Springer-Verlag, 1983.
11. A. Kaul, M. A. O'Connor, and V. Srinivasan. Computing Minkowski sums of regular polygons. In *Proc. 3rd Canad. Conf. Comput. Geom.*, pages 74–77, 1991.
12. K. Kedem, R. Livne, J. Pach, and M. Sharir. On the union of Jordan regions and collision-free translational motion amidst polygonal obstacles. *Discrete and Computational Geometry*, **1**:59–71, 1986.
13. K. Mehlhorn and S. Näher. LEDA*: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, UK, 2000.
14. T. Needham. *Visual Complex Analysis*. Oxford University Press, Oxford, UK, 1997.
15. G. D. Ramkumar. An algorithm to compute the Minkowski sum outer-face of two simple polygons. In *Proc. 12th Sympos. Comput. Geom.*, pages 234–241, 1996.
16. R. Wein, E. Fogel, B. Zukerman, and D. Halperin. Advanced programming techniques applied to CGAL's arrangement package. In *Proc. Wrkshp. Library-Centric Software Design*, 2005. `http://lcsd05.cs.tamu.edu/#program`.

Author Index