Andreas S. Schulz
Dorothea Wagner (Eds.)

ARCoSS

LNCS 8737

Algorithms – ESA 2014

22nd Annual European Symposium
Wroclaw, Poland, September 8–10, 2014
Proceedings

Springer

Andreas S. Schulz   Dorothea Wagner (Eds.)

Algorithms – ESA 2014

22nd Annual European Symposium
Wroclaw, Poland, September 8-10, 2014
Proceedings

Springer

Volume Editors

Andreas S. Schulz
Massachusetts Institute of Technology
Cambridge, MA, USA
E-mail: schulz@mit.edu

Dorothea Wagner
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
E-mail: dorothea.wagner@kit.edu

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-662-44776-5 e-ISBN 978-3-662-44777-2
DOI 10.1007/978-3-662-44777-2
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2014947982

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

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

This volume contains the extended abstracts selected for presentation at ESA 2014, the 22nd European Symposium on Algorithms, held in Wrocław, Poland, September 8–10, 2014, as part of ALGO 2014. The ESA series of conferences highlights recent developments in the design, analysis, engineering, and application of algorithms and data structures. Information on past symposia, including locations and LNCS volume numbers, is maintained at http://esa-symposium.org.

Ever since 2002, ESA has had two tracks, the Design and Analysis Track, aka Track A, intended for papers on the design and mathematical analysis of algorithms, and the Engineering and Application Track (Track B), for submissions dealing with real-world applications, engineering, and experimental analysis of algorithms. In response to the call for papers, the program committee for Track A received 221 submissions, 11 of which were eventually withdrawn; the Track B program committee got 48 submissions. With the help of more than 800 expert reviews and more than 400 external reviewers, the two committees selected 69 papers for inclusion in the scientific program of ESA 2014, 57 in Track A and 12 in Track B.

In addition, the symposium featured invited lectures by Thomas Rothvoß (University of Washington, Seattle, USA) and Marc van Kreveld (Utrecht University, The Netherlands).

The European Association for Theoretical Computer Science, EATCS, sponsors a best student paper award and a best paper award at ESA. A submission is eligible for the best student paper award if all authors were doctoral, master or bachelor students at the time of submission. Joshua R. Wang received the best student paper award for his work on "Space-Efficient Randomized Algorithms for K-SUM". The best paper award went to Pooya Davoodi, Jeremy T. Fineman, John Iacono, and Özgür Özkan for their paper entitled "Cache-Oblivious Persistence".

We are extremely grateful to the members of our program committees, to the external reviewers, to the local organizers, and to Cool Press Ltd., the company that owns and runs EasyChair, for all their hard work, help and support. We also thank Andreas Gemsa (Karlsruhe Institute of Technology, Germany) for his assistance in putting together the proceedings.

July 2014

Andreas S. Schulz
Dorothea Wagner

Organization

Program Committee

Design and Analysis (Track A)

Ittai Abraham	Microsoft Research Silicon Valley, USA
Amihood Amir	Bar-Ilan University, Israel
Marek Cygan	University of Warsaw, Poland
Daniel Dadush	NYU, USA
Ilias Diakonikolas	University of Edinburgh, UK
Michael Elkin	Ben-Gurion University, Israel
David Eppstein	UC Irvine, USA
Fabian Kuhn	University of Freiburg, Germany
Marc Lelarge	Inria, France
Maarten Löffler	Utrecht University, The Netherlands
Aleksander Mądry	EPFL, Switzerland
Dániel Marx	Hungarian Academy of Sciences, Hungary
Shay Mozes	IDC Herzliya, Israel
Ofer Neiman	Ben-Gurion University, Israel
Arlindo Oliveira	Lisbon University, Portugal
Laura Sanità	University of Waterloo, Canada
Andreas Schulz **(chair)**	MIT, USA
Anastasios Sidiropoulos	Ohio State University, USA
Rossano Venturini	University of Pisa, Italy
László Végh	LSE, UK
Jan Vondrák	IBM Research Almaden, USA
David Woodruff	IBM Research Almaden, USA
Rico Zenklusen	ETH, Switzerland

Engineering and Applications (Track B)

Glencora Borradaile	Oregon State University, USA
Ulrik Brandes	University of Konstanz, Germany
Kevin Buchin	TU Eindhoven, The Netherlands
Markus Chimani	University of Osnabrück, Germany
Edith Cohen	Microsoft Research Silicon Valley, USA
Giuseppe Di Battista	Third University of Rome, Italy
Dan Halperin	Tel Aviv University, Israel
Andrea Lodi	University of Bologna, Italy
Giacomo Nannicini	Singapore University of Technology and Design, Singapore

Kirk Pruhs	University of Pittsburgh, USA
Christian Sohler	TU Dortmund, Germany
Christian Sommer	Cupertino, USA
Monique Teillaud	Inria Sophia Antipolis, France
Takeaki Uno	National Institute of Informatics Tokyo (NII), Japan
Dorothea Wagner **(chair)**	Karlsruhe Institute of Technology (KIT), Germany

Additional Reviewers

Abed, Fidaa
Adamczyk, Marek
Adjiashvili, David
Afshani, Peyman
Ahmadian, Sara
Aichholzer, Oswin
Ailon, Nir
Ajwani, Deepak
Aloupis, Greg
Amit, Mika
An, Hyung-Chan
Anagnostopoulos, Aris
Andoni, Alexandr
Angelini, Patrizio
Angelopoulos, Spyros
Antoniadis, Antonios
Anunciação, Orlando
Araki, Tetsuya
Austrin, Per
Azar, Yossi
Ballard, Grey
Barenboim, Leonid
Bauer, Ulrich
Belovs, Aleksandrs
Ben Avraham, Rinat
Bevern, René Van
Beyer, Stephan
Bhaskar, Umang
Bhaskara, Aditya
Bhattacharya, Sayan
Bhattacharyya, Arnab
Biedl, Therese
Bienkowski, Marcin
Biro, Peter
Bliznets, Ivan
Bläsius, Thomas
Bock, Adrian
Bodlaender, Hans L.
Bonichon, Nicolas
Bonnet, François
Bonsma, Paul
Boria, Nicolas
Brauner, Nadia
Broutin, Nicolas
Buchin, Maike
Buriol, Luciana
Byrka, Jaroslaw
Cacchiani, Valentina
Cai, Leizhen
Cai, Yang
Calinescu, Gruia
Cao, Yixin
Carmi, Paz
Castelli Aleardi, Luca
Castro, Jorge
Cerulli, Raffaele
Chechik, Shiri
Chekuri, Chandra
Chen, Danny Z.
Chowdhury, Rezaul
Clementi, Andrea
Cohen, Ilan
Cohen, Sarel
Cohen-Steiner, David
Colin de Verdière, Éric
Cornelsen, Sabine
Costa, Alberto
Cunial, Fabio

Da Lozzo, Giordano
Daum, Sebastian
De, Anindya
Dell, Holger
Della Vedova, Gianluca
Delling, Daniel
Demetrescu, Camil
Dereniowski, Dariusz
Devillers, Olivier
Di Bartolomeo, Marco
Di Giacomo, Emilio
Didimo, Walter
Dinitz, Michael
Dlotko, Pawel
Doerr, Benjamin
Doty, David
Driemel, Anne
Duret-Lutz, Alexandre
Edwards, John
Ehsanfar, Ebrahim
El Hallaoui, Issmail
Elbassioni, Khaled
Elkind, Edith
Englert, Matthias
Epstein, Leah
Evans, William
Even, Guy
Faenza, Yuri
Fagerberg, Rolf
Farczadi, Linda
Feldman, Michal
Feldman, Moran
Feldmann, Andreas
Fertin, Guillaume
Fischer, Johannes
Fischetti, Matteo
Fleiner, Tamas
Fleszar, Krzysztof
Francisco, Alexandre
Friedrich, Tobias
Friggstad, Zachary
Fuchs, Fabian
Fujie, Tetsuya
Fukunaga, Takuro
Gagie, Travis
Galanis, Andreas
Gasieniec, Leszek
Gaspers, Serge
Gavoille, Cyril
Georgiadis, Loukas
Gkatzelis, Vasilis
Gog, Simon
Goldberg, David
Grandoni, Fabrizio
Green Larsen, Kasper
Grossi, Roberto
Gupta, Anupam
Gutwenger, Carsten
Hackl, Thomas
Haghpanah, Nima
Hajiaghayi, Mohammadtaghi
Halldorsson, Magnus M.
Halldórsson, Magnús
Har-Peled, Sariel
Hardt, Moritz
Harks, Tobias
Hassidim, Avinatan
Hedtke, Ivo
Heeringa, Brent
Hermelin, Danny
Hoefer, Martin
Hoy, Darrell
Huang, Chien-Chung
Huang, Zhiyi
Huber, Stefan
Höhn, Wiebke
Høyer, Peter
Iacono, John
Im, Sungjin
Imahori, Shinji
Italiano, Giuseppe F.
Ito, Takehiro
Jacob, Riko
Jaillet, Patrick
Jaklin, Norman
Jansen, Bart M.P.
Jerrum, Mark
Jeż, Łukasz
Jin, Ruoming
Kamei, Sayaka

Kamiński, Marcin
Kamma, Lior
Kane, Daniel
Kaplan, Haim
Kapralov, Michael
Karrenbauer, Andreas
Katoh, Naoki
Kavitha, Telikepalli
Kerber, Michael
Kern, Walter
Kesselheim, Thomas
Khandwawala, Mustafa
Khanna, Neelesh
Kierstad, Hal
Kim, Eun Jung
Kiraly, Tamas
Kiss, Sándor
Klein, Kim-Manuel
Klein, Shmuel Tomi
Knauer, Christian
Knust, Sigrid
Kobayashi, Koji M.
Kobayashi, Yusuke
Koike, Atsushi
Komosa, Paweł
Kontogiannis, Spyros
Kopelowitz, Tsvi
Kopparty, Swastik
Kortsarz, Guy
Korula, Nitish
Koutis, Ioannis
Kowalik, Lukasz
Kowalski, Dariusz
Kral, Daniel
Kranakis, Evangelos
Krauthgamer, Robert
Kreutzer, Stephan
Kriege, Nils
Krinninger, Sebastian
Kucherov, Gregory
Kumar, Amit
Kärkkäinen, Juha
Köhler, Ekkehard
Laekhanukit, Bundit
Laguna, Pablo
Lampis, Michael
Larkin, Daniel
Le, Hung
Leconte, Mathieu
Lee, Troy
Levin, Asaf
Lewi, Kevin
Li, Fei
Li, Shi
Liberti, Leo
Lim, Sejoon
Lodi, Andrea
Lokshtanov, Daniel
Lopez-Ortiz, Alejandro
Louis, Anand
Lovett, Shachar
Lucarelli, Giorgio
Ludovica, Adacher
M.S., Ramanujan
Mach, Lukas
Mahini, Hamid
Makowsky, Johann
Malec, David
Mambelli, Francesco
Maneva, Elitza
Manocha, Dinesh
Mastrolilli, Monaldo
Mathieu, Claire
Matuschke, Jannik
Mccauley, Samuel
McGregor, Andrew
Megow, Nicole
Mestre, Julian
Meyer, Ulrich
Mihalák, Matúš
Miltersen, Peter Bro
Mitani, Jun
Mittal, Shashi
Miyamoto, Yuichiro
Mnich, Matthias
Moldenhauer, Carsten
Morić, Filip
Mosteiro, Miguel
Mucha, Marcin
Mueller, Rudolf

Mulzer, Wolfgang
Nayyeri, Amir
Nederlof, Jesper
Nekrich, Yakov
Newport, Calvin
Nguyen, Huy
Nguyen, Thanh
Nicosia, Gaia
Niedermann, Benjamin
Nikolov, Aleksandar
Nöllenburg, Martin
Okamoto, Yoshio
Olver, Neil
Orlin, James
Osipov, Vitaly
Ossona De Mendez, Patrice
Ottaviano, Giuseppe
Oveis Gharan, Shayan
Ozkan, Ozgur
Pajor, Thomas
Panagiotou, Konstantinos
Panahi, Fateneh
Panigrahi, Debmalya
Paparas, Dimitris
Parriani, Tiziano
Parter, Merav
Paschos, Vangelis
Pasqualetti, Fabio
Pathak, Vinayak
Patil, Manish
Paulusma, Daniel
Peis, Britta
Pettie, Seth
Pfetsch, Marc
Phillips, Jeff
Piliouras, Georgios
Pilipczuk, Marcin
Pilipczuk, Michal
Pilz, Alexander
Poggi, Marcus
Poloczek, Matthias
Pontarelli, Salvatore
Porat, Ely
Pszona, Paweł
Puglisi, Simon
Pérennes, Stéphane
Radhakrishnan, Jaikumar
Raj Tiwary, Hans
Raman, Rajeev
Ranade, Abhiram
Rao, Michael
Rebennack, Steffen
Reem, Daniel
Reidl, Felix
Reyzin, Lev
Roditty, Liam
Roeloffzen, Marcel
Roma, Nuno
Roselli, Vincenzo
Rote, Günter
Russo, Luís
Rusu, Irena
Rutter, Ignaz
Röglin, Heiko
Saha, Barna
Saitoh, Toshiki
Sankowski, Piotr
Sarne, David
Sasakawa, Hirohito
Saumell, Maria
Saurabh, Saket
Schmid, Stefan
Schmidt, Daniel
Schwartz, Roy
Schwiegelshohn, Chris
Sen, Siddhartha
Shah, Rahul
Shaharabani, Doron
Shalev-Schwartz, Shai
Shannigrahi, Saswata
Shapira, Dana
Sharma, Ankit
Shioura, Akiyoshi
Silveira, Rodrigo
Singer, Yaron
Sinha, Amitabh
Sioutas, Spyros
Sitters, Rene
Sly, Allan
Smid, Michiel

Solomon, Shay
Son, Wanbin
Soto, Jose A.
Speckmann, Bettina
Spieksma, Frits
Srivastava, Piyush
Staals, Frank
Stefankovic, Daniel
Stein, Clifford
Storandt, Sabine
Strash, Darren
Strasser, Ben
Straszak, Damian
Streib, Noah
Sun, He
Sun, Xiaorui
Svensson, Ola
Swamy, Chaitanya
Tamir, Tami
Tarnawski, Jakub
Thankachan, Sharma V.
Thomopulos, Dimitri
Thorup, Mikkel
Tokuyama, Takeshi
Torng, Eric
Tripathi, Pushkar
Tsakalidis, Konstantinos
Tsichlas, Kostas
Tsur, Dekel
Tzamos, Christos
Uetz, Marc
Uno, Takeaki
Van Den Heuvel, Jan
Van Kreveld, Marc
van Leeuwen, Erik Jan
van Stee, Rob
van Zuylen, Anke
Vardi, Shai
Vassilevska Williams, Virginia
Venkatasubramanian, Suresh
Verbeek, Kevin
Verbitsky, Oleg
Verschae, José
Vigneron, Antoine
Viola, Emanuele
Viswanathan, Krishnamurthy
von Heymann, Frederik
Wagner, Uli
Wahlström, Magnus
Walen, Tomasz
Wang, Haitao
Wang, Yusu
Wang, Zhenbo
Weimann, Oren
Wenk, Carola
Werneck, Renato
Wieder, Udi
Wiedermann, Jiri
Wiese, Andreas
Wilkinson, Bryan T.
Wollan, Paul
Woods, Damien
Xie, Ning
Yamashita, Masafumi
Yannakakis, Mihalis
Yasuda, Norihito
Yasui, Yuichiro
Yeo, Anders
Yin, Yitong
Young, Neal
Yun, Se-Young
Zambelli, Giacomo
Zehendner, Elisabeth
Zhang, Qin
Zheng, Baigong
Zhou, Yuan
Zhu, Pingan
Ziv-Ukelson, Michal
Zwick, Uri
Zych, Anna

Table of Contents

Losing Weight by Gaining Edges 1
Amir Abboud, Kevin Lewi, and Ryan Williams

Optimal Coordination Mechanisms for Multi-job Scheduling Games 13
Fidaa Abed, José R. Correa, and Chien-Chung Huang

Theory and Practice of Chunked Sequences 25
Umut A. Acar, Arthur Charguéraud, and Mike Rainey

Convex Hulls under Uncertainty 37
Pankaj K. Agarwal, Sariel Har-Peled, Subhash Suri, Hakan Yıldız, and Wuzhou Zhang

The Space-Stretch-Time Tradeoff in Distance Oracles 49
Rachit Agarwal

Distribution-Sensitive Construction of the Greedy Spanner 61
Sander P.A. Alewijnse, Quirijn W. Bouts, and Alex P. ten Brink

Recognizing Shrinkable Complexes is NP-Complete 74
Dominique Attali, Olivier Devillers, Marc Glisse, and Sylvain Lazard

Improved Approximation Algorithms for Box Contact Representations 87
Michael A. Bekos, Thomas C. van Dijk, Martin Fink, Philipp Kindermann, Stephen Kobourov, Sergey Pupyrev, Joachim Spoerhase, and Alexander Wolff

Minimum Partial-Matching and Hausdorff RMS-Distance under Translation: Combinatorics and Algorithms 100
Rinat Ben-Avraham, Matthias Henze, Rafel Jaume, Balázs Keszegh, Orit E. Raz, Micha Sharir, and Igor Tubis

The Batched Predecessor Problem in External Memory 112
Michael A. Bender, Martín Farach-Colton, Mayank Goswami, Dzejla Medjedovic, Pablo Montes, and Meng-Tsung Tsai

Polynomial Decompositions in Polynomial Time 125
Arnab Bhattacharyya

Fault-Tolerant Approximate Shortest-Path Trees 137
Davide Bilò, Luciano Gualà, Stefano Leucci, and Guido Proietti

Fast Witness Extraction Using a Decision Oracle 149
Andreas Björklund, Petteri Kaski, and Łukasz Kowalik

Complexity of Higher-Degree Orthogonal Graph Embedding in the Kandinsky Model 161
Thomas Bläsius, Guido Brückner, and Ignaz Rutter

A Subexponential Parameterized Algorithm for Proper Interval Completion 173
Ivan Bliznets, Fedor V. Fomin, Marcin Pilipczuk, and Michał Pilipczuk

Computing Persistent Homology with Various Coefficient Fields in a Single Pass 185
Jean-Daniel Boissonnat and Clément Maria

De-anonymization of Heterogeneous Random Graphs in Quasilinear Time 197
Karl Bringmann, Tobias Friedrich, and Anton Krohmer

Competitive Algorithms for Restricted Caching and Matroid Caching ... 209
Niv Buchbinder, Shahar Chen, and Joseph (Seffi) Naor

Improved Algorithms for Resource Allocation under Varying Capacity 222
Venkatesan T. Chakaravarthy, Anamitra Roy Choudhury, Shalmoli Gupta, Sambuddha Roy, and Yogish Sabharwal

Nearly Tight Approximability Results for Minimum Biclique Cover and Partition 235
Parinya Chalermsook, Sandy Heydrich, Eugenia Holm, and Andreas Karrenbauer

Succinct Indices for Path Minimum, with Applications to Path Reporting 247
Timothy M. Chan, Meng He, J. Ian Munro, and Gelin Zhou

Online Bipartite Matching with Decomposable Weights 260
Moses Charikar, Monika Henzinger, and Huy L. Nguyễn

A Faster Algorithm for Computing Straight Skeletons 272
Siu-Wing Cheng, Liam Mencel, and Antoine Vigneron

Optimal Time-Space Tradeoff for the 2D Convex-Hull Problem 284
Omar Darwish and Amr Elmasry

Cache-Oblivious Persistence 296
Pooya Davoodi, Jeremy T. Fineman, John Iacono, and Özgür Özkan

Lightweight Approximate Selection 309
Brian C. Dean, Rommel Jalasutram, and Chad Waters

Robust Distance Queries on Massive Networks 321
Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck

A Dynamic Data Structure for MSO Properties in Graphs with Bounded Tree-Depth 334
Zdeněk Dvořák, Martin Kupec, and Vojtěch Tůma

Large Independent Sets in Triangle-Free Planar Graphs 346
Zdeněk Dvořák and Matthias Mnich

GRASP. Extending Graph Separators for the Single-Source Shortest-Path Problem 358
Alexandros Efentakis and Dieter Pfoser

Switching Colouring of *G(n,d/n)* for Sampling up to Gibbs Uniqueness Threshold 371
Charilaos Efthymiou

From Graph to Hypergraph Multiway Partition: Is the Single Threshold the Only Route? 382
Alina Ene and Huy L. Nguyễn

Deterministic Stateless Centralized Local Algorithms for Bounded Degree Graphs 394
Guy Even, Moti Medina, and Dana Ron

Bicriteria Data Compression: Efficient and Usable 406
Andrea Farruggia, Paolo Ferragina, and Rossano Venturini

Amortized $\tilde{O}(|V|)$-Delay Algorithm for Listing Chordless Cycles in Undirected Graphs 418
Rui Ferreira, Roberto Grossi, Romeo Rizzi, Gustavo Sacomoto, and Marie-France Sagot

LP Approaches to Improved Approximation for Clique Transversal in Perfect Graphs 430
Samuel Fiorini, R. Krithika, N.S. Narayanaswamy, and Venkatesh Raman

Representative Sets of Product Families 443
Fedor V. Fomin, Daniel Lokshtanov, Fahad Panolan, and Saket Saurabh

Weighted Ancestors in Suffix Trees 455
Paweł Gawrychowski, Moshe Lewenstein, and Patrick K. Nicholson

Improved Practical Matrix Sketching with Guarantees 467
Mina Ghashami, Amey Desai, and Jeff M. Phillips

Computing Regions Decomposable into m Stars 480
Matt Gibson, Kasturi Varadarajan, and Xiaodong Wu

The Parameterized Complexity of Graph Cyclability 492
Petr A. Golovach, Marcin Kamiński, Spyridon Maniatis, and Dimitrios M. Thilikos

Dimension Reduction via Colour Refinement 505
Martin Grohe, Kristian Kersting, Martin Mladenov, and Erkal Selman

How Experts Can Solve LPs Online 517
Anupam Gupta and Marco Molinaro

Parameterized Complexity of the k-Arc Chinese Postman Problem 530
Gregory Gutin, Mark Jones, and Bin Sheng

Approximating the Maximum Overlap of Polygons under Translation ... 542
Sariel Har-Peled and Subhro Roy

Ordering without Forbidden Patterns 554
Pavol Hell, Bojan Mohar, and Arash Rafiey

Halving Balls in Deterministic Linear Time 566
Michael Hoffmann, Vincent Kusters, and Tillmann Miltzow

Turing Kernelization for Finding Long Paths and Cycles in Restricted Graph Classes 579
Bart M.P. Jansen

Optimal Parallel Quantum Query Algorithms 592
Stacey Jeffery, Frederic Magniez, and Ronald de Wolf

Sublinear Space Algorithms for the Longest Common Substring Problem 605
Tomasz Kociumaka, Tatiana Starikovskaya, and Hjalte Wedel Vildhøj

Nested Set Union 618
Daniel H. Larkin and Robert E. Tarjan

Improved Explicit Data Structures in the Bitprobe Model 630
Moshe Lewenstein, J. Ian Munro, Patrick K. Nicholson, and Venkatesh Raman

Deeper Local Search for Better Approximation on Maximum Internal Spanning Trees 642
Wenjun Li, Jianer Chen, and Jianxin Wang

FPTAS for Counting Weighted Edge Covers 654
Jingcheng Liu, Pinyan Lu, and Chihao Zhang

Solving MULTICUT Faster than 2^n 666
Daniel Lokshtanov, Saket Saurabh, and Ondřej Suchý

Tight Bounds for Active Self-assembly Using an Insertion Primitive 677
Caleb Malchik and Andrew Winslow

Trace Reconstruction Revisited 689
Andrew McGregor, Eric Price, and Sofya Vorotnikova

PReaCH: A Fast Lightweight Reachability Index Using Pruning and Contraction Hierarchies 701
Florian Merz and Peter Sanders

Polynomial-Time Approximation Schemes for Circle Packing Problems 713
Flávio K. Miyazawa, Lehilton L.C. Pedrosa, Rafael C.S. Schouery, Maxim Sviridenko, and Yoshiko Wakabayashi

Document Retrieval on Repetitive Collections 725
Gonzalo Navarro, Simon J. Puglisi, and Jouni Sirén

An Improved Analysis of the Mömke-Svensson Algorithm for Graph-TSP on Subquartic Graphs 737
Alantha Newman

The Input/Output Complexity of Sparse Matrix Multiplication 750
Rasmus Pagh and Morten Stöckel

Faster FPTASes for Counting and Random Generation of Knapsack Solutions 762
Romeo Rizzi and Alexandru I. Tomescu

Improved Guarantees for Tree Cut Sparsifiers 774
Harald Räcke and Chintan Shah

Representative Families: A Unified Tradeoff-Based Approach 786
Hadas Shachnai and Meirav Zehavi

A Branch and Price Procedure for the Container Premarshalling Problem 798
Martijn van Brink and Ruben van der Zwaan

Space-Efficient Randomized Algorithms for K-SUM 810
Joshua R. Wang

Equivalence between Priority Queues and Sorting in External Memory 830
Zhewei Wei and Ke Yi

Amortized Bounds for Dynamic Orthogonal Range Reporting 842
Bryan T. Wilkinson

Author Index 857

Losing Weight by Gaining Edges

Amir Abboud, Kevin Lewi, and Ryan Williams

Computer Science Department, Stanford University, Stanford, CA, USA

Abstract. We present a new way to encode weighted sums into unweighted pairwise constraints, obtaining the following results.

- Define the k-SUM problem to be: given n integers in $[-n^{2k}, n^{2k}]$ are there k which sum to zero? (It is well known that the same problem over *arbitrary* integers is equivalent to the above definition, by linear-time randomized reductions.) We prove that this definition of k-SUM remains W[1]-hard, and is in fact W[1]-*complete*: k-SUM can be reduced to $f(k) \cdot n^{o(1)}$ instances of k-Clique.
- The maximum node-weighted k-Clique and node-weighted k-dominating set problems can be reduced to $n^{o(1)}$ instances of the *unweighted* k-Clique and k-dominating set problems, respectively. This implies a strong *equivalence* between the time complexities of the node weighted problems and the unweighted problems: any polynomial improvement on one would imply an improvement for the other.
- A triangle of weight 0 in a node weighted graph with m edges can be deterministically found in $m^{1.41}$ time.

1 Introduction

One of the most basic problems over integers, studied in geometry, cryptography, and combinatorics, is k-SUM, the parameterized version of the classical NP-complete problem SUBSET-SUM.

Definition 1.1 (k-SUM). *The* (k, M)-`SUM` *problem is to determine, given n integers $x_1, \ldots, x_n \in [0, M]$ and a target integer $t \in [0, M]$, if there exists a subset $S \subseteq [n]$ of size $|S| = k$ such that $\sum_{i \in S} x_i = t$.* [1] *We define k-SUM $\triangleq$* (k, n^{2k})-`SUM`.

Our definition of k-SUM is justified via the following known proposition:

Proposition 1.2. *Every instance S of* (k, M)-`SUM` *can be randomly reduced in $O(kn \log M)$ time to an instance S' of k-SUM as defined above.*

That is, there is an efficient randomized reduction from k-SUM over arbitrary integers, which we call k-SUM-$\mathbb{Z}$, to our definition of k-SUM. Furthermore, we

[1] Without loss of generality, the range of integers can be $[-M, M]$ and the target integer can be zero.

show in the full version that this reduction can be made deterministic under standard hardness assumptions.

A classical meet-in-the-middle algorithm solves k-SUM in $\tilde{O}(n^{\lceil k/2 \rceil})$ time and it has been a longstanding open problem to obtain an $O(n^{\lceil k/2 \rceil - \varepsilon})$ algorithm for any integer $k \geq 3$ and constant $\varepsilon > 0$. Logarithmic improvements are known for the $k = 3$ case [5,18] (that is, the famous 3-SUM problem). The k-SUM conjecture [25,1] states that k-SUM requires $n^{\lceil k/2 \rceil - o(1)}$ time and is known to imply tight lower bounds for many problems in computational geometry [17,19,6] (and many more) and has recently been used to show conditional lower bounds for discrete problems as well [25,29,21,2]. A matching $\Omega(n^{\lceil k/2 \rceil})$ lower bound for k-SUM was shown for a restricted model of computation called k-linear decision trees (LDTs) [15,3], although it was recently shown that depth $O(n^{k/2}\sqrt{\log n})$ suffices for $(2k-2)$-LDTs [18]. It is also known that if there is an unbounded function $s : \mathbb{N} \to \mathbb{N}$ such that for infinitely many k, k-SUM is in $n^{k/s(k)}$, then the Exponential Time Hypothesis is false [26].

Despite intensive research on this simple problem, our understanding is still lacking in many ways, one of which is from the viewpoint of parameterized complexity. In their seminal work on parameterized intractability, Downey and Fellows [12,13] proved that k-SUM-$\mathbb{Z}$ is $\mathsf{W}[1]$-hard and is contained in $\mathsf{W}[P]$. The even simpler Perfect Code problem was conjectured to lie between the classes $\mathsf{W}[1]$ and $\mathsf{W}[2]$ [13] until Cesati proved it was $\mathsf{W}[1]$-complete in 2002 [10]. Classifying k-SUM-$\mathbb{Z}$ within a finite level of the W-hierarchy was open until in 2007, when Buss and Islam [8] proved that k-SUM-$\mathbb{Z} \in \mathsf{W}[3]$.

The primary contribution of this work is a novel and generic way to efficiently convert problems concerning sums of numbers into problems on unweighted pairwise constraints. We call this technique "Losing weight by gaining edges" and report several interesting applications of it. One application is a new parameterized reduction from k-SUM to k-Clique and therefore the resolution of the parameterized complexity of k-SUM (for numbers in $[-n^{2k}, n^{2k}]$). Under standard lower bound hypotheses, we also obtain a deterministic reduction from k-SUM-$\mathbb{Z}$ to k-Clique as well.

Theorem 1.3. *k-SUM is $\mathsf{W}[1]$-complete.*

The significance of showing $\mathsf{W}[1]$-hardness for a problem is well known (as it rules out FPT algorithms). The significance of showing that a problem is in $\mathsf{W}[1]$ is less obvious, so let us provide some motivation. First, although $\mathsf{W}[1]$-complete problems are probably not FPT, prominent problems in $\mathsf{W}[1]$ (such as k-Clique) can still be solved substantially faster than exhaustive search over all $\binom{n}{k}$ subsets [23]. In contrast, analogous problems in $\mathsf{W}[2]$ (such as k-Dominating Set) do not have such algorithms unless CNF Satisfiability is in $2^{\delta n}$ time for some $\delta < 1$ [26], which is a major open problem in exact algorithms. Therefore, understanding which parameterized problems lie in $\mathsf{W}[1]$ is closely related to understanding which problems can be solved faster than exhaustive search. Second, showing that a problem is in $\mathsf{W}[1]$ rather than $\mathsf{W}[3]$ means that it can be expressed in an apparently weaker logic than before, with fewer quantifiers [16].

That is, putting a problem in $\mathsf{W}[1]$ decreases the descriptive complexity of the problem.

Theorem 1.3 has applications to parameterized complexity, yielding a new characterization of the class $\mathsf{W}[1]$ as the problems FPT-reducible to k-SUM. Since k-SUM is quite different in nature from the previously known $\mathsf{W}[1]$-complete problems, we are able to put other such "intermediate" problems in $\mathsf{W}[1]$, including weighted graph problems and problems with application to coding theory such as Weight Distribution [9] (for which the details are given in the full version).

To show that k-SUM $\in \mathsf{W}[1]$, we prove a very tight reduction from k-SUM to k-Clique. Given an instance of k-SUM on n numbers, we generate $f(k) \cdot n^{o(1)}$ instances of k-Clique on n node graphs, such that one of these graphs contains a k-clique if and only if our k-SUM instance has a solution. This implies that any algorithm for k-Clique running in time $O(n^c)$ for some $c \geq 2$ yields an algorithm for k-SUM running in time $n^{c+o(1)}$. Hence, the k-SUM conjecture implies an $n^{\lceil k/2 \rceil - o(1)}$ lower bound for k-Clique as well.

Generalizing our ideas further, we are able to prove surprising consequences regarding other weighted problems.

Removing Node Weights. Two fundamental graph problems are k-Clique and k-Dominating Set. Natural extensions of these problems allow the input graph to have weights on its nodes. The problem can then be to find a k-clique or a k-dominating set of minimum or maximum sum of node weights (the *min* and *max* versions), or to find a k-clique or a k-dominating set with total weight exactly 0 (the *exact* version, defined below).

Definition 1.4 (The Node-Weight k-Clique-Sum Problem). *For integers $k, M > 0$, the (k, M)-*`NW-CLIQUE` *problem is to determine, given a graph G, a node-weight function $w : V(G) \to [0, M]$, and a target weight $t \in [0, M]$, if there is a set S of k nodes which form a clique such that $\sum_{v \in S} w(v) = t$. We define the Node-Weight k-Clique-Sum problem as (k, n^{2k})-*`NW-CLIQUE`*.*

Definition 1.5 (Node-Weight k-Dominating-Set-Sum). *For an integer $k > 0$, the Node-Weight k-Dominating-Set-Sum problem is to determine, given a graph G, a node-weight function $w : V(G) \to [0, n^{2k}]$, and a target weight $t \in [0, n^{2k}]$, if there is a set S of k nodes which form a dominating set such that $\sum_{v \in S} w(v) = t$.*

These additional node weights increase the expressibility of the problem and allow us to capture more applications. How much harder are these node weighted versions compared to the unweighted versions? By weight scaling arguments, one can show that the "exact" version is harder than the max and min versions, in the sense that any algorithm for "exact" implies an algorithm for max or min with only a logarithmic overhead [22] (and Theorem 3.3 in [29]). But how much harder is (for example) Node-Weight k-Clique-Sum than the case where there are no weights at all?

For k divisible by 3, the best k-Clique algorithms reduce the problem to 3-Clique on $n^{k/3}$ nodes, then use an $O(n^\omega)$ time algorithm for triangle detection [20] for a running time of $O(n^{\omega k/3})$ [23]. This reduction to the $k =$

3 case works for the node weighted case as well; combined with the recent $n^{\omega+o(1)}$ algorithms for node weighted triangle (3-clique) problems [11,29], we obtain $n^{\omega k/3+o(1)}$ running times for node weighted k-clique problems. The best k-Dominating Set algorithms reduce the problem to a rectangular matrix multiplication of matrices of dimensions $n^{k/2} \times n$ and $n \times n^{k/2}$ and run in time $n^{k+o(1)}$ [14]. These algorithms allow us to find all k-dominating sets in the graph and therefore can also solve the node weighted versions without extra cost.

Therefore, the state of the art algorithms for k-Clique and k-Dominating Set suggest that adding node weights does not make the problems much harder. Is that due to our current algorithms for the unweighted problems, or is there a deeper connection? Using the "Losing weight by gaining edges" ideas, we show that the node weighted versions of k-Clique and k-Dominating Set (and, in fact, *any* problem that allows us to implement certain "pairwise constraints") are essentially "equivalent" to the unweighted versions.

Theorem 1.6. *If k-Clique on n node and m edge graphs can be solved in time $T(n, m, k)$, then Node-Weight k-Clique-Sum on n node and m edge graphs can be solved in time $n^{o(1)} \cdot T(kn, k^2m, k)$. If k-Dominating Set on n node graphs can be solved in time $T(n, k)$, then Node-Weight k-Dominating-Set on n node graphs can be solved in time $n^{o(1)} \cdot T(k^2n, k)$.*

Interestingly, Theorem 1.6 yields a short and simple $n^{\omega+o(1)}$ algorithm for the node-weighted triangle problems, while a series of papers were required to recently conclude the same upper bound using different techniques [28,30,11,29]. Moreover, unlike the previous techniques, our approach extends to $k > 3$ and applies to more problems like k-Dominating Set.

Applying Theorem 1.6 to the $O(m^{\frac{2\omega}{\omega+1}})$ triangle detection algorithm of Alon, Yuster and Zwick [4], we obtain a *deterministic* algorithm for Node-Weight Triangle-Sum in sparse graphs, improving the previous $n^{\omega+o(1)}$ upper bound [29] and matching the running time of the best randomized algorithm [29].

Corollary 1.7. *Node-Weight Triangle-Sum can be solved deterministically in $m^{1.41+o(1)}$ time.*

1.1 Overview of the Proofs

Let us give some intuition for Theorem 1.3. Both the containment in W[1] and the hardness for W[1] require new technical ideas. Downey and Fellows [12,13] proved that k-SUM-$\mathbb{Z}$ is W[1]-hard by a reduction requiring fairly large numbers: they are exponential in n, but can still be generated in an FPT way. To prove that k-SUM is W[1]-hard even when the numbers are only exponential in $k \log n$, we need a much more efficient encoding of k-Clique instances. We apply some machinery from additive combinatorics, namely a construction of large sets of integers avoiding trivial solutions to the linear equation $\sum_{i=1}^{k-1} x_i = (k-1)x_k$ [24]. These sets allow us to efficiently "pack" a k-Clique instance into a $(\binom{k}{2}+k)$-SUM instance on small numbers.

Proving that k-SUM is in W[1] takes several technical steps. We provide a parameterized reduction from k-SUM on n numbers to only $f(k) \cdot n^{o(1)}$ graphs on $O(kn)$ nodes, such that some graph has a k-clique if and only if the original n numbers have a k-SUM. To efficiently reduce from numbers to graphs, we first reduce the numbers to an analogous problem on vectors. We define an intermediate problem (k, M)-`VECTOR-SUM`, in which one is given a list of n vectors from $\{-kM, \ldots, 0, \ldots, kM\}^d$, and is asked to determine if there are k vectors which sum to the all-zero vector. We give an FPT reduction from k-SUM to (k, M)-`VECTOR-SUM` where M and d are "small" (such that M^d is approximately equal to the original weights of the k-SUM instance). Next, we "push" the weights in these vectors onto the edges of a graph connecting the vectors, where the edge weights are much smaller than the original numbers: we reduce from (k, M)-`VECTOR-SUM` to edge-weighted k-clique-sum using a polynomial "squaring trick" which creates a graph with "small" edge weights, closely related in size to M. Finally, we reduce from the weighted problem to the unweighted version of the problem by brute-forcing all feasible weight combinations on the edges; as the edge weights are small, this creates $f(k) \cdot n^{o(1)}$ unweighted k-Clique instances for some function f.

Combining all these steps into one, one can view our approach as follows. We enumerate over all $\binom{k}{2}$-tuples of numbers $t = (\alpha_{i,j})_{i,j \in [k]}$ such that $\sum_{i,j} \alpha_{i,j} = 0$ where $\alpha_{i,j} \in [-M, M]$ for $M = f(k) \cdot \operatorname{poly}\log n$, and for each such tuple t we generate an instance of the unweighted problem. In this instance, two nodes are allowed to both be a part of our final solution (e.g. there is an edge between them in the k-clique case) if and only if some expression on the weights of the objects v_i and v_j evaluates to $F(w(v_i), w(v_j)) = \alpha_{i,j}$. The formulas are defined, via the "squaring trick", in such a way that there are k nodes satisfying these $\binom{k}{2}$ equations for some $\binom{k}{2}$-tuple t if and only if the sum of the weights of these k nodes is 0.

To implement our approach for k-Dominating Set we follow similar steps, except that we cannot implement the constraints on having a certain pair of objects in our solution by removing the edge between them anymore, since this does not prevent them from being in a feasible k-dominating set. This can be done, however, by adding extra nodes X to the graph such that the inclusion of pairs of nodes v_i, v_j in the solution S that do not satisfy our equations, $F(w(v_i), w(v_j)) \neq \alpha_{i,j}$, will prevent S from dominating all the nodes in X.

1.2 Related Work

There has been recent work in relating the complexity of k-SUM and variations of k-Clique for the specific case of $k = 3$. Pătraşcu [25] shows a tight reduction from 3-SUM to listing 3-cliques; a reduction from listing 3-cliques to 3-SUM is given by Jafargholi and Viola [21]. Vassilevska and Williams [29] consider the exact edge-weight 3-clique problem and give a tight reduction from 3-SUM. For the case of $k > 3$, less is known, as the techniques used for the case of $k = 3$ do not seem to generalize easily. Abboud and Lewi [1] give reductions between

k-SUM and various exact edge-weighted graph problems where the goal is to find an instance of a specific subgraph whose edge weights sum to 0.

2 Preliminaries

For $i < j \in \mathbb{Z}$, define $[i, j] \triangleq \{i, \ldots, j\}$. As shorthand, we define $[n] \triangleq [1, n]$. For a vector $\mathbf{v} \in \mathbb{Z}^d$, we denote by $\mathbf{v}[j]$ the value in the j^{th} coordinate of $\mathbf{v}$. We let $\mathbf{0}$ denote the all zeros vector. The default domain and range of a function is $\mathbb{N}$.

We define the k-Clique problem as follows.

Definition 2.1 (The *k*-Clique Problem). *For integers $k, n, m > 0$, the k-clique problem is to determine, given a graph G, if there is a size-k subset $S \subseteq [n]$ such that S is a clique in G.*

The following problems are referred to in Corollary 3.8. They are simply the unparameterized versions of k-SUM and Exact Edge-Weight k-Clique, respectively.

Definition 2.2 (The Subset-SUM Problem). *The Subset-SUM problem is to determine, given a set of integers $x_1, \ldots, x_n, t$, if there exists a subset $S \subseteq [n]$ such that $\sum_{i \in S} x_i = t$.*

Definition 2.3 (The Exact Edge-Weight Clique Problem). *For integers $n, m, M > 0$, the Exact Edge-Weight k-Clique problem is to determine, given an instance of a graph G on n vertices and m edges, a weight function $w : E(G) \to [-M, M]$, if there exists a set of nodes which form a clique with total weight 0.*

3 From Numbers to Edges

Our results begin by showing how to reduce k-SUM to k-Clique. To do this, we first give a new reduction from k-SUM to k-Vector-Sum on n vectors in C^d for a set C which is relatively small compared to the numbers in the original instance. From k-Vector-Sum, we give a reduction to Edge-Weight k-Clique-Sum with small weights. Then, we can brute-force all possibilities for the $\binom{k}{2}$ edge weights for k-SUM and reduce to the (unweighted) k-Clique problem. Altogether, we conclude that k-SUM is in $\mathsf{W}[1]$.

3.1 Reducing *k*-SUM to *k*-Vector-Sum

We present a generic way to map numbers into vectors over small numbers such that the k-sums are preserved. We define the k-Vector-Sum problem as follows.

Definition 3.1 (The *k*-Vector-Sum Problem). *For integers $k, n, M, d > 0$, the k-vector-sum problem (k, M)-`VECTOR-SUM` is to determine, given vectors $\mathbf{v}_1, \ldots, \mathbf{v}_n, \mathbf{t} \in [0, kM]^d$, if there is a size-$k$ subset $S \subseteq [n]$ such that $\sum_{i \in S} \mathbf{v}_i = \mathbf{t}$.*

Note that the problem was considered by Bhattacharyya et al. [7] and also by Cattaneo and Perdrix [9].

Lemma 3.2. *Let $k, p, d, s, M \in \mathbb{N}$ satisfy $k < p$, $p^d \geq kM + 1$, and $s = (k+1)^{d-1}$. There is a collection of mappings $f_1, \ldots, f_s : [0, M] \times [0, kM] \to [-kp, kp]^d$, each computable in time $O(\text{poly}\log M + k^d)$, such that for all numbers $x_1, \ldots, x_k \in [0, M]$ and targets $t \in [0, kM]$,*

$$\sum_{j=1}^{k} x_j = t \qquad \Leftrightarrow \qquad \exists\, i \in [s] \quad \textit{such that} \quad \sum_{j=1}^{k} f_i(x_j, t) = \mathbf{0}.$$

The idea is simple: in a natural translation of numbers into vectors, to preserve k-sums we have to keep track of the carries that may occur. These f_i's effectively try "all possible" carries there can be among a sum of k numbers. The proof is given in the full version.

Corollary 3.3. *Let $k, p, d, M, n > 0$ be integers with $k < p$ and $p^d \geq kM + 1$. k-SUM on n integers in the range $[0, M]$ can be reduced to $O(k^d)$ instances of $(k, p-1)$-*`VECTOR-SUM` *on n vectors in $[0, p-1]^d$.*

3.2 Reducing to k-Clique

Here, we consider a generalization of the k-SUM problem—namely, the Node-Weight k-Clique-Sum problem. We give a reduction from Node-Weight k-Clique-Sum to Edge-Weight k-Clique-Sum (defined below), where the new edge weights are much smaller than the original node weights. We then show how to reduce to many instances of the unweighted version of the problem, where each instance corresponds to a possible setting of edge weights. Then, we give an application of this general reduction to the Node-Weight k-Clique-Sum problem.

Definition 3.4 (The Edge-Weight k-Clique-Sum Problem). *For integers $k, M > 0$, the edge-weight k-clique-sum problem (k, M)-*`EW-CLIQUE` *is to determine, given a graph G, an edge-weight function $w : E(G) \to [0, M]$, and a target weight $t \in [0, M]$, if there is a set S of k nodes which form a clique such that $\sum_{(u,v) \in S} w(u, v) = t$.*

Lemma 3.5. *Let $k, p, d, M > 0$ be integers such that $k < p$ and $p^d \geq kM + 1$, and let $M' = O(k^3 d p^2)$. (k, M)-*`NW-CLIQUE` *can be deterministically reduced to $O(k^d)$ instances of (k, M')-*`EW-CLIQUE` *in time $O(k^d \cdot n^2 \cdot \text{poly}\log M)$.*

Thinking of $p + d$ as "small", but $\text{poly}(p, d) \approx kM$ as "large", we get a substantial reduction in the weights of the problem by "spreading" the node weights over the edges.

Proof. Let $G = (V, E)$ be a graph with a node weight function $w : V \to [0, M]$ and a target number $t \in [0, kM]$. Recall the mappings $f_i : [0, M] \times \times [0, kM] \to [-kp, kp]^d$ for $i \in [s]$ from Lemma 3.2, which maps numbers from $[0, M]$ into a collection of $s = O(k^d)$ length-d vectors with entries in $[-kp, kp]$. We translate the node-weight vector problem into an edge-weight problem via a "squaring trick," as follows. For each $i \in [s]$, we define an edge weight function $w_i : E \to [-M', M']$. For $(u, v) \in E$, let $\mathbf{u} = f_i(w(u), t)$ and $\mathbf{v} = f_i(w(v), t)$, and define

$$w_i(u, v) \triangleq \sum_{j=1}^{d} \left(\mathbf{u}[j]^2 + \mathbf{v}[j]^2 + 2(k-1)\mathbf{u}[j] \cdot \mathbf{v}[j]\right).$$

Note that for $M' = O(kdp^2)$, $w_i(u, v) \in [-M', M']$. We show that there is a k-clique in (G, w) of node-weight t if and only if for some $i \in [s]$, the edge-weighted graph (G, w_i) contains a k-clique of edge-weight 0. First, observe that for any k vectors $\mathbf{v}_1, \ldots, \mathbf{v}_k \in \mathbb{Z}^d$,

$$\sum_{i=1}^{k} \mathbf{v}_i = \mathbf{0} \iff \sum_{j=1}^{d} \left(\sum_{i=1}^{k} \mathbf{v}_i[j]\right)^2 = 0.$$

Consider a set $S = \{u_1, \ldots, u_k\} \subseteq V$ that forms a k-clique in G. For any $i \in [s]$ and $u_a, u_b \in S$, let $\mathbf{u}_a = f_i(w(u_a), t)$ and $\mathbf{u}_b = f_i(w(u_b), t)$. Then, the edge-weight of S in (G, w_i) is

$$\sum_{1 \le a < b \le k} w_i(u_a, u_b) = (k-1) \sum_{a=1}^{k} \sum_{j=1}^{d} \mathbf{u}_a[j]^2 + 2(k-1) \sum_{1 \le a < b \le k} \sum_{j=1}^{d} \mathbf{u}_a[j] \cdot \mathbf{u}_b[j].$$

Since the sum is evaluated over all pairs $a, b \in [k]$ where $a < b$, the above quantity is equal to

$$(k-1) \cdot \sum_{j=1}^{d} \left(\sum_{u \in S} f_i(w(u), t)[j]\right)^2.$$

Therefore, for all $i \in [s]$, the edge-weight of S in (G, w_i) equals 0 if and only if the sum of the vectors $\sum_{u \in S} f_i(w(u), t)$ equals $\mathbf{0}$. And, by the properties of the mappings f_i from Lemma 3.2, the latter occurs for some $i \in [s]$ if and only if the node-weight of S in (G, w), $\sum_{u \in S} w(u)$, is equal to t, as desired. ∎

We observe that in the graphs produced by the above reduction, all k-cliques have non-negative weight. Therefore, Lemma 3.5 can also be viewed as a reduction to the "minimum-weight" k-Clique problem with edge weights, where the edge sum is minimized.

Finally, small weights on edges can simply be eliminated using a brute-force step. The proof of the following lemma is given in the full version.

Lemma 3.6. *For all integers $k, M > 0$, there is an $O(M^{\binom{k}{2}} \cdot n^2)$ time reduction from the problem (k, M)-`EW-CLIQUE` to $O(M^{\binom{k}{2}})$ instances of k-Clique on n nodes and $m \cdot \binom{k}{2}$ edges.*

3.3 k-SUM is in $\mathsf{W}[1]$

Using the above lemmas, we can efficiently reduce k-SUM to k-Clique. Consider a k-SUM instance (S, t) where $S = \{x_1, \ldots, x_n\} \subseteq [0, M]$ and $t \in [0, kM]$ with $M = n^{2k}$. Let $G = (V, E)$ be a node-weighted clique on n nodes $V = \{v_1, \ldots, v_n\}$ with weight function $w : V \to [0, M]$ such that $w(v_i) = x_i$ for all $i \in [n]$. Clearly, (S, t) has a k-SUM solution if and only if the instance (G, w, t) of (k, M)-`NW-CLIQUE` has a solution.

Set $d = \lceil \log n / \log\log n \rceil$ and $p = \lceil \log^{4k} n \rceil$, so that $p^d \geq (n)^{4k} > kM$. Using Lemma 3.5 the instance (G, w, t) of (k, M)-`NW-CLIQUE` can be reduced to $O(k^d) = O(n^{\log k / \log\log n})$ instances of (k, M')-`EW-CLIQUE`, where $M' = O(k^3 \cdot \log^{8k+1} n / \log\log n)$. Then, using Lemma 3.6, we can generate $g(n, k) = O(n^{\frac{\log k}{\log\log n}} \cdot k^{3k^2} \log^{8k^2+k} n)$ graphs on n nodes and $O(n^2)$ edges such that some graph has a k-Clique if and only if the original k-SUM instance has a solution.

For constant k, note that $g(n, k) = n^{o(1)}$, and hence:

Theorem 3.7. *For any $c > 2$, if k-Clique can be solved in time $O(n^c)$, then k-SUM can be solved in time $n^{c+o(1)}$.*

Furthermore, we remark that by applying the above reduction from k-SUM to k-Clique to the respective *unparameterized* versions of these problems, we obtain a reduction from Subset-SUM on arbitrary weights to Exact Edge-Weight Clique with small edge weights.

Corollary 3.8. *For any $\varepsilon > 0$, Subset-SUM on n numbers in $[-2^{O(n)}, 2^{O(n)}]$ can be reduced to $2^{\varepsilon n}$ instances of Exact Edge-Weight Clique on n nodes with edge weights are in $[-n^{O(1/\varepsilon)}, n^{O(1/\varepsilon)}]$.*

Note that Subset-SUM on n numbers in $[-2^{O(n)}, 2^{O(n)}]$ is as hard as the general case of Subset-SUM, and the fastest known algorithm for Subset-SUM on n numbers runs in time $O(2^{n/2})$. The unweighted Max-Clique problem, which asks for the largest clique in a graph on n nodes, can be solved in time $O(2^{n/4})$ [27]. Corollary 3.8 shows that even when the edge weights are small, the edge-weighted version of Max-Clique requires time $\Omega(2^{n/2})$ unless Subset-SUM can be solved faster.

An FPT Reduction. We show how to make the reduction fixed-parameter tractable. We can modify the oracle reduction for k-Clique above to get a many-one reduction to k-Clique if we simply take the disjoint union of the $g(n, k)$ k-Clique instances as a single k-Clique instance. The resulting graph has $n \cdot g(n, k)$ nodes, $O(n^2 \cdot g(n, k))$ edges, and has a k-clique if and only if one of the original graphs has a k-Clique. Then, we make the following standard argument to appropriately bound $g(n, k)$ via case analysis. If $k < \lceil \log\log n \rceil$, then $g(n, k) \leq n^{o(1)} \cdot 2^{f(k) \cdot \mathrm{poly}(k)}$. If $k \geq \lceil \log\log n \rceil$, then since $n \leq 2^{2^k}$, we have that $g(n, k) \leq 2^{2^k + f(k) \cdot \mathrm{poly}(k)}$. Therefore, $g(n, k) \leq n^{o(1)} \cdot h(k)$ for some computable $h : \mathbb{N} \to \mathbb{N}$, and we have shown the following:

Lemma 3.9. *k-SUM is in* $\mathsf{W}[1]$.

In the full version, we show how to obtain a randomized FPT reduction from the k-SUM problem over the integers to k-Clique, and how under plausible circuit lower bound assumptions, we can derandomize this reduction to show that k-SUM over the integers is in $\mathsf{W}[1]$. This yields the first half of Theorem 1.3 (and we show the remainder, that k-SUM is $\mathsf{W}[1]$-hard, in the next section).

3.4 Node-Weight *k*-Clique-Sum

The reduction of Section 3.3 shows that the Node-Weight k-Clique-Sum problem can be reduced to $n^{o(1)}$ instances of k-Clique, when k is a fixed constant. We observe that if the input graph has m edges, then the graphs generated by the reduction have no more than $k^2 m$ edges. Therefore, we have a tight reduction from node-weight clique to k-Clique.

This concludes the proof of the first half of Theorem 1.6 referencing k-Clique. We defer the proof of the second half of Theorem 1.6 concerning k-Dominating Set to the full version.

4 From *k*-Clique to *k*-SUM

In this section, we give a new reduction from k-clique to k-SUM in which the numbers generated are all in the interval $[-n^{2k}, n^{2k}]$. This proves that k-SUM is in fact $\mathsf{W}[1]$-hard. We can view the result as an alternate proof for the $\mathsf{W}[1]$-hardness of k-SUM without use of the Perfect Code problem, as done by Downey and Fellows [12]. The reduction is given from k-Clique to k-Vector-Sum (recall Definition 3.1), and then from k-Vector-Sum to k-SUM.

The proof the following lemma is given in the full version.

Lemma 4.1. *For an integer $k > 1$, k-Clique on n nodes and m edges reduces to an instance of $(k + \binom{k}{2}, k \cdot n^{1+o(1)})$-*`VECTOR-SUM` *deterministically in time $O(n^2)$.*

The following lemma gives a simple reduction from k-Vector-Sum to k-SUM, by the usual trick of converting from vectors to integers (via a Freiman isomorphism of order k). We give the proof in the full version.

Lemma 4.2. (k, M)-`VECTOR-SUM` *can be reduced to k-SUM on n integers in the range $[0, (kM + 1)^d]$ in $O(n \log M)$ time.*

We remark that in some cases, the proof can be changed slightly to yield smaller numbers in the k-SUM instance produced by the reduction. In particular, when reducing k-Clique to k-Vector-Sum, only the numbers in the first k coordinates can be as large as $k \cdot n^{1+o(1)}$ while the numbers in the last $k^2 + 1$ coordinates are bounded by k, and therefore, when reducing to $(k + \binom{k}{2}, M)$-`SUM` on $kn + \binom{k}{2}m$ numbers, the numbers generated can be bounded by $M = k^d \cdot (kn^{1+o(1)})^k \cdot k^{k^2+1} = O(k^{2k^2} \cdot n^{k+o(k)})$. In other words, we have reduced k-Clique to k'-SUM with numbers in the range $\left[-n^{\sqrt{k'}}, n^{\sqrt{k'}}\right]$, where $k' = k + \binom{k}{2}$.

The composition of Lemma 4.1 and Lemma 4.2 yields an FPT reduction, and we have obtained:

Lemma 4.3. *k-SUM is* W[1]*-hard.*

This concludes the proof of Theorem 1.3.

Acknowledgements. We would like to thank the anonymous reviewers for their helpful comments. This work was supported in part by a David Morgenthaler II Faculty Fellowship, and NSF CCF-1212372.

References

1. Abboud, A., Lewi, K.: Exact Weight Subgraphs and the k-Sum Conjecture. In: Fomin, F.V., Freivalds, R., Kwiatkowska, M., Peleg, D. (eds.) ICALP 2013, Part I. LNCS, vol. 7965, pp. 1–12. Springer, Heidelberg (2013)
2. Abboud, A., Williams, V.V.: Popular conjectures imply strong lower bounds for dynamic problems. CoRR, abs/1402.0054 (2014)
3. Ailon, N., Chazelle, B.: Lower bounds for linear degeneracy testing. J. ACM 52(2), 157–171 (2005)
4. Alon, N., Yuster, R., Zwick, U.: Finding and counting given length cycles. Algorithmica 17(3), 209–223 (1997)
5. Baran, I., Demaine, E.D., Pătraşcu, M.: Subquadratic algorithms for 3SUM. Algorithmica 50(4), 584–596 (2008)
6. Barequet, G., Har-Peled, S.: Polygon Containment and Translational Min-Hausdorff-Distance Between Segment Sets are 3SUM-Hard. Int. J. Comput. Geometry Appl. 11(4), 465–474 (2001)
7. Bhattacharyya, A., Indyk, P., Woodruff, D.P., Xie, N.: The complexity of linear dependence problems in vector spaces. In: ICS, pp. 496–508 (2011)
8. Jonathan, F.: Buss and Tarique Islam. Algorithms in the W-Hierarchy. Theory Comput. Syst. 41(3), 445–457 (2007)
9. Cattanéo, D., Perdrix, S.: The parameterized complexity of domination-type problems and application to linear codes. CoRR, abs/1209.5267 (2012)
10. Cesati, M.: Perfect Code is W[1]-complete. Inf. Process. Lett. 81(3), 163–168 (2002)
11. Czumaj, A., Lingas, A.: Finding a heaviest vertex-weighted triangle is not harder than matrix multiplication. SIAM J. Comput. 39(2), 431–444 (2009)
12. Downey, R.G., Fellows, M.R.: Fixed-parameter intractability. In: Structure in Complexity Theory Conference, pp. 36–49 (1992)
13. Downey, R.G., Fellows, M.R.: Fixed-parameter tractability and completeness II: On completeness for W[1]. Theor. Comput. Sci. 141(1&2), 109–131 (1995)
14. Eisenbrand, F., Grandoni, F.: On the complexity of fixed parameter clique and dominating set. Theor. Comput. Sci. 326(1-3), 57–67 (2004)
15. Erickson, J.: Lower bounds for linear satisfiability problems. In: SODA, pp. 388–395 (1995)
16. Flum, J., Grohe, M.: Parameterized Complexity Theory. Springer (2006)
17. Gajentaan, A., Overmars, M.H.: On a class of $O(n^2)$ problems in computational geometry. Computational Geometry 5(3), 165–185 (1995)
18. Grønlund, A., Pettie, S.: Threesomes, Degenerates, and Love Triangles. CoRR, abs/1404.0799 (2014)

19. Hernández-Barrera, A.: Finding an O(n^2logn) Algorithm Is Sometimes Hard. In: CCCG, pp. 289–294 (1996)
20. Itai, A., Rodeh, M.: Finding a minimum circuit in a graph. In: STOC 1977, pp. 1–10. ACM, New York (1977)
21. Jafargholi, Z., Viola, E.: 3SUM, 3XOR, Triangles. CoRR, abs/1305.3827 (2013)
22. Nederlof, J., van Leeuwen, E.J., van der Zwaan, R.: Reducing a target interval to a few exact queries. In: MFCS, pp. 718–727 (2012)
23. Nešetřil, J., Poljak, S.: On the complexity of the subgraph problem. Commentationes Mathematicae Universitatis Carolinae 26(2), 415–419 (1985)
24. O'Bryant, K.: Sets of integers that do not contain long arithmetic progressions. Electr. J. Comb. 18(1) (2011)
25. Patrascu, M.: Towards polynomial lower bounds for dynamic problems. In: STOC, pp. 603–610 (2010)
26. Pătraşcu, M., Williams, R.: On the Possibility of Faster SAT Algorithms. In: SODA, pp. 1065–1075 (2010)
27. Robson, J.M.: Finding a maximum independent set in time O($2^{n/4}$). Technical report, 1251-01, LaBRI, Université de Bordeaux I (2001)
28. Vassilevska, V., Williams, R.: Finding a maximum weight triangle in $n^{3-\mathrm{delta}}$ time, with applications. In: STOC, pp. 225–231 (2006)
29. Vassilevska, V., Williams, R.: Finding, minimizing, and counting weighted subgraphs. In: STOC, pp. 455–464 (2009)
30. Vassilevska, V., Williams, R., Yuster, R.: Finding the smallest *H*-subgraph in real weighted graphs and related problems. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4051, pp. 262–273. Springer, Heidelberg (2006)

Optimal Coordination Mechanisms for Multi-job Scheduling Games[⋆]

Fidaa Abed[1], José R. Correa[2], and Chien-Chung Huang[3]

[1] Max-Planck-Institut für Informatik
fabed@mpi-inf.mpg.de
[2] Departamento de Ingenieria Industrial, Universidad de Chile
correa@uchile.cl
[3] Chalmers University of Technology
huangch@chalmers.se

Abstract. We consider the unrelated machine scheduling game in which players control subsets of jobs. Each player's objective is to minimize the weighted sum of completion time of her jobs, while the social cost is the sum of players' costs. The goal is to design simple processing policies in the machines with small coordination ratio, i.e., the implied equilibria are within a small factor of the optimal schedule. We work with a weaker equilibrium concept that includes that of Nash. We first prove that if machines order jobs according to their processing time to weight ratio, a.k.a. Smith-rule, then the coordination ratio is at most 4, moreover this is best possible among nonpreemptive policies. Then we establish our main result. We design a preemptive policy, *externality*, that extends Smith-rule by adding extra delays on the jobs accounting for the negative externality they impose on other players. For this policy we prove that the coordination ratio is $1 + \phi \approx 2.618$, and complement this result by proving that this ratio is best possible even if we allow for randomization or full information. Finally, we establish that this externality policy induces a potential game and that an ε-equilibrium can be found in polynomial time. An interesting consequence of our results is that an $\varepsilon-$local optima of $R| \, | \sum w_j C_j$ for the jump (a.k.a. move) neighborhood can be found in polynomial time and are within a factor of 2.618 of the optimal solution. The latter constitutes the first direct application of purely game-theoretic ideas to the analysis of a well studied local search heuristic.

1 Introduction

Machine scheduling originates in the optimization of manufacturing systems and their formal mathematical treatment dates back to at least the pioneering work of Smith [38]. In general, scheduling problems can be described as follows. Consider a set $\mathcal{J}$ of n jobs that have to be processed on a set $\mathcal{M}$ of m parallel machines. If processed on machine i, job j requires a certain processing time p_{ij} to be

[⋆] Research partially supported by the Millenium Nucleus Information and Coordination in Networks ICM/FIC P10-024F.

completed. Job j also has a weight w_j and, in addition, it may have other characteristics such as release dates, time windows, delays when switching a task from one machine to another, or precedence constraints. The goal is to find an assignment of jobs to machines, and an ordering within each machine so that a certain objective functions is minimized. Denoting, for any such assignment and ordering, the *completion time* C_j of job j the time at which job j completes, we may write the two most widely studied objectives as $C_{\max} = \max_{j \in \mathcal{J}} C_j$ (the makespan) and $\sum_{j \in \mathcal{J}} w_j C_j$ (the sum of weighted completion times). In terms of the machine environment the most basic model is that of *identical* machines, where the processing times of jobs are the same on all machines. In the *related* machines environment each machine has a speed, and the processing time of a job on a machine is inversely proportional to the speed of that machine. Finally in *unrelated* machine scheduling the processing times are arbitrary, thus capturing all the above models as special cases. This latter machine environment, with the sum of weighted completion times objective, denoted by $R| \,| \sum w_j C_j$, is the focus of our paper.

Since the early work of Smith for the $\sum w_j C_j$ objective, a lot of work has been put in designing *centralized* algorithms providing reasonably close to optimal solutions with limited computational effort for these NP-hard problems [5,15,18,21,23,24,32,33,34,35,36,37]. The underlying assumption is that all information is gathered by a single entity which can enforce a particular schedule. However, as distributed environments emerge, understanding scheduling problems where jobs are managed by different selfish agents (players), who are interested in their own completion time, becomes a central question.

Coordination Mechanisms. In recent times there has been quite some effort to understand these scheduling games in the special case in which agents control a single job in the system, which we call *single-job games*. In this context, there is a vast amount of work studying existence, uniqueness, the *price of anarchy* [26], and other characteristics of equilibrium when, given some processing rules, each agent seeks to minimize her own completion time. In the scheduling game each job is a fully informed player wanting to minimize its individual completion time, and its set of strategies correspond to the set of machines. Job j's completion time on a machine depends on the strategies chosen by other players, and on the *policy* (or processing rule) of the chosen machine. While the cost of a job is its weighted completion time, $w_j C_j$. A *coordination mechanism* is then a set of *local policies*, one per machine, specifying how the jobs assigned to that machine are scheduled. In a *local policy* the schedule on a machine depends on the full vector $(p_{1j}, p_{2j}, \ldots, p_{mj})$ and weights w_j of jobs assigned to that machine. In contrast, in a *strongly local policy* the schedule on machine i must be a function only of the processing times p_{ij} and weights w_j of the jobs assigned to i. In evaluating the efficiency of these policies, one needs a benchmark to compare this social cost against. The definition of the price of anarchy of the induced game considers a social optimum with respect to the costs specified by the chosen machine policies. However, to measure the quality of a coordination mechanism we consider the worst case ratio of the social cost at an equilibrium to the optimal social cost that could be achieved by the centralized optimization approach. We refer to this as the *coordination ratio* of a mechanism.

In this paper we take a step forward and study *multi-job games*, in which there is a set of agents $\mathcal{A}$ who control arbitrary sets of jobs. Specifically the set of jobs controlled by player $\alpha \in \mathcal{A}$ is denoted by $J(\alpha) \subset \mathcal{J}$ and its cost given a particular schedule is the sum of weighted completion times of its own jobs $\sum_{j \in J(\alpha)} w_j C_j$. As in single-job games, we concentrate on designing coordination mechanisms leading to small coordination ratios, when the social cost is the sum of weighted completion times of all jobs (or equivalently of all agents).

Machine Policies. Throughout we assume that policies are prompt: they do not introduce deliberate idle time. In other words, if jobs $j_1, \ldots, j_k$ are assigned to machine i, then by time $\sum_{\ell=1}^{k} p_{ij_\ell}$ all jobs have been completed and released. Besides distinguishing between local and strongly local policies we distinguish between *nonpreemptive*, *preemptive*, and *randomized* policies. In nonpreemptive policies jobs are processed in some fixed deterministic order that may depend arbitrarily on the set of jobs assigned to the machine (processing time, weight, and ID), and once a job is completed it is released. On the other hand, preemptive policies may suspend a job before it completes in order to execute another job and the suspended job is resumed later. Interestingly, such policies can be considered as nonpreemptive policies, but where jobs may be held back after completion [11,12]. Finally, randomized policies have the additional power that they can schedule jobs at random according to some distribution depending on the assigned jobs' characteristics. Another usual distinction is between policies that are *anonymous* and *non-anonymous.* In the former jobs with the same characteristics (except for IDs) must be treated equally and thus assigned the same completion time. In the latter, jobs may be distinguished using their IDs.

For instance consider the widely used policy known as Smith-rule (**SR**), which sorts jobs in nondecreasing order of their processing time to weight ratios. Formally **SR** processes jobs in nondecreasing order of $\rho_{ij} = p_{ij}/w_j$, and breaks ties using the job's IDs. This policy is strongly local, nonpreemptive, and non-anonymous.

Equilibrium Concepts. For the single-job scheduling game the underlying concept of equilibrium is, quite naturally, that of Nash (NE)[28]. However, once we allow players to control many jobs and endow them with the weighted completion time cost, already computing a best response to a given situation may be NP-complete. Therefore, it is rather unlikely that such an equilibrium will be attained. To overcome this difficulty we consider a weaker equilibrium concept, which we call *weak equilibrium* (WE), namely, a schedule of all jobs is a WE if no player $\alpha \in \mathcal{A}$ can find a job $j \in J(\alpha)$ such that moving j to a different machine will strictly decrease her cost $\sum_{j \in J(\alpha)} w_j C_j$. We extend the WE concept to mixed (randomized) strategies by allowing player α to keep the distribution of all but one job $j \in \mathcal{J}(\alpha)$ and move job j to any machine. Observe that in the single-job game NE and WE coincide. Throughout, we provide bounds on the coordination ratio of policies for the weak equilibrium, and since NE are also WE our bounds hold for NE as well.

As the reader may have noticed, there is a close connection between WE and local optima of the *jump* (also called *move*) neighborhood (e.g. [39]). In a locally optimal solution of $R| \,|\sum w_j C_j$ for the jump neighborhood, no single job $j \in J$ may be moved to a different machine while decreasing the overall cost. Such solution is exactly a WE when a single player in the scheduling game controls all jobs and the machines use **SR**.

To illustrate the concept of weak equilibrium and the difference between the single-job and the multi-job games consider the following example on 4 machines, $m_1, \ldots, m_4$, with the **SR** policy. There are 4 unit-weight jobs called a, b, c, d such that $p_{m_1,a} = 1 + \varepsilon$, $p_{m_1,b} = 1$, $p_{m_2,b} = 1.5$, $p_{m_2,c} = 2$, $p_{m_3,c} = 3$, $p_{m_3,d} = 2$, $p_{m_4,d} = 2$, and all other $p_{ij} = +\infty$. In this situation an equilibrium for the single-job game is that jobs a and b go to m_1, job c goes to m_2, and job d goes to m_3, leading to a total cost of $7 + \varepsilon$. Consider now the multi-job game in which one player controls a, b and another player controls c, d. A NE is obtained when a goes to m_1, b goes to m_2, c goes to m_3, and d goes to m_4, and this has total cost $7.5 + \varepsilon$. A WE is obtained from instance when a goes to m_1, b goes to m_2, c goes to m_2, and d goes to m_3, having a total cost of $8 + \varepsilon$.

Related Literature. The study of coordination mechanisms for single-job scheduling games, taking the makespan as social cost, was initiated by Christodoulou et al. [9]. However the implied bounds on the price of anarchy are constant only for simple environments such as when machines are identical. Indeed, Azar et al. [2], and Fleischer and Svitkina [19] show that, even for a restricted uniform machines environment "almost" no deterministic machine policies can achieve a constant price of anarchy. The result was finally established by Abed and Huang [1], who proved that no symmetric coordination mechanism satisfying the so-called "independence of irrelevant alternatives" property, even if preemption is allowed, can achieve a constant price of anarchy for the makespan objective. The existence of a randomized machine policy with such a desirable property is unknown. It is worth mentioning that there is a vast amount of related work considering the makespan social cost [6,8,14,16,25,27].

The situation changes quite dramatically for the sum of weighted completion times objective. In this case Correa and Queyranne show that, for restricted related machines, smith rule induces a game with price of anarchy at most 4 [13], improving results implied by Farzad et al. [17] and Caragiannis et al [8] obtained in different contexts. Cole et al., extend this result to unrelated machines, and also design an improved preemptive policy, proportional sharing, achieving an approximation bound of 2.618 and an even better randomized policy [11,12]. Further recent works include extensions and improvements by Bhattacharya et al. [3], Cohen et al. [10] and by Rahn and Schäfer [30], Hoeksma and Uetz [22].

Finally, performance guarantee results for the $\sum w_j C_j$ objective using natural local search heuristics are scarse, despite the vast amount of computational work [7,29]. We are only aware of the results of Brueggemann et al. [4] who proved that for identical machines local optima for the jump neighborhood are within a factor of 3/2 of the optimal schedule.

Our Results. We start by considering deterministic policies and prove that the coordination ratio of SR under WE is exactly 4. This generalizes the result for single-job games [12] and therefore it is the best possible coordination ratio that can be achieved nonpreemptively. We prove the upper bound of 4 for SR with mixed WE. This is relevant since a pure strategy NE may not exist in this setting [13]. Moreover, it is unclear whether the smoothness framework of Roughgarden [31] can be applied here: On the one hand our results hold for the more general framework of WE, while on the other hand having players that control multiple jobs makes it more difficult to prove the (λ, μ)-smoothness.

Before designing improved policies we observe that no anonymous policy may obtain a coordination ratio better than 4, and basically no policy, be it preemptive or randomized, local or strongly local, can achieve a coordination ratio better than 2.618. The latter is in sharp contrast with the case in which players control just one job where better ratios can be achieved with randomized policies [12]. Quite surprisingly we are able to design an "optimal" policy, which we call *externality* (EX), that guarantees a coordination ratio of 2.618 for WE. Under this EX policy, jobs are processed according to Smith rule but are held back (and not released) for some additional time after completion. This additional time basically equals the negative externality that this particular job imposes over other players. Additionally, we prove that EX defines a potential game, so that pure WE exists, and that the convergence time is polynomial. It is worth mentioning that in the single-job game EX coincides with the proportional-sharing (PS) policy [12], which in turn extends the EQUI policy of the unit-weight case [16]. On the other side, when a single player controls all jobs, EX coincides with SR. The idea of making jobs incorporate the externality they impose has also been used by Heydenreich et al. [20]. However their goal is different; they incorporate the externality in the form of payments to obtain truthful mechanisms rather than to improve efficiency.

Interestingly, our result for EX in case just one player controls all jobs implies a tight approximation guarantee of 2.618 for local optima under the jump neighborhood for $R||\sum w_j C_j$. This tight guarantee also holds for the *swap* neighborhood, in which one is additionally allowed to swap jobs between machines so long as the objective function value decreases [39]. In addition, our fast convergence result for EX implies another new result, namely, that local search with the jump neighborhood, when only maximum gain steps are taken, converges in polynomial time. These facts appear to be quite surprising since, despite the very large amount of work on local search heuristics for scheduling problems [7,29], performance guarantees, or polynomial time convergence results are are only known for identical machines [4].

Methodologically our work is based on the inner product framework of [12], but more is needed to deal with the multi-job environment. Our main contribution is however conceptual: On the one hand, we demonstrate that the natural economic idea of externalities leads to approximately optimal, and in a way best possible, outcomes even in decentralized systems with only partial information (in a full information and centralized setting one can easily design policies leading to optimal outcomes). On the other hand, we provide the first direct application of purely game-theoretic ideas to the analysis of natural and well studied local search heuristics that lead to the currently best known results.

Preliminaries. Recall that for a player $\alpha \in \mathcal{A}$, the set of job she controls is denoted by $J(\alpha) \subset \mathcal{J}$. Moreover, $\alpha(j)$ denotes the player controlling job j, so that $J(\alpha(j))$ is the set of jobs controlled by who is controlling j.

A *pure strategy profile* is a matrix $\mathbf{x} \in \{0,1\}^{\mathcal{M}\times\mathcal{J}}$ in which $\mathbf{x}_{ij} = 1$ if job j is assigned to machine i. By denoting $\mathbf{x}^{\alpha}$ the columns of $\mathbf{x}$ corresponding to jobs controlled by player α we say that $\mathbf{x}^{\alpha}$ is a pure strategy for this player. A mixed strategy for player α is a probability distribution over all $\mathbf{x}^{\alpha} \in \{0,1\}^{\mathcal{M}\times J(\alpha)}$. A set of mixed strategies for all players $\alpha \in \mathcal{A}$ leads to a (mixed) strategy profile $\mathbf{x} \in [0,1]^{\mathcal{M}\times\mathcal{J}}$ where $\mathbf{x}_{ij}$ is the probability of job j assigned to machine i. Note that the distributions of the different columns of $\mathbf{x}$ may not be independent. We denote by $\mathbf{x}_{-k}$ the matrix obtained by deleting the $k-$th column of $\mathbf{x}$. Observe that $\mathbf{x}_{-k}$ results from the joint probability distribution of all jobs $j' \neq k$ according to $\mathbf{x}$. More precisely $\mathbf{x}_{-k} \in [0,1]^{\mathcal{M}\times\mathcal{J}\setminus\{k\}}$ can be equivalently seen as the mixed strategy profile obtained when players different from $\alpha(k)$ continue using the same strategy, while player $\alpha(k)$ forgets job k and if she was playing the pure strategy $\mathbf{x}^{\alpha(k)} \in \{0,1\}^{\mathcal{M}\times J(\alpha)}$ with probability q, she plays the pure strategy for her jobs different from k, $\mathbf{x}_{-k}^{\alpha(k)} \in \{0,1\}^{\mathcal{M}\times J(\alpha)\setminus\{k\}}$ with probability q (these probabilities add up if she was playing with positive probability two strategies that were equal except for job k). We define $\mathbf{x}_{-K}$ analogously for a set of jobs $K \subseteq \mathcal{J}$.

Given a mechanism $\mathbf{M} \in \{\mathbf{SR}, \mathbf{EX}\}$ and a strategy profile $\mathbf{x}$, $\mathbb{E}[C_j^{\mathbf{M}}(\mathbf{x})]$ is the expected completion time of job j. The conditional expected completion time of job j on machine i when job k is assigned to machine i is denoted $\mathbb{E}[C_j^{\mathbf{M}}(\mathbf{x}_{-k}, k \to i)]$. The expected cost of the strategy profile $\mathbf{x}$ is $\mathbb{E}[C^{\mathbf{M}}(\mathbf{x})] = \sum_{j\in\mathcal{J}} w_j \mathbb{E}[C_j^{\mathbf{M}}(\mathbf{x})]$ and the expected cost of a player α under $\mathbf{x}$ is $\mathbb{E}[C_\alpha^{\mathbf{M}}(\mathbf{x})] = \sum_{j\in J(\alpha)} w_j \mathbb{E}[C_j^{\mathbf{M}}(\mathbf{x})]$. For convenience we also define $\mathbb{E}[C_\alpha^{\mathbf{M}}(\mathbf{x}_{-k}, k \to i)] = \sum_{j\in J(\alpha)} w_j \mathbb{E}[C_j^{\mathbf{M}}(\mathbf{x}_{-k}, k \to i)]$. Note that $\mathbb{E}[C^{\mathbf{M}}(\mathbf{x})] = \sum_{\alpha\in\mathcal{A}} \mathbb{E}[C_\alpha^{\mathbf{M}}(\mathbf{x})]$.

A Nash equilibrium (NE) is therefore a strategy profile $\mathbf{x}$ such that for all player $\alpha \in \mathcal{A}$ and all strategy profiles $\mathbf{y}^{\alpha}$ for player α we have that:

$$\mathbb{E}[C_\alpha^{\mathbf{M}}(\mathbf{x})] \leq \mathbb{E}[C_\alpha^{\mathbf{M}}(\mathbf{y}^{\alpha}, \mathbf{x}_{-J(\alpha)})].$$

Similarly, a weak equilibrium (WE) is a strategy profile $\mathbf{x}$ such that for all player $\alpha \in \mathcal{A}$, all jobs $k \in J(\alpha)$, and all machines $i \in \mathcal{M}$, we have that:

$$\mathbb{E}[C_\alpha^{\mathbf{M}}(\mathbf{x})] \leq \mathbb{E}[C_\alpha^{\mathbf{M}}(\mathbf{x}_{-k}, k \to i)].$$

The optimal assignement is the assignment in which the jobs are processed non-preemptively on the machines so that the cost is minimized. Throughout the paper, $\mathbf{x}^*$ denotes the optimal assignment (thus $\mathbf{x}^*$ is a pure strategy), and we define X_i^* as the set of jobs assigned to machine i under the optimal assignment. Given the assignment of jobs to machines, it is well-known that Smith Rule minimizes the total cost of jobs. Therefore $C^{\mathbf{SR}}(\mathbf{x}^*)$ is the optimal cost.

2 Nonpreemptive Mechanisms

We now study nonpreemptive mechanisms (jobs have IDs, needed to break ties between identically looking jobs) and prove that SR has a coordination ratio of 4 for mixed WE. We work with mixed strategies since SR does not guarantee that existence of pure WE. As mentioned earlier, our result is best possible among nonpreemptive mechanisms [12].

Recall that under SR, each machine i schedules nonpreemptively its assigned jobs j in nondecreasing order of $\rho_{ij} = p_{ij}/w_j$, and ties are broken using the IDs. To simplify the presentation, we say that $\rho_{ik} < \rho_{ij}$ if k comes earlier than j in the SR order of machine i. Thus, given a strategy profile $\mathbf{x}$ we have $\mathbb{E}[C_j^{\text{SR}}(\mathbf{x}_{-j}, j \to i)] = p_{ij} + \sum_{k:\rho_{ik}<\rho_{ij}} \mathbf{x}_{ik} p_{ik}$ so that,

$$\begin{aligned}\mathbb{E}\left[C^{\text{SR}}(\mathbf{x})\right] &= \sum_{j\in\mathcal{J}} w_j \sum_{i\in\mathcal{M}} \mathbf{x}_{ij}\mathbb{E}[C_j^{\text{SR}}(\mathbf{x}_{-j}, j \to i)] \qquad (1)\\ &= \sum_{i\in\mathcal{M}}\sum_{j\in\mathcal{J}} \mathbf{x}_{ij} w_j (p_{ij} + \sum_{k:\rho_{ik}<\rho_{ij}} \mathbf{x}_{ik} p_{ik}).\end{aligned}$$

Extending the inner product space technique of Cole et al. [12], we let $\varphi : \mathbf{x} \to L_2([0,\infty])^{\mathcal{M}}$, which maps every strategy profile $\mathbf{x}$ to a vector of functions (one for each machine) as follows. If $\boldsymbol{f} = \varphi(\mathbf{x})$, then for each $i \in \mathcal{M}$, the i-th component of $\boldsymbol{f}$ is the function $f_i(y) := \sum_{j\in\mathcal{J},\rho_{ij}\geq y} \mathbf{x}_{ij} w_j$. Letting $\langle f_i, g_i\rangle = \int_0^\infty f_i(y) g_i(y) dy$ be the standard inner product on L_2 we get that $\langle \boldsymbol{f}, \mathbf{g}\rangle = \sum_{i\in\mathcal{M}} \langle f_i, g_i\rangle$. Additionally, we let $\eta_i(\mathbf{x}) = \sum_{j\in\mathcal{J}} w_j \mathbf{x}_{ij} p_{ij}$ and $\eta(\mathbf{x}) = \sum_{i\in\mathcal{M}} \eta_i(\mathbf{x})$.

The next lemma and expressions (2) and (3) follow easily from the derivations of Cole et al. [12]. The only difference is that here we need to prove the results for mixed strategies. We defer the proofs of this section to the full version.

Lemma 1. *For a strategy profile* $\mathbf{x}$ *and the optimal assignment* $\mathbf{x}^*$, *let* $\boldsymbol{f} = \varphi(\mathbf{x})$ *and* $\boldsymbol{f}^* = \varphi(\mathbf{x}^*)$. *Then* $\langle f_i, f_i^*\rangle = \sum_{j\in X_i^*}\sum_{k\in\mathcal{J}} w_j w_k \mathbf{x}_{ik} \min\{\rho_{ij}, \rho_{ik}\}$.

Similarly to Lemma 1, and using equation (1), we may evaluate

$$||\varphi(\mathbf{x})||^2 \leq 2\mathbb{E}\left[C^{\text{SR}}(\mathbf{x})\right]. \qquad (2)$$

Additionally, when $\mathbf{x}$ is a pure strategy we have:

$$C^{\text{SR}}(\mathbf{x}) = \frac{1}{2}||\varphi(\mathbf{x})||^2 + \frac{1}{2}\eta(\mathbf{x}). \qquad (3)$$

In what follows, let $\mathbf{x}$ denote a mixed weak equilibrium and $\mathbf{x}^*$ the optimal assignment. Let $\boldsymbol{f} = \varphi(\mathbf{x})$ and $\boldsymbol{f}^* = \varphi(\mathbf{x}^*)$.

Lemma 2. *Consider* $X_i^*(\alpha) = X_i^* \cap J(\alpha)$, *the jobs of player* α *assigned to machine* i *in the optimal solution. Then for each* $j \in X_i^*(\alpha)$ *we have:*

$$w_j \mathbb{E}\left[C_j^{\text{SR}}(\mathbf{x})\right] \leq w_j(p_{ij} + \sum_{k:\rho_{ik}<\rho_{ij}} \mathbf{x}_{ik} p_{ik}) + p_{ij} \sum_{k\in J(\alpha)\setminus\{j\},\rho_{ik}>\rho_{ij}} w_k \mathbf{x}_{ik}.$$

Lemma 3. *For a machine* $i \in \mathcal{M}$, $\sum_{j\in X_i^*} w_j \mathbb{E}[C_j^{\text{SR}}(\mathbf{x})] - \eta_i(\mathbf{x}^*) \leq \langle f_i, f_i^*\rangle$.

Theorem 1. $\mathbb{E}[C^{\text{SR}}(\mathbf{x})] \leq 4C^{\text{SR}}(\mathbf{x}^*)$.

3 Preemptive Mechanisms

Finding policies that beat the coordination ratio of 4 for WE is impossible if we restrict to nonpreemptive ones. This holds even for the single-job game [12], where WE and NE coincide. Therefore we need to consider preemptive or randomized policies. We first observe that even with preemption, if we restrict to anonymous policies, beating the ratio of 4 is not possible. Furthermore, we prove that the absolute limit for basically any policy, be it preemptive or randomized, using even global information, and even if different machines use different policies, is $1 + \phi \approx 2.618$, where ϕ is the golden ratio. The precise set of policies for which this lower bound holds are those such that when machine $i \in M$ is assigned a single job, $j \in J$, then $C_j = p_{ij}$.

As the performance of **SR** coincides in the single-job and multi-job games one may wonder whether natural preemptive policies, that work well in the single-job game, also do in the multi-job game. Unfortunately this is not the case. Indeed we prove that the champion preemptive policy for the single-job game, Proportional-sharing [12,16], has a coordination ratio of at least 5.848 for WE and at least 2.848 for NE. It is thus rather surprising that we can actually achieve this ratio with a fairly natural policy, externality (**EX**). A key ingredient of this policy is that it heavily relies on the ownership of the jobs, a feature that policies for the single-job game certainly do not share.

The results in this section are presented for pure strategy profiles. This is primarily done for simplicity and also because, as we will show later, our preemptive policy induces a potential game and therefore pure WE are guaranteed to exist. Thus, given a pure strategy profile $\mathbf{x}$, we may refer to $\mathbf{x}$ as an assignment, and we may let X_i denote the set of jobs assigned to machine i under $\mathbf{x}$, i.e., $j \in X_i$ if $\mathbf{x}_{ij} = 1$. Let also $X_i(\alpha) = X_i \cap J(\alpha)$ be the set of jobs controlled by player α on this machine i under $\mathbf{x}$.

Recall that in the proportional sharing policy (**PS**) [12], the machine processing power is split among the assigned jobs proportionally to their weight. Given an assignment $\mathbf{x}$, if job j is assigned to machine i, it can be observed that:

$$C_j^{\mathbf{PS}}(\mathbf{x}) = p_{ij} + \sum_{k \in X_i, \rho_{ik} < \rho_{ij}} p_{ik} + p_{ij} \sum_{k \in X_i \setminus \{j\}, \rho_{ik} > \rho_{ij}} \frac{w_k}{w_j}.$$

Proposition 1 ([12]). *Given an assignment* $\mathbf{x}$, $C^{\mathbf{PS}}(\mathbf{x}) = ||\varphi(\mathbf{x})||^2$.

In our externality policy, **EX**, given an assignment $\mathbf{x}$, the machine processes the jobs according to **SR** but once a job is completed, it is delayed for an amount of time accounting for the negative externality it is imposing on other players. Thus in **EX** the cost for the owner of job j due to this job will be

$$w_j C_j^{\mathbf{EX}}(\mathbf{x}) = w_j p_{ij} + w_j \sum_{k \in X_i, \rho_{ik} < \rho_{ij}} p_{ik} + p_{ij} \sum_{k \in X_i \setminus J(\alpha(j)), \rho_{ik} > \rho_{ij}} w_k.$$

The completion time of j is then defined by the previous equation. Observe that in the single-job game, **EX** reduces to **PS**, while if all jobs are controlled by a single player EX reduces to SR. Also, EX induces feasible schedules since no completion time can be smaller than that given by Smith-rule. Policy EX can be seen as a preemptive policy in which jobs are processed as in SR, except for an infinitesimal piece that is processed at the time defined by previous equation. Moreover EX is strongly local and nonanonymous. A consequence of the definitions of SR, PS, and EX is that for a fixed assignment $\mathbf{x}$ their costs satisfy:

$$\begin{aligned} C^{\text{EX}}(\mathbf{x}) &= C^{\text{SR}}(\mathbf{x}) + \sum_{i\in\mathcal{M}}\sum_{j\in X_i} p_{ij} \sum_{k\in X_i\setminus J(\alpha(j)),\rho_{ik}>\rho_{ij}} w_k \\ &= C^{\text{PS}}(\mathbf{x}) - \sum_{i\in\mathcal{M}}\sum_{j\in X_i} p_{ij} \sum_{k\in X_i(\alpha(j)),\rho_{ik}>\rho_{ij}} w_k . \end{aligned} \tag{4}$$

In the following, let $\mathbf{x}^*$ be an optimal assignment and $\mathbf{x}$ a WE. We also let $\varphi(\mathbf{x}) = \boldsymbol{f}$ and $\varphi(\mathbf{x}^*) = \boldsymbol{f}^*$ be as in the previous section.

Lemma 4. *Consider a job $j \in X_i^*$ and assume j is on i' under $\mathbf{x}$. Then*

$$w_j C_j^{\text{EX}}(\mathbf{x}) \leq w_j(p_{ij} + \sum_{\substack{k\in X_i,\\ \rho_{ik}<\rho_{ij}}} p_{ik}) + p_{ij} \sum_{\substack{k\in X_i,\\ \rho_{ik}>\rho_{ij}}} w_k - p_{i'j} \sum_{\substack{k\in X_{i'}(\alpha(j)),\\ \rho_{i'k}>\rho_{i'j}}} w_k .$$

Proof. The case $i' = i$ is immediate. For $i' \neq i$, consider the cost of jobs belonging player $\alpha(j)$ on machines i or i' under $\mathbf{x}$, which is,

$$w_j C_j^{\text{EX}}(\mathbf{x}) + \sum_{k\in((X_i(\alpha)\cup X_{i'}(\alpha))\setminus\{j\}} w_k C_k^{\text{EX}}(\mathbf{x}). \tag{5}$$

Suppose that she moves j from machine i' to i, then the total cost of the same set of jobs is

$$\begin{aligned} &\sum_{k\in((X_i(\alpha)\cup X_{i'}(\alpha))\setminus\{j\}} w_k C_k^{\text{EX}}(\mathbf{x}) - p_{i'j} \sum_{k\in X_{i'}(\alpha(j)),\rho_{i'k}>\rho_{i'j}} w_k + \\ &w_j(p_{ij} + \sum_{\substack{k\in X_i,\\ \rho_{ik}<\rho_{ij}}} p_{ik}) + p_{ij} \sum_{\substack{k\in X_i\setminus J(\alpha(j)),\\ \rho_{ik}>\rho_{ij}}} w_k + p_{ij} \sum_{\substack{k\in X_i(\alpha(j)),\\ \rho_{ik}>\rho_{ij}}} w_k . \end{aligned} \tag{6}$$

Here the second term is the saving of the cost for those jobs $k \in \alpha(j)$ on machine i' that have larger ratios $\rho_{i'k}$ than $\rho_{i'j}$; the third and fourth terms are the cost of job j on machine i; and the fifth term is the increase of the cost of those jobs $k \in \alpha(j)$ on machine i that have larger ratios ρ_{ik} than ρ_{ij}. As $\mathbf{x}$ is a WE, the term (5) is upper bounded by (6). □

Lemma 5. $C^{\text{EX}}(\mathbf{x}) \leq \eta(\mathbf{x}^*) + \langle \boldsymbol{f}, \boldsymbol{f}^* \rangle - \sum_{i\in\mathcal{M}}\sum_{j\in X_i} p_{ij} \sum_{k\in X_i(\alpha(j)),\rho_{ik}>\rho_{ij}} w_k .$

Proof. By Lemma 4 and summing over all jobs in $\mathcal{J}$, we have that the total cost under EX, $\sum_{j\in\mathcal{J}} w_j C_j^{\text{EX}}(\mathbf{x})$ is upper bounded by

$$\eta(\mathbf{x}^*) + \sum_{i\in\mathcal{M}} (\sum_{j\in X_i^*} w_j \sum_{\substack{k\in X_i,\\ \rho_{ik}<\rho_{ij}}} p_{ik} + \sum_{j\in X_i^*} p_{ij} \sum_{\substack{k\in X_i,\\ \rho_{ik}>\rho_{ij}}} w_k - \sum_{j\in X_i} p_{ij} \sum_{\substack{k\in X_i(\alpha(j)),\\ \rho_{ik}>\rho_{ij}}} w_k). \tag{7}$$

By Lemma 1 and the fact that $\mathbf{x}$ is pure, we have

$$\langle f_i, f_i^* \rangle = \sum_{j \in X_i^*} \sum_{k \in X_i} w_j w_k \min\{\rho_{ij}, \rho_{ik}\} = \sum_{j \in X_i^*} (w_j \sum_{\substack{k \in X_i, \\ \rho_{ik} \leq \rho_{ij}}} p_{ik} + p_{ij} \sum_{\substack{k \in X_i, \\ \rho_{ik} > \rho_{ij}}} w_k).$$

Summing over $i \in \mathcal{M}$ and subtracting the latter from (7)

$$C^{\mathrm{EX}}(\mathbf{x}) - \langle \boldsymbol{f}, \boldsymbol{f}^* \rangle \leq \eta(\mathbf{x}^*) - \sum_{i \in \mathcal{M}} \sum_{j \in X_i} p_{ij} \sum_{k \in X_i(\alpha(j)), \rho_{ik} > \rho_{ij}} w_k. \qquad \square$$

Theorem 2. *Let ϕ be the golden ratio. Then $C^{\mathrm{EX}}(\mathbf{x}) \leq (1+\phi) C^{\mathrm{SR}}(\mathbf{x}^*)$.*

Proof. Lemma 5 and Cauchy-Schwartz inequality imply that for $\beta > 1/4$

$$\begin{aligned} C^{\mathrm{EX}}(\mathbf{x}) &\leq \eta(\mathbf{x}^*) + \beta ||\boldsymbol{f}^*||^2 + \frac{1}{4\beta} ||\boldsymbol{f}||^2 - \sum_{i \in \mathcal{M}} \sum_{j \in X_i} p_{ij} \sum_{k \in X_i(\alpha(j)), \rho_{ik} > \rho_{ij}} w_k \\ &\leq \eta(\mathbf{x}^*) + \beta ||\boldsymbol{f}^*||^2 + \frac{1}{4\beta} ||\boldsymbol{f}||^2 - \frac{1}{4\beta} \sum_{i \in \mathcal{M}} \sum_{j \in X_i} p_{ij} \sum_{k \in X_i(\alpha(j)), \rho_{ik} > \rho_{ij}} w_k \\ &\leq \eta(\mathbf{x}^*) + 2\beta C^{\mathrm{SR}}(\mathbf{x}^*) - \beta \eta(\mathbf{x}^*) + \frac{1}{4\beta} C^{\mathrm{EX}}(\mathbf{x}) \\ &\leq (\beta + 1) C^{\mathrm{SR}}(\mathbf{x}^*) + \frac{1}{4\beta} C^{\mathrm{EX}}(\mathbf{x}), \end{aligned}$$

where the third inequality follows from equation (3), from Proposition 1 and from equation (4). By letting $\beta = \frac{1+\sqrt{5}}{4}$ the result follows. $\square$

As mentioned earlier, it turns out that EX is best possible. The proof of this fact is deferred to the full version.

Theorem 3. *The coordination ratio for weak equilibrium of any prompt mechanism is at least $1 + \phi$.*

4 Final Remarks

We have proved that SR is the best possible nonpreemptive policy, and to beat its coordination ratio we have used EX, a policy that, as opposed to SR, importantly relies on who owns which job. We conjecture that if we restrict to policies that ignore the ownership of the jobs the ratio of 4 cannot be improved. This is indeed the case for nonpreemptive policies, and also for fully preemptive policies. Also, for natural policies with this property such as PS or the RAND policy [12] the technique in this paper only lead to larger bounds.

Our lower bound on general prompt seems to be the natural limit. Non-prompt policies that are allowed to use global information can certainly beat this as they can simply introduce very large delays for jobs that are not assigned to it in an optimal schedule. By doing this, such policies can easily achieve low coordination ratio (say optimal if they have unlimited computational power or

3/2 if they use the best known approximation algorithms. It would be interesting to explore what happens with this non-prompt policies when they can only use local information.

Another interesting question refers to the quality of the actual NE of this game. Of course our upper bounds applies to that equilibrium concept, and furthermore we know that the coordination ratio of **EX** for NE is exactly 2.618 as in the single job case it coincides with **PS** [12]. However it may be possible that another deterministic policy has a better coordination ratio for NE.

Finally, we note that by mimicking the analysis in [12] we obtain a similar $2+\varepsilon$ approximation algorithm for $R||\sum w_jC_j$, independent of which jobs belong to which players. It is possible that by carefully choosing the game structure this can be beaten.

References

1. Abed, F., Huang, C.-C.: Preemptive coordination mechanisms for unrelated machines. In: Epstein, L., Ferragina, P. (eds.) ESA 2012. LNCS, vol. 7501, pp. 12–23. Springer, Heidelberg (2012)
2. Azar, Y., Jain, K., Mirrokni, V.S. (Almost) Optimal coordination mechanisms for unrelated machine scheduling. In: SODA (2008)
3. Bhattacharyay, S., Imz, S., Kulkarnix, J., Munagala, K.: Coordination mechanisms from (almost) all scheduling policies. In: ITCS (2014)
4. Brueggemann, T., Hurink, J.L., Kern, W.: Quality of move-optimal schedules for minimizing total weighted completion time. Oper. Res. Lett. 34(5), 583–590 (2006)
5. Bruno, J., Coffman, E.G., Sethi, R.: Scheduling independent tasks to reduce mean finishing time. Commun. ACM 17, 382–387 (1974)
6. Caragiannis, I.: Efficient coordination mechanisms for unrelated machine scheduling. In: SODA (2009)
7. Chen, B., Potts, C.N., Woeginger, G.J.: A review of machine scheduling: Complexity, algorithms and approximability. In: Handbook of Combinatorial Optimization, vol. 3, Kluwer Academic Publishers (1998)
8. Caragiannis, I., Flammini, M., Kaklamanis, C., Kanellopoulos, P., Moscardelli, L.: Tight bounds for selfish and greedy load balancing. Algorithmica 61(3), 606–637 (2011)
9. Christodoulou, G., Koutsoupias, E., Nanavati, A.: Coordination mechanisms. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 345–357. Springer, Heidelberg (2004)
10. Cohen, J., Dürr, C., Thang, N.K.: Smooth inequalities and equilibrium inefficiency in scheduling games. In: Goldberg, P.W. (ed.) WINE 2012. LNCS, vol. 7695, pp. 350–363. Springer, Heidelberg (2012)
11. Cole, R., Correa, J.R., Gkatzelis, V., Mirrokni, V.S., Olver Inner, N.: product spaces for MinSum coordination mechanisms. In: STOC (2011)
12. Cole, R., Correa, J.R., Gkatzelis, V., Mirrokni, V., Olver, N.: Decentralized utilitarian mechanisms for scheduling games. In: Game. Econ. Behav. (to appear)
13. Correa, J.R., Queyranne, M.: Efficiency of equilibria in restricted uniform machine scheduling with total weighted completion time as social cost. Naval Res. Logist. 59, 384–395 (2012)
14. Czumaj, A., Vöcking, B.: Tight bounds for worst-case equilibria. ACM T. Algo. 3 (2007)
15. Davis, E., Jaffe, J.M.: Algorithms for scheduling tasks on unrelated processors. J. ACM 28(4), 721–736 (1981)

16. Dürr, C., Nguyen, K.T.: Non-clairvoyant scheduling games. In: Mavronicolas, M., Papadopoulou, V.G. (eds.) SAGT 2009. LNCS, vol. 5814, pp. 135–146. Springer, Heidelberg (2009)
17. Farzad, B., Olver, N., Vetta, A.: A priority-based model of routing. Chic. J. Theor. Comput. (2008)
18. Finn, G., Horowitz, E.: A linear time approximation algorithm for multiprocessor scheduling. BIT 19, 312–320 (1979)
19. Fleischer, L., Svitkina, Z.: Preference-constrained oriented matching. In: ANALCO (2010)
20. Heydenreich, B., Müller, R., Uetz, M.: Mechanism Design for Decentralized Online Machine Scheduling. Oper. Res. 58(2), 445–457 (2010)
21. Hoogeveen, H., Schuurman, P., Woeginger, G.J.: Non-approximability results for scheduling problems with minsum criteria. In: Bixby, R.E., Boyd, E.A., Ríos-Mercado, R.Z. (eds.) IPCO 1998. LNCS, vol. 1412, p. 353. Springer, Heidelberg (1998)
22. Hoeksma, R., Uetz, M.: The Price of Anarchy for Minsum Related Machine Scheduling. In: Solis-Oba, R., Persiano, G. (eds.) WAOA 2011. LNCS, vol. 7164, pp. 261–273. Springer, Heidelberg (2012)
23. Horn, W.A.: Minimizing average flow time with parallel machines. Oper. Res. 21(3), 846–847 (1973)
24. Ibarra, O.H., Kim, C.E.: Heuristic algorithms for scheduling independent tasks on nonidentical processors. J. ACM 24(2), 280–289 (1977)
25. Immorlica, N., Li, L., Mirrokni, V.S., Schulz, A.S.: Coordination mechanisms for selfish scheduling. Theor. Comput. Sci. 410(17), 1589–1598 (2009)
26. Koutsoupias, E., Papadimitriou, C.: Worst-case equilibria. In: Meinel, C., Tison, S. (eds.) STACS 1999. LNCS, vol. 1563, p. 404. Springer, Heidelberg (1999)
27. Lu, P., Yu, C.: Worst-Case Nash Equilibria in Restricted Routing. In: Papadimitriou, C., Zhang, S. (eds.) WINE 2008. LNCS, vol. 5385, pp. 231–238. Springer, Heidelberg (2008)
28. Nash, J.: Equilibrium points in N-person games. PNAS 36, 48–49 (1950)
29. Potts, C.N., Strusevich, V.: Fifty years of scheduling: a survey of milestones. J Oper. Res. Society 60(1), 41–68 (2009)
30. Rahn, M., Schäfer, G.: Bounding the inefficiency of altruism through social contribution games (2013) (manuscript)
31. Roughgarden, T.: Intrinsic robustness of the price of anarchy. In: STOC (2009)
32. Sahni, S., Cho, Y.: Bounds for list schedules on uniform processors. SIAM J. Comput. 9, 91–103 (1980)
33. Schulz, A.S., Skutella, M.: Scheduling unrelated machines by randomized rounding. SIAM J. Discrete Math. 15(4), 450–469 (2002)
34. Schuurman, P., Vredeveld, T.: Performance guarantees of local search for multiprocessor scheduling. In: Aardal, K., Gerards, B. (eds.) IPCO 2001. LNCS, vol. 2081, p. 370. Springer, Heidelberg (2001)
35. Sethuraman, J., Squillante, M.S.: Optimal scheduling of multiclass parallel machines. In: SODA (1999)
36. Skutella, M.: Convex quadratic and semidefinite programming relaxations in scheduling. J. ACM 48(2), 206–242 (2001)
37. Skutella, M., Woeginger, G.J.: A ptas for minimizing the total weighted completion time on identical parallel machines. Math. Oper. Res. 25(1), 63–75 (2000)
38. Smith, W.: Various optimizers for single stage production. Naval Res. Logist. Quart. 3(1-2), 59–66 (1956)
39. Vredeveld, T., Hurkens, C.: Experimental comparison of approximation algorithms for scheduling unrelated parallel machines. INFORMS J. Comput. 14, 175–189 (2002)

Theory and Practice of Chunked Sequences

Umut A. Acar[1,2], Arthur Charguéraud[1,3], and Mike Rainey[1]

[1] Inria
[2] Carnegie Mellon University
[3] LRI, Université Paris Sud, CNRS

Abstract. Sequence data structures, i.e., data structures that provide operations on an ordered set of items, are heavily used by many applications. For sequence data structures to be efficient in practice, it is important to amortize expensive data-structural operations by *chunking* a relatively small, constant number of items together, and representing them by using a simple but fast (at least in the small scale) sequence data structure, such as an array or a ring buffer. In this paper, we present chunking techniques, one direct and one based on bootstrapping, that can reduce the practical overheads of sophisticated sequence data structures, such as finger trees, making them competitive in practice with special-purpose data structures. We prove amortized bounds showing that our chunking techniques reduce runtime by amortizing expensive operations over a user-defined chunk-capacity parameter. We implement our techniques and show that they perform well in practice by conducting an empirical evaluation. Our evaluation features comparisons with other carefully engineered and optimized implementations.

1 Introduction

Sequence data structures, i.e., data structures that store an ordered set of elements and support operations on them, are fundamental in computer science. There exist several variants of sequences, such as LIFO queues (stacks), FIFO queues, doubly-ended queues (deques), and more general data structures, such as finger-search trees. The common operations on sequences include push and pop operations at one or two ends, a split operation that partitions the data structure at a desired position, and a concatenation operation that joins two sequences.

Many asymptotically efficient data structures for sequences have been developed. Resizable circular arrays support constant-time push, pop and random access operations, but require linear time for concatenation and splitting. Doubly-linked lists improve the bound for concatenation to $O(1)$, but splitting at a given index requires linear time. More sophisticated data structures, such as Kaplan and Tarjan's functional catenable sorted lists, support push and pop operations in constant time, while also supporting splitting and concatenation in logarithmic time [10]. Their catenable sorted list is one instance of a finger search tree, a type of tree that has been studied extensively since the 1970s [7].

A more recent functional finger-tree data structure by Hinze and Paterson achieves similar bounds and accepts a simple implementation [8].

Practical performance is a major concern for sequence data structures because of their widespread use in applications. While there has been much focus on developing asymptotically efficient sequence data structures, there is relatively little rigorous work on practical data structures that can guarantee small constant factors on modern computers. To understand the significance of practical concerns we implemented in C++ an optimized version of Hinze and Paterson finger tree data structure [8], and compared it to a resizable circular array, which is simpler but asymptotically efficient only for a narrower set of operations including push and pop. Our experiments show that the finger tree is over 20 times slower for push/pop operations than with circular arrays.[1] This is unfortunate, because such gaps in performance can prevent the use of these asymptotically efficient data structures in practice. It would be nice to have the best of both worlds by guaranteeing both theoretical and practical efficiency. We are therefore interested in the question: *can we design asymptotically and practically efficient data structures for sequences that can support a broad range of operations, including push/pop operations on both ends, concatenation, and split at a specified position?*

In practice, simpler data structures can out-class sophisticated, asymptotically efficient data structures because the latter tends to perform many more expensive operations, such as memory operations and manipulations of tree nodes, than their simpler counterparts. To reduce such overheads, practitioners represent a sequence data structure as a hierarchical data structure consisting of an *underlying sequence* data structure that stores *chunks* of items instead of individual items. Each chunk in turn is represented as an array, which is basically a fast sequence data structure (in small scale). The idea is to amortize the cost of expensive memory operations on the underlying sequence over the items in the chunks. This chunking technique can be applied to essentially any underlying sequence data structure. For example, the C++ Standard Template Library (STL) [14] includes a deque data structure represented as a resizable circular array of chunks of 512 single-word items. Similarly, the Haskell "yi" package [2] provides a chunked finger-tree data structure for character sequences. While chunking can be effective in practice, all applications of this technique known to us are merely heuristics: they provide no worst-case efficiency guarantees. In fact, as we describe in Section 2, their time and space efficiency can degenerate significantly on certain sequences of operations.

In this paper, we give chunking algorithms that yield tight amortized, worst-case bounds with small constant factors. To support splits and random accesses efficiently, we consider a slightly more general interface for sequences: we associate weights with the items and support a *weighted-split* operation. The weighted-split operation takes a sequence s and a weight w, and it decomposes s in three parts (s_1, x, s_2), in such a way that $|s_1| \leq w < |s_1| + |x|$, where $|x|$

[1] We specifically measured the time for pushing 100 million integers and then popping them in FIFO order.

denotes the weight of the item x, and $|s_1|$ denotes the sum of the weights of the items in s_1.

Given any underlying weighted sequence data structure, we show in Section 3 how to build a weighted (or unweighted) sequence data structure by using K-capacity chunks

– that guarantees constant-time push and pop operations with excellent constant factors, in particular such that every allocation operation is amortized over at least K push or pop operations,
– that supports concatenation and split efficiently by introducing an additive overhead proportional to K, and
– that requires approximately a factor-2 increase in space usage, thus ensuring reasonably good space utilization.

At a high level, our techniques speed up the push and pop operations (usually the most common operations) without significantly affecting the performance for the other operations. We note that in this paper, we consider ephemeral (as opposed to persistent) data structures only.

Since our techniques can be applied to any sequence data structure, including to a chunked sequence data structure, it can be applied recursively to permit bootstrapping. We describe such a bootstrapped data structure in Section 4, which uses structural decomposition [6,4,3] and recursive slowdown [9].

In our proofs, in addition to considering the chunk size as a parameter, we also differentiate between memory allocation and other operations. For memory allocation, we introduce a parameter A to denote the cost of allocating and later deallocating a structure of bounded-size (e.g., a chunk or a record), and reserve the $O(1)$ notation to account for the other (relatively cheaper) operations. We show that all allocation operations are well amortized. As we describe briefly, in our chunking technique, allocation correlates with other expensive memory operations. This approach thus gives us a good indication of practical overheads.

To understand the actual practical efficiency of our proposed techniques, we have implemented them all in C++. We perform an empirical evaluation by comparing our data structures to more specialized data structures that are optimized for a narrower set of operations such as STL deques and ropes, which are carefully engineered and highly optimized. Our practical results confirm our theoretical results showing that our data structures perform well in practice, usually within 10% of the actual run time of the best known data structure, while still supporting a broader set of operations.

The contributions of the paper include the chunking techniques that guarantee worst-case bounds, their analysing and the proofs, the bootstrapped data structure, and the implementation and its evaluation. Our implementations and test scripts are available for download at `http://deepsea.inria.fr/chunkedseq/`.

2 Challenges

We consider common chunking strategies used in prior implementations such as those employed by the Standard Template Library for C++ and identify two limitations that can lead to significantly degraded performance and underutilization of memory (space) by breaking the amortization benefits of chunking.

Push-Pop Sequences. A common chunking strategy is to create and dispose of chunks on a need by need basis. For example, to push an item x to the front of a sequence, we first check if there is space in the first chunk. If so, we push x into that chunk. Otherwise, we create a new chunk, place x in it, and push this chunk to the front. Symmetrically, to pop an item from the front, we extract the first item stored in the first chunk. If the first chunk becomes empty as a result, then we pop the chunk from the front and dispose of it.

This strategy can fail to amortize the cost of push/pop operations on chunks, which are expensive. For example, starting from a sequence whose front chunk is full, repeat the following pattern: push one item and pop it immediately. It is not difficult to see that each operation requires pushing/popping a chunk. This chunking strategy, employed by the C++ Standard Template Library (STL) Deques, runs 10 times slower in the worst case. To test this, we wrote a program that starting from an initial deque obtained by pushing a given number of items, performs a sequence of push and pop operations on 64-bit integers. The programs runs 10 times slower when the initial deque has a size equal to 511 modulo 512 than with a different initial deque. All chunked sequence data structures that we have seen (and their naive variants) suffer from the same or similar problems.

Sparse Chunks. Chunking delivers efficiency improvements by amortizing the cost of slow operations over a number of fast operations. Such amortization works, of course, only if chunks are densely populated. When chunks are sparsely populated, then the amortization arguments breaks and performance and memory utilization drops. For example, if chunks have capacity K but store only 1 item, then amortization fails entirely and the memory footprint of the sequence is roughly K times bigger than necessary. It is not difficult to create sparse chunks by using concatenation operations. Consider for example a chunked sequence consisting of 2 chunks each containing a single item. Such a sequence can be obtained by pushing $K + 1$ items to the front, then popping $K - 1$ from the back, where K is the capacity of a chunk. Once we have two sequences each with two sparse chunks, we can create one with arbitrary number of chunks by repeatedly concatenating them.

The "yi" package of Haskell [2] implements a refinement of this strategy: to concatenate two sequences s_1 and s_2, first check whether the back chunk in s_1 and the front chunk in s_2 would fit into a single chunk; if so, merge these two chunks before concatenating the underlying sequences. This strategy does not prevent sparse chunks. For example, the concatenation of two sequences each made of two chunks of size 1 produces a sequence made of three chunks of size 1, 2, and 1. Concatenating two such sequences produces a sequence made of chunks of size 1, 2, 2, 2, and 1. By iterating the process, we obtain an arbitrarily-long sequence made of sparse chunks containing no more than 2 items each. This example demonstrates that a provably efficient chunking strategy requires techniques to prevent sparse chunks from being formed.

3 Efficient Chunked Sequences

One of our main results is a theorem (Theorem 1 below) that shows that chunking can be applied to any (underlying) sequence data structure. The theorem states the bounds for the resulting chunked sequence, parametrized by the bounds of the underlying sequence. To simplify the analysis, we combine the cost of push and pop. More precisely, we charge all the cost of a pop operation to the push operation associated with the corresponding item. Doing so is correct because we consider ephemeral sequences and conduct an amortized analysis. For the theorem, we define a *chunk* as a circular array of fixed capacity K and we assume that cost function for the underlying sequence (e.g., $\mathcal{C}_{split}(n)$) are nondecreasing functions of size.

Theorem 1 (Efficiency of Chunked Sequence). *Consider an underlying weighted sequence that supports the following operations:*

- *Push and pop, with cost $\mathcal{C}_{pushpop}$. For simplicity, we assume this cost to not depend on the number of items in the sequence.*
- *Concatenation, with cost $\mathcal{C}_{concat}(n)$, where n is the minimum of the sizes of the two input sequences.*
- *Weighted split, with cost $\mathcal{C}_{split}(n)$, where n is the minimum of the sizes of the two output sequences.*
- *Space usage bounded by $\mathcal{C}_{space}(n)$, where n is the number of single-word items stored in the sequence.*

Let $K \geq 2$ denote the capacity of a chunk, a value that may be freely chosen. Assume that chunks are implemented with a structure that supports $O(1)$ push and pop operations and that requires $K+3$ words to store K single-word items —e.g., using fixed-capacity circular arrays. Recall that A denotes the cost of allocation, including subsequent deallocation.

Then, we can implement a (weighted or unweighted) sequence that achieves the amortized bounds shown below, where, for each operation, n is a size defined as above, and where $p_n = \lfloor \frac{2(n-1)}{K+1} \rfloor + 1$, for whatever the local definition of $n > 0$ is. Intuitively, p_n bounds the number of chunks stored in the underlying sequence.

- *Push and pop, with cost: $O(1) + \frac{1}{K}\big(A + \mathcal{C}_{pushpop}\big)$.*
- *Concatenation, with cost: $\mathcal{C}_{concat}(p_n) + O(K) + 4 \cdot \mathcal{C}_{pushpop}$.*
- *Split, or weighted split, with cost: $\mathcal{C}_{split}(p_n) + O(K) + 6A$.*
- *Space usage, bounded by: $2(1 + \frac{2}{K+1}) \cdot n + \mathcal{C}_{space}(p_n) + 5K + O(1)$ words.*

We present the representation and the invariants of the data structure that satisfies Theorem 1 and describe the implementation of the operations. The proof of the theorem can be found in the long version [1].

Representation. As discussed in Section 2, the main challenge in efficient chunking as required by Theorem 1 is to ensure that all operations on the underlying sequence data structure, which stores chunks are well amortized. To ensure such amortization, we use a representation that keeps two chunks to store the items at the front of the sequence, and two chunks to store the items at the back. We refer to each of the special chunks stored at the two ends as a *buffer*. We then represent a sequence as a quintuple made of a *front-outer buffer*, a *front-inner buffer*, a *middle sequence*, which is an underlying sequence of chunks, a *back-inner buffer*, and a *back-outer buffer*. We write, e.g., (f', f, m, b, b') to denote such a quintuple.

Invariants. To guarantee efficiency, we maintain the invariant that the inner buffers are, at all time, either completely empty or completely full. Moreover, chunks in the middle sequence are never empty, and, to prevent sparse chunks from being formed, we ensure that any 2 consecutive chunks from the middle sequence have an average density of more than 50%. Our invariants are summarized as shown below, where $|c|$ denotes the number of items stored in a chunk c.

1. The front-inner and the back-inner buffers are either empty or full.
2. If c is a chunk from the middle sequence, then $0 < |c| \leq K$.
3. If c and c' are two consecutive chunks in the middle sequence, $|c| + |c'| > K$.

Operations. We implement the sequence operations as described below.

push-Front. Consider a sequence (f', f, m, b, b') and an item x to push to the front of this sequence. If f' is full, we make room as follows. If f is empty, we simply exchange f with f', by swapping pointers. Otherwise, if f is full, we update the sequence to (c, f', m', b, b'), where c is a fresh chunk and where m' is the result of pushing the full chunk f to the front of m. At this point, the front-outer buffer is not full, so we push x to the front of this buffer.

pop-Front. Consider a sequence (f', f, m, b, b'). If f' is empty, we populate it as follows. If f is not empty, in which case it must be full, we swap f with f'. Otherwise, assume f to be empty. If m is not empty, we pop from m, obtaining a nonempty chunk c and a new middle sequence m'; we then update the sequence to (c, f, m', b, b'). Otherwise, assume m to be empty. If b is not empty, in which case it must be full, we swap b with f'. Otherwise, if b is empty, we swap b' with f'. (Alternatively, we may directly pop from the front of b'.) At this point, the front-outer buffer is not empty, so we can pop from this buffer.

push-Buffer-Back. This auxiliary function is used to implement concat. When applied to a middle sequence m and to a chunk c, the function push-buffer-back modifies m so as to concatenate the items from c at its back, proceeding as follows. If c is empty, there is nothing to do. Otherwise, we perform the following two steps. (1) If m is nonempty and has a back chunk c' such that $|c| + |c'| \leq K$, then we pop c' out of m and merge the items from c' into c. (2) We push the chunk c to the back of m.

push-back and **pop-back** and **push-buffer-front** are defined symmetrically.

concat. Consider two sequences $(f_1', f_1, m_1, b_1, b_1')$ and $(f_2', f_2, m_2, b_2, b_2')$. To concatenate them, we start by concatenating the chunks b_1, b_1' at the back of m_1, by applying twice the function push-buffer-back. Symmetrically, we concatenate f_2' and f_2 to the front of m_2, using push-buffer-front. If m_1 and m_2 are both nonempty at this point, let c_1 be the back chunk of m_1 and c_2 be the front chunk of m_2. If $|c_1| + |c_2| \leq K$, then we pop c_1 and c_2, merge the items from c_2 into c_1, and push c_1 back into m_1. (Remark: the pop and push operations on c_1 may be factorized with the earlier calls to push-buffer-back.) At this point, we concatenate the two underlying sequences m_1 and m_2 to get a new middle sequence, call it m_{12}. The final result of the concatenation is $(f'_1, f_1, m_{12}, b_2, b'_2)$.

split. Consider a sequence (f', f, m, b, b') and an index i denoting the split position. There are five cases; we consider the first one that applies.

- Case $i \leq |f'|$. We return two sequences $(f'_1, \emptyset, \emptyset, \emptyset, \emptyset)$ and (f'_2, f, m, b, b'), where (f'_1, f'_2) is the result of splitting the chunk f' at index i. More precisely, f'_1 denotes f' restricted to its items stored at index less than i, and f'_2 denotes a fresh chunk into which we move the items at index i or more in f'.
- Case $i \leq |f'| + |f|$. We return two sequences $(f', \emptyset, \emptyset, \emptyset, f_1)$ and $(f_2, \emptyset, m, b, b')$, where (f_1, f_2) is the result of splitting the chunk f at index $i - |f'|$.
- Case $i \leq |f'| + |f| + |m|$, where $|m|$ denotes the total number of items stored in all the chunks of m. Let j be equal to $i - |f'| - |f|$. We invoke the weighted split operation on the middle sequence to split m into a triple (m_1, c, m_2), such that the chunk c contains the item located at index j in m. Let (c_1, c_2) is the result of splitting the chunk c at index $j - |m_1|$, where $|m_1|$ denotes the weight of m_1 (i.e., the sum of the weights of the chunks in m_1). We then return the two sequences $(f', f, m_1, \emptyset, c_1)$ and $(c_2, \emptyset, m_2, b, b')$.
- The remaining two cases, $i \leq |f'| + |f| + |m| + |b|$ and $i > |f'| + |f| + |m| + |b|$ are essentially symmetrical to the first two cases.

4 Bootstrapped Chunked Sequences

The construction presented in Section 3 shows that, we can build a chunked sequence data structure on top of an underlying weighted sequence data structure. We can thus build a *bootstrapped* weighted sequence data structure by instantiating the underlying sequence to the structure produced by the theorem. To initiate the bootstrapping process, we can use a single chunk. The resulting bootstrapped chunked sequence data structure is a weighted sequence that, for a fixed value of K, achieves the asymptotic bounds as finger trees: constant time push and pop operations at the two ends, and logarithmic time concatenation and split. Unlike finger trees, however, our structure achieves constant factors amortized over K for push and pop operations, without significantly increasing the constant factors in concatenation and split. The precise bounds for our bootstrapped structure are as follows.

Theorem 2 (Efficiency of Bootstrapped Chunked Sequence). *A bootstrapped chunked sequence has depth zero when $n \leq 1$, and has depth $d \leq \lfloor \log_{(K+1)/2} n \rfloor + 1$ otherwise. It achieves the following bounds:*

- *Push and pop, with cost: $O(1) + \frac{4A}{K-1}$.*
- *Concatenation, with cost: $(d+1) \cdot \big(O(K) + \frac{16A}{K-1}\big)$.*

– *Weighted split, with cost:* $(d+1) \cdot \big(O(K) + 6A\big)$.
– *Space usage, with a bound asymptotically equivalent to:* $2(1 + \frac{4}{K-1}) \cdot n$.

At first approximation, our bootstrapped data structure implements push and pop in $O(1) + \frac{A}{K}$, and concatenation and weighted split in $O(K \cdot \log_{K/2} n)$. Since $\log_{K/2} n$ is a rather small value the concatenation and split operations are competitive with the corresponding operations on finger trees, of cost $O(\log_2 n)$, with small values of K.

We note that since the bootstrapped data structure stores chunks of chunks (of chunks and so on), its nodes have high fanout, like some other data structures such as B+ trees [11]. A benefit of large fanout is that it decreases depth. Unlike B+ trees, however, our structure stores both ends of the sequence very close to the root, achieving constant-time access to the ends of the sequence.

We present the representation and the invariants of the data structure that satisfies Theorem 2 and describe the implementation of the operations. The proof of the theorem can be found in the long version [1].

Representation. We represent a bootstrapped chunked sequence as a list of *levels*. The deepest level is a *shallow* level that consists of a single weighted chunk. Every other level is a *deep* level that consists of a weight field and of pointers to the front-outer, front-inner, back-inner and back-outer weighted chunks. Chunks attached at depth 0 store individual items, chunks attached at depth 1 one store chunks of items, chunks at depth 2 store chunks of chunks of items, and so on...

We may choose different chunk capacities for different levels. However, our goal is to minimize both the product of the chunk sizes (to reduce the depth) and the sum of the chunk sizes (for fast split and concatenation). It therefore makes sense to select the same chunk capacity at every level.

Invariant. We enforce that if a level stores zero or one element (which may be items or chunks, depending on the level), then it is shallow. For all but the last level, we enforce the same invariants as those presented previously in Section 3.

Operations. We implement the sequence operations as described below. Operations on deep levels are similar to those described in Section 3, making recursive calls on the lower levels of the bootstrapped structure when operating on the middle sequence. Operations on deep levels also require updating the weight field. Below, we only focus on the treatment of shallow levels and the transitions between shallow and deep levels.

check. The purpose of this auxiliary function is to enforce the invariant that if a level contains zero or one element, then it is shallow. To that end, if the sequence is deep, we execute the following two steps, in order. (1) If all four buffers are empty and the middle sequence is nonempty, we pop a chunk from the front of the middle sequence and set it as new front-outer buffer. (2) If the sequence has an empty middle sequence, and all four buffers contain zero or one item in total, then we change the representation of the sequence to shallow (reusing one of the four buffers as chunk to represent the shallow level).

push-Front. First, if the sequence is shallow and is made of a full chunk, we change its representation to deep, setting the chunk as back-outer buffer. Then, we push the incoming item to the front of the (shallow or deep) level.

pop-Front. We pop an item from the structure, which may be shallow or deep. If the structure is deep, then we call check to possibly make it shallow.

concat. If both structures are deep, we call the concatenation procedure described in Section 3, then call check on the result. Else, we pop the items of the shortest sequence one by one and push them into the other one.

split. If the structure is shallow, we split its chunk at the appropriate position in order to isolate the targeted item, and we produce two shallow structures. If the structure is deep, we split it and then call check on both subsequences.

5 Benchmarks

To evaluate our chunking techniques, we wrote an implementation in C++ consisting of a few generic classes and two data structures that we benchmark. The first class is a generic C++ class that implements our chunking technique of Section 3. This chunked-sequence class is a templated class that is parameterized over the representation of its underlying sequence. Recall that we define the underlying sequence as any underlying sequence data structure that provides the full set of operations for maintaining a sequence of chunks. For the first data structure we benchmarked, we used an instantiation of our chunked-sequence class for which the underlying sequence is represented by our own ephemeral C++ implementation of Hinze and Patterson's finger tree. In addition, we coded a C++ class that implements our bootstrapped chunked sequence of Section 4. For the second data structure we benchmarked, we used an instantiation of our chunked-sequence class for which the underlying sequence is represented by our bootstrapped chunked sequence.

We ran all of our experiments with the same settings for K (i.e., chunk capacity) that we found to deliver good performance overall. For our chunked finger tree, we used 512; for our bootstrapped chunked sequence, we used 512 and 32 for the chunk-capacity settings of the outer and underlying sequences, respectively. We compiled all programs with GCC version 4.9.0, using optimizations `-O2 -march=native`. For the measurements we report in the abstract, we considered an Ubuntu Linux machine with kernel `v3.2.0-58-generic` and an 2.4GHz Intel Xeon 4870 processor with 1TB of RAM. We have obtained similar results on an AMD machine.

Our first study is a comparison between our chunked data structures and the STL deque, which as discussed earlier is also a chunked data structure that uses a chunk-capacity setting of 512 items. To measure the relative efficiency of long sequences of similar accesses to the ends of the sequence, we ran two simple benchmarks, namely LIFO and FIFO. Our LIFO benchmark proceeds in two steps: the first step is to fill a previously empty target sequence by pushing on the back end n 64-bit items and the second is to empty the target sequence by popping repeatedly from the back of the sequence. Our FIFO benchmark does the same thing as LIFO but pops from the front instead of the back end. Table 5 shows the data from our experiments. The results in the first six rows of the table show that our two chunked data structures are at worst a few percent slower than the STL deque.

To measure the relative efficiency of interleaved sequences of pushes and pops, we ran experiments involving the depth-first and breadth-first search of a directed graph. Our depth-first and breadth-first codes are serial implementations of DFS and BFS that are each parameterized by C++ template parameter over the representation of their respective frontiers (i.e., lifo stack and fifo queue ADTs). We considered three graphs that each demonstrates key characteristics of our sequence data structures. Each graph is represented in adjacency-list format and uses 64-bit integer values to represent vertex ids. Looking at DFS and BFS, we see that, in every case except for DFS on tree, our chunked data structures are competitive with STL deques — sometimes slower and sometimes faster, but never differing by more than a few percent. In the case of DFS on tree, our chunked finger tree and bootstrapped chunked sequence are each nearly 40% slower than STL deque. This benchmark demonstrates a weakness of our implementations: the empty check is relatively costly because of the need to frequently check the emptiness of the two inner buffers and the middle sequence each time around main the DFS loop. The cost of the empty check is so pronounced in this particular case because the cost is not well amortized by sufficiently many push operations: the peak size of the DFS frontier is just a few tens of items. Although it affects implementations, this weakness is not inherent to our general technique. If performance on such small sequences is important, one can adjust the code to sacrifice a few instructions on each push and pop operation to cache the size of the structure. We plan to experiment with such optimizations in future work.

We ran an experiment involving single-processor executions of Leiserson and Schardl's parallel BFS algorithm (PBFS) [12]. The original PBFS uses a special-purpose bag data structure to manage the frontier of the graph traversal. During a given round, PBFS traverses its frontier in a divide-and-conquer fashion, using push and pop in the sequentialized leaves and split and concat in the divide and conquer stages, respectively. Their bag data structure is represented by a chunked binomial tree that bears some resemblance to our chunked representations. Despite the similarities, Leiserson and Schardl's structure provides access only to the front and supports only an approximate split-in-half operation. In our experiment, we consider the same chunk capacity as in the original PBFS paper, namely 128, and we applied to our data structure a few basic optimizations exploiting the fact that sequence order needs not be maintained —in particular, the back buffers become unnecessary. We see from the results table that our (bag-specialized) chunked data structures perform either better, or at worst a few percent slower, than the PBFS bag structure.

In the long version [1], we report on two additional experiments to more thoroughly evaluate performance in scenarious that mix push, pop, split and concatenate, comparing in particular against the STL rope data structure [13].

Table 1. Measurements of benchmark runs. All measurements were taken from our Intel machine. Each data point represents wall-clock time in seconds. For each data point in the table, we made five runs and took the mean. The amount of noise that we observed between runs of the same application was below 1%. All data points that are no more than 10% slower than the best time are displayed in boldface. Our grid 2D graph is a grid graph in two-dimensional space, where each vertex is connected to each of its four neighbors in two dimensions. We used number of vertices $n = 2$ billion and number of edges $m = 4$ billion. Our tree graph is a perfect binary tree of 2^{29} nodes. Our friendster graph is a social networking graph that has $n = 65$ million vertices and $m = 1.8$ billion edges [5]. For LIFO and DFS, we use the stack optimization and for PBFS we use the bag optimization as described in the long version [1], while for the other benchmarks we use the plain double-ended sequence-ordered chunk representation. For PBFS, we use linear-time split and concat for STL deque (only).

Experiment	Seq. length	Nb. repeat	PBFS bag	STL deque	Our chunked finger tree	Our bootstr. chunked
LIFO	10^3	10^6		**5.46**	6.40	6.99
	10^6	10^3		**9.15**	10.95	10.97
	10^9	10^0		**12.07**	**13.28**	13.47
FIFO	10^3	10^6		**5.51**	6.34	6.40
	10^6	10^3		**9.16**	10.96	10.52
	10^9	10^0		**12.32**	**13.53**	**13.31**
DFS on grid 2D				**4.84**	**5.17**	**5.27**
DFS on tree				**11.25**	15.53	15.46
DFS on friendster				**63.43**	**64.67**	**65.28**
BFS on grid 2D				**39.89**	**36.74**	**36.68**
BFS on tree				**15.23**	20.54	21.08
BFS on friendster				**72.84**	**72.76**	**72.68**
PBFS on grid 2D			**39.87**	**38.17**	**38.71**	**38.67**
PBFS on tree			**19.00**	75.53	**21.53**	**20.46**
PBFS on friendster			**117.11**	137.36	**117.45**	**117.04**

Our experiments show that our chunked data structures deliver excellent performance relative to the state-of-the-art data structures that we considered, even though each of these other data structures are highly tuned for a strictly narrower set of operations. Moreover, in contrast to the other state-of-the-art chunked data structures, ours come along with strong guarantees against worst case behavior. Furthermore, our benchmarks show promise for our chunked data structures to serve in roles that were previously not filled. On the one hand, for many sequential-programming applications, our data structures can be used in place of STL deque, and as a bonus, offer fast logarithmic-time split and concatenate operations. On the other, the PBFS application demonstrates potential of our chunked data structures in multicore applications as generic sequence containers and as splittable work-queue data structures in load-balancing algorithms.

6 Conclusion and Future Work

We presented algorithmic and implementation techniques for designing practically efficient sequence data structures that amortize expensive operations over a collection of items arranged as a chunk. We proved tight bounds by parameterizing our analysis by the cost of memory allocations, which, in our approach, correlate with expensive operations, and by counting such operations separately. We show that the proposed techniques perform well in practice. In future work, we plan to investigate the use of stronger invariants on consecutive chunks for increased space utilization, and consider persistent data structures.

Acknowledgements. This research is partially supported by the European Research Council under grant number ERC-2012-StG-308246 and the National Science Foundation under grant number CCF-1320563.

References

1. Acar, U.A., Charguéraud, A., Rainey, M.: Theory and practice of chunked sequences, `http://deepsea.inria.fr/chunkedseq` (full version)
2. Bernardy, J.-P.: The Haskell yi package, `http://hackage.haskell.org/package/yi-0.6.2.3/docs/src/Data-Rope.html`
3. Buchsbaum, A.L., Tarjan, R.E.: Confluently persistent deques via data-structural bootstrapping. J. Algorithms 18(3), 513–547 (1995)
4. Buchsbaum, A.L.: Data-structural bootstrapping and catenable deques. PhD thesis, Princeton University (1993)
5. Stanford Large Network Dataset Collection. Friendster graph, `http://snap.stanford.edu/data/com-Friendster.html`
6. Dietz, P.F.: Maintaining order in a linked list. In: STOC 1982, Baltimore, USA, pp. 122–127. ACM Press (May 1982)
7. Guibas, L.J., McCreight, E.M., Plass, M.F., Roberts, J.R.: A new representation for linear lists. In: STOC 1977, pp. 49–60. ACM, New York (1977)
8. Hinze, R., Paterson, R.: Finger trees: a simple general-purpose data structure. JFP 16(2), 197–218 (2006)
9. Kaplan, H., Tarjan, R.E.: Persistent lists with catenation via recursive slow-down. In: TOC 1995, pp. 93–102. ACM (1995)
10. Kaplan, H., Tarjan, R.E.: Purely functional representations of catenable sorted lists. In: STOC 1996, pp. 202–211. ACM, New York (1996)
11. Knuth, D.E.: The Art of Computer Programming: Sorting and Searching, 2nd edn., vol. 3, ch. 6, pp. 481–489. Addison-Wesley (1998)
12. Leiserson, C.E., Schardl, T.B.: A work-efficient parallel breadth-first search algorithm. In: SPAA 2010, pp. 303–314 (June 2010)
13. SGI. Stl rope, `http://www.sgi.com/tech/stl/Rope.html`
14. Stepanov, A., Lee, M.: The Standard Template Library, volume 1501. HP Laboratories (1995)

Convex Hulls under Uncertainty

Pankaj K. Agarwal[1], Sariel Har-Peled[2], Subhash Suri[3], Hakan Yıldız[3], and Wuzhou Zhang[1]

[1] Duke University, United States
[2] University of Illinois, Urbana-Champaign, United States
[3] University of California, Santa Barbara, United States

Abstract. We study the convex-hull problem in a probabilistic setting, motivated by the need to handle data uncertainty inherent in many applications, including sensor databases, location-based services and computer vision. In our framework, the uncertainty of each input point is described by a probability distribution over a finite number of possible locations including a *null* location to account for non-existence of the point. Our results include both exact and approximation algorithms for computing the probability of a query point lying inside the convex hull of the input, time-space tradeoffs for the membership queries, a connection between Tukey depth and membership queries, as well as a new notion of β-hull that may be a useful representation of uncertain hulls.

1 Introduction

The convex hull of a set of points is a fundamental structure in mathematics and computational geometry, with wide-ranging applications in computer graphics, image processing, pattern recognition, robotics, combinatorics, and statistics. Worst-case optimal as well as output-sensitive algorithms are known for computing the convex hull; see the survey [15] for an overview of known results.

In many applications, such as sensor databases, location-based services or computer vision, the location and sometimes even the existence of the data is uncertain, but statistical information can be used as a probability distribution guide for data. This raises the natural computational question: what is a robust and useful convex hull representation for such an uncertain input, and how well can we compute it? We explore this problem under two simple models in which both the location and the existence (presence) of each point is described probabilistically, and study basic questions such as what is the probability of a query point lying inside the convex hull, or what does the probability distribution of the convex hull over the space look like.

Uncertainty models. We focus on two models of uncertainty: unipoint and multipoint. In the *unipoint model*, each input point has a fixed location but it only exists probabilistically. Specifically, the input $\mathcal{P}$ is a set of pairs $\{(p_1, \gamma_1), \ldots, (p_n, \gamma_n)\}$ where each p_i is a point in $\mathbb{R}^d$ and each γ_i is a real number in the range $(0, 1]$ denoting the probability of p_i's existence. The existence probabilities of different points are independent; $P = \{p_1, \ldots, p_n\}$ denotes the set of sites in $\mathcal{P}$.

In the *multipoint model*, each point probabilistically exists at one of multiple possible sites. Specifically, $\mathcal{P}$ is a set of pairs $\{(P_1, \Gamma_1), \ldots, (P_m, \Gamma_m)\}$ where each P_i is a set of n_i points and each Γ_i is a set of n_i real values in the range $(0, 1]$. The set $P_i = \{p_i^1, \ldots, p_i^{n_i}\}$ describes the possible sites for the ith point of $\mathcal{P}$ and the set $\Gamma_i = \{\gamma_i^1, \ldots, \gamma_i^{n_i}\}$ describes the associated probability distribution. The probabilities γ_i^j correspond to disjoint events and therefore sum to at most 1. By allowing the sum to be less than one, this model also accounts for the possibility of the point not existing (i.e. the *null* location)—thus, the multipoint model generalizes the unipoint model. In the multipoint model, $P = \bigcup_{i=1}^m P_i$ refers to the set of all sites and $n = |P|$.

Our results. The main results of our paper can be summarized as follows.

(A) We show (in Section 2) that the membership probability of a query point $q \in \mathbb{R}^d$, namely, the probability of q being inside the convex hull of $\mathcal{P}$, can be computed in $O(n \log n)$ time for $d = 2$. For $d \geq 3$, assuming the input and the query point are in general position, the membership probability can be computed in $O(n^d)$ time. The results hold for both unipoint and multipoint models.

(B) Next we describe two algorithms (in Section 3) to preprocess $\mathcal{P}$ into a data structure so that for a query point its membership probability in $\mathcal{P}$ can be answered quickly. The first algorithm constructs a *probability map* $\mathbb{M}(\mathcal{P})$, a partition of $\mathbb{R}^d$ into convex cells, so that all points in a single cell have the same membership probability. We show that $\mathbb{M}(\mathcal{P})$ has size $\Theta(n^{d^2})$, and for $d = 2$ it can be computed in optimal $O(n^4)$ time. The second one is a sampling-based Monte Carlo algorithm for constructing a near-linear-size data structure that can approximate the membership probability with high likelihood in sublinear time for any fixed dimension.

(C) We show (in Section 4) a connection between the membership probability and the Tukey depth, which can be used to approximate cells of high membership probabilities. For $d = 2$, this relationship also leads to an efficient data structure.

(D) Finally, we introduce the notion of *β-hull* (in Section 5) as another approximate representation for uncertain convex hulls in the multipoint model: a convex set C is called *β-dense* for $\mathcal{P}$, for $\beta \in [0, 1]$, if C contains at least β fraction of each uncertain point. The β-hull of $\mathcal{P}$ is the intersection of all β-dense sets for $\mathcal{P}$. We show that for $d = 2$, the β-hull of $\mathcal{P}$ can be computed in $O(n \log^3 n)$ time.

Because of lack of space, many technical details and proofs are omitted from this version and can be found in the full version [3].

Related work. There is extensive and ongoing research in the database community on uncertain data; see [7] for a survey. In the computational geometry community, the early work relied on deterministic models for uncertainty (see e.g. [11]), but more recently probabilistic models of uncertainty, which are closer to the models used in statistics and machine learning, have been explored [1, 2, 9, 10, 14, 16]. The convex-hull problem over uncertain data has received some attention very recently. Suri *et al.* [16] showed that the problem of computing the most likely convex hull of a point set in the multipoint model is NP-hard. Even in the unipoint model, the problem is NP-hard for $d \geq 3$. They also presented an $O(n^3)$-time algorithm for computing the most likely convex hull under the unipoint model in $\mathbb{R}^2$. Zhao *et al.* [17] investigated the problem of computing the probability of each uncertain point lying on the convex hull, where they aimed to return the set of (uncertain) input points whose probabilities of being on the convex hull are at least some threshold. Jørgensen *et al.* [8] showed that the distribution of properties, such as areas or perimeters, of the convex hull of $\mathcal{P}$ may have $\Omega(\Pi_{i=1}^m n_i)$ complexity.

2 Computing the Membership Probability

For simplicity, we describe our algorithms under the unipoint model, and then discuss their extension to the multipoint model. We begin with the 2D case.

2.1 The Two-Dimensional Case

Let $\mathcal{P} = \{(p_1, \gamma_1), \ldots, (p_n, \gamma_n)\}$ be a set of n uncertain points in $\mathbb{R}^2$ under the unipoint model. Recall that $P = \{p_1, \ldots, p_n\}$ is the set of all sites of $\mathcal{P}$. For simplicity, we make the general position assumption on the input, namely, that all coordinates are distinct and no three sites are collinear. A subset $B \subseteq P$ is the outcome of a probabilistic experiment with probability $\gamma(B) = \prod_{p_i \in B} \gamma_i \times \prod_{p_i \notin B} \overline{\gamma_i}$, where $\overline{\gamma_i}$ is the complementary probability $1 - \gamma_i$. By definition, for a point q, the *probability* of q to lie in the convex-hull of B is $\mu(q) = \sum_{B \subseteq P \mid q \in \textsc{ch}(B)} \gamma(B)$, where $\textsc{ch}(B)$ is the convex hull of B. This unfortunately involves an exponential number of terms. However, observe that for a subset $B \subseteq P$, the point q is *outside* $\textsc{ch}(B)$, if and only if q is a vertex of the convex hull $\textsc{ch}(B \cup \{q\})$. So, let $C = \textsc{ch}(B \cup \{q\})$, and V be the set of vertices of C. Then $\mu(q) = 1 - \Pr[\, q \in V\,]$.

If $B = \emptyset$, then clearly $C = \{q\}$ and $q \in V$. Otherwise, $|V| \geq 2$ and $q \in V$ implies that q is an endpoint of exactly two edges on the boundary of C.[1] In this case, the first edge following q in the counter-clockwise order of C is called the *witness edge* of q being in V. Thus, $q \in V$ if and only if $B = \emptyset$ or (exclusively) B has a witness edge, i.e.,

$$\Pr\Big[\, q \in V \,\Big] = \Pr\Big[\, B = \emptyset \,\Big] + \sum_{i=1}^{n} \Pr\Big[\, qp_i \text{ is the witness edge of } q \in V \,\Big].$$

The first term can be computed in linear time. To compute the ith term in the summation, we observe that qp_i is the witness edge of B if and only if $p_i \in B$ and B contains no sites to the right of the oriented line spanned by the vector

[1] If B consists of a single site p_i, then C is the line segment qp_i. In this case, we consider the boundary of C to be a cycle formed by two edges: one going from q to p_i, and one going from p_i back to q.

$\overrightarrow{qp_i}$, which occurs with probability $\gamma_i \cdot \prod_{p_j \in G_i} \overline{\gamma_j}$, where G_i is the set of sites to the right of $\overrightarrow{qp_i}$. This expression can be computed in $O(n)$ time. It follows that $1 - \mu(q)$, and therefore $\mu(q)$, can be computed in $O(n^2)$ time. The computation time can be improved to $O(n \log n)$ as described in the following paragraph.

Improving the running time. The main idea is to compute the witness edge probabilities in radial order around q. We sort all sites in counter-clockwise order around q. Without loss of generality, assume that the circular sequence $p_1, \ldots, p_n$ is the resulting order. We first compute, in $O(n)$ time, the probability that qp_1 is the witness edge. Then, for increasing values of i from 2 to n, we compute, in $O(1)$ amortized time, the probability that qp_i is the witness edge by updating the probability for qp_{i-1}. In particular, let W_i denote the set of sites in the open wedge bounded by the vectors $\overrightarrow{qp_{i-1}}$ and $\overrightarrow{qp_i}$. Notice that $G_i = G_{i-1} \cup \{p_{i-1}\} \setminus W_i$. It follows that the probability for qp_i can be computed by multiplying the probability for qp_{i-1} with $\frac{\gamma_i}{\gamma_{i-1}} \times \frac{\overline{\gamma_{i-1}}}{\prod_{p_j \in W_i} \overline{\gamma_j}}$. The amortized cost of a single update is $O(1)$ because the total number of multiplications in all the updates is at most $4n$. (Each site affects at most 4 updates.) Finally, notice that we can easily keep track of the set W_i during our radial sweep, as changes to this set follow the same radial order.

Theorem 1. *Given a set of n uncertain points in $\mathbb{R}^2$ under the unipoint model, the membership probability of a query point q can be computed in $O(n \log n)$ time.*

2.2 The *d*-Dimensional Case

The difficulty in extending the above to higher dimensions is an appropriate generalization of witness edges, which allow us to implicitly sum over exponentially many outcomes without over-counting. Our algorithm requires that all sites, including the query point q, are in general position in the following sense: for $2 \le k \le d$, the projection of no $k+1$ points of $P \cup \{q\}$ on a subspace spanned by any subset of k coordinates lies on a $(k-1)$-hyperplane.

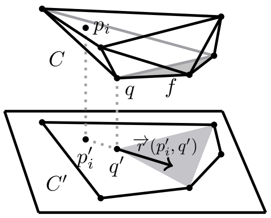
Let B be an outcome, $C = \text{CH}(B \cup \{q\})$ its convex hull, and V the vertices of C. Let $\lambda(B \cup \{q\})$ denote the point with the lowest x_d-coordinate in $B \cup \{q\}$. Clearly, if q is $\lambda(B \cup \{q\})$ then $q \in V$; otherwise, we condition the probability based on which point among B is $\lambda(B \cup \{q\})$. Therefore, we can write

$$\Pr\Big[q \in V\Big] = \Pr\Big[q = \lambda(B \cup \{q\})\Big] + \sum_{1 \le i \le n} \Pr\Big[p_i = \lambda(B \cup \{q\}) \ \wedge \ q \in V\Big].$$

It is easy to compute the first term. We show below how to compute each term of the summation in $O(n^{d-1})$ time, which gives the desired bound of $O(n^d)$.

Consider an outcome B. Let p_i be an arbitrary point in B. We use p_i as a reference point known to be contained in the hull $C = \text{CH}(B \cup \{q\})$. Let B', p'_i and q' denote the projections of B, p_i and q respectively on the hyperplane $x_d = 0$, which we identify with $\mathbb{R}^{d-1}$. Let us define $C' = \text{CH}(B' \cup \{q'\}) \subset \mathbb{R}^{d-1}$, and let V' be the vertices of C'.

Let $\overrightarrow{r}(p'_i, q')$ denote the open ray emanating from q' in the direction of the vector $\overrightarrow{p'_i q'}$ (that is, this ray is moving "away" from p'_i). A facet f of C is a p_i*-escaping* facet for q, if q is a vertex of f and the projection of f on $\mathbb{R}^{d-1}$ intersects $\overrightarrow{r}(p'_i, q')$. See the figure on the right. The following lemma is key to our algorithm. The points of C projected into $\partial C'$ form the *silhouette* of C.



Lemma 1. *(A) If $q' \in V'$ then q is a silhouette vertex of C and vice versa.*

(B) q has at most one p_i-escaping facet on C.

(C) The point q is a non-silhouette vertex of the convex-hull C if and only if q has a (single) p_i-escaping facet on C.

Given a subset of sites $P_\alpha \subseteq P \setminus \{p_i\}$ of size $(d-1)$, define $f(P_\alpha)$ to be the $(d-1)$-dimensional simplex $\text{CH}(P_\alpha \cup \{q\})$. Since $p_i = \lambda(B \cup \{q\})$ implies $p_i \in B$, we can use Lemma 1 to decompose the ith term as follows:

$$\Pr\Big[p_i = \lambda(B \cup \{q\}) \ \wedge \ q \in V\Big] = \Pr\Big[p_i = \lambda(B \cup \{q\}) \ \wedge \ q' \in V'\Big]$$
$$+ \sum_{\substack{P_\alpha \subseteq P \setminus \{p_i\} \\ |P_\alpha| = (d-1) \\ f(P_\alpha) \text{ is } p_i\text{-escaping for } q}} \Pr\Big[p_i = \lambda(B \cup \{q\}) \ \wedge \ f(P_\alpha) \text{ is a facet of } C\Big].$$

The first term is an instance of the same problem in $(d-1)$ dimensions (for the point q' and the projection of P), and thus is computed recursively. For the second term, we compute the probability that $f(P_\alpha)$ is a facet of C as follows. Let $G_1 \subseteq P$ be the subset of sites which are on the other side of the hyperplane supporting $f(P_\alpha)$ with respect to p_i. Let $G_2 \subseteq P$ be the subset of sites that are below p_i along the x_d-axis. Clearly, $f(P_\alpha)$ is a facet of C (and $p_i = \lambda(B \cup \{q\})$) if and only if all points in P_α and p_i exist in B, and all points in $G_1 \cup G_2$ are absent from B. The corresponding probability can be written as $\gamma_i \times \prod_{p_j \in P_\alpha} \gamma_j \times \prod_{p_j \in G_1 \cup G_2} \overline{\gamma_j}$. This formula is valid only if $P_\alpha \cap G_2 = \emptyset$ and p_i has a lower x_d-coordinate than q; otherwise we set the probability to zero. This expression can be computed in linear time, and the whole summation term can be computed in $O(n^d)$ time. Then, by induction, the computation of the ith term takes $O(n^d)$ time. Notice that the base case of our induction requires computing the probability $\Pr\Big[p_i = \lambda(B \cup \{q\}) \ \wedge \ q^{(d-2)} \in V^{(d-2)}\Big]$ (where ${}^{(d-2)}$ indicates a projection to $\mathbb{R}^2$). Computing this probability is essentially a two-dimensional membership probability problem on q and P, but is conditioned on the existence of p_i and the non-existence of all sites below p_i along dth axis. Our two dimensional algorithm can be easily adapted to solve this variation in $O(n \log n)$ time as well. Finally, we can improve the computation time for the ith term to $O(n^{d-1})$ by considering the facets $f(P_\alpha)$ in radial order. See the full version of the paper [3] for details.

Remark. The degeneracy of the input is easy to handle in two dimensions, but creates some technical difficulties in higher dimensions that we are currently investigating.

Theorem 2. *Let $\mathcal{P}$ be an uncertain set of n points in the unipoint model in $\mathbb{R}^d$ and q be a point. If the input sites and q are in general position, then one can compute the membership probability of q in $O(n^d)$ time, using linear space.*

Extension to the multipoint model. The algorithm extends to the multipoint model easily by modifying the computation of the probability for an edge or facet. See the full version of the paper [3] for details.

Theorem 3. *Given an uncertain set $\mathcal{P}$ of n points in the multipoint model in $\mathbb{R}^d$ and a point $q \in \mathbb{R}^d$, we can compute the membership probability of q in $O(n \log n)$ time for $d = 2$, and in $O(n^d)$ time for $d \geq 3$ if input sites and q are in general position.*

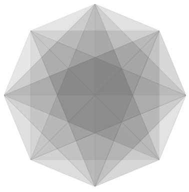
3 Membership Queries

We describe two algorithms – one deterministic and one Monte Carlo – for preprocessing a set of uncertain points for efficient membership-probability queries.

Probability map. The *probability map* $\mathbb{M}(\mathcal{P})$ is the subdivision of $\mathbb{R}^d$ into maximal connected regions so that $\mu(q)$ is the same for all query points q in a region. The following lemma gives a tight bound on the size of $\mathbb{M}(\mathcal{P})$.

Lemma 2. *The worst-case complexity of the probability map of a set of uncertain points in $\mathbb{R}^d$ is $\Theta(n^{d^2})$, under both the unipoint and the multipoint model, where n is the total number of sites in the input.*

Proof. We prove the result for the unipoint model, as the extension to the multipoint model is straightforward. For the upper bound, consider the set H of $O(n^d)$ hyperplanes formed by all d-tuples of points in $\mathcal{P}$. In the arrangement $\mathcal{A}(H)$ formed by these planes, each (open) cell has the same value of $\mu(q)$. This arrangement, which is a refinement of $\mathbb{M}(\mathcal{P})$, has size $O((n^d)^d) = O(n^{d^2})$, establishing the upper bound.



For the lower bound, consider the problem in two dimensions; extension to higher dimensions is straightforward. We choose the sites to be the vertices $p_1, \dots, p_n$ of a regular n-gon, where each site exists with probability γ, $0 < \gamma < 1$. See the figure on the right. Consider the arrangement $\mathcal{A}$ formed by the line segments $p_i p_j$, $1 \leq i < j \leq n$, and treat each face as relatively open. If $\mu(f)$ denotes the membership probability for a face f of $\mathcal{A}$, then for any two faces f_1 and f_2 of $\mathcal{A}$, where f_1 bounds f_2 (i.e., $f_1 \subset \partial f_2$), we have $\mu(f_1) \geq \mu(f_2)$, and $\mu(f_1) > \mu(f_2)$ if $\gamma < 1$. Thus, the size of the arrangement $\mathcal{A}$ is also a lower bound on the complexity of $\mathbb{M}(\mathcal{P})$. This proves that the worst-case complexity of $\mathbb{M}(\mathcal{P})$ in $\mathbb{R}^d$ is $\Theta(n^{d^2})$. $\square$

We can preprocess this arrangement into a point-location data structure, giving us the following result for $d = 2$.

Theorem 4. *Let $\mathcal{P}$ be a set of uncertain points in $\mathbb{R}^2$, with a total of n sites. $\mathcal{P}$ can be preprocessed in $O(n^4)$ time into a data structure of size $O(n^4)$ so that for any point $q \in \mathbb{R}^d$, $\mu(q)$ can be computed in $O(\log n)$ time.*

See the full version of the paper [3] for details.

Remark. For $d \geq 3$, due to our general position assumption, we can compute the membership probability only for d-faces of $\mathbb{M}(\mathcal{P})$, and not for the lower-dimensional faces. In that case, by utilizing a point-location technique in [5], one can build a structure that can report the membership probability of a query point (inside a d-face) in $O(\log n)$ time, with a preprocessing cost of $O(n^{d^2+d})$.

Monte Carlo algorithm. The size of the probability map may be prohibitive even for $d = 2$, so we describe a simple, space-efficient Monte Carlo approach for quickly approximating the membership probability, within absolute error. Fix a parameter $s > 1$, to be specified later. The preprocessing consists of s rounds, where the algorithm creates an outcome A_j of $\mathcal{P}$ in each round j. Each A_j is preprocessed into a data structure so that for a query point $q \in \mathbb{R}^d$, we can determine whether $q \in \text{CH}(A_j)$.

For $d \leq 3$, we can build each $\text{CH}(A_j)$ explicitly and use linear-size point-location structures with $O(\log n)$ query time. This leads to total preprocessing time $O(sn \log n)$ and space $O(sn)$. For $d \geq 4$, we use the data structure in [13] for determining whether $q \in A_j$, for all $1 \leq j \leq s$. For a parameter t such that $n \leq t \leq n^{\lfloor d/2 \rfloor}$ and for any constant $\sigma > 0$, using $O(st^{1+\sigma})$ space and preprocessing, it can compute in $O(\frac{sn}{t^{1/\lfloor d/2 \rfloor}} \log^{2d+1} n)$ time whether $q \in \text{CH}(A_j)$ for every j.

Given a query point $q \in \mathbb{R}^d$, we check for membership in $\text{CH}(A_j)$, for every $j \leq s$. If q lies in k of them, we return $\widehat{\mu}(q) = k/s$ as our estimate of $\mu(q)$. Thus, the query time is $O(\frac{sn}{t^{1/\lfloor d/2 \rfloor}} \log^{2d+1} n)$ for $d \geq 4$, $O(s \log n)$ for $d = 3$, and $O(\log n + s)$ for $d = 2$ (using fractional cascading).

It remains to determine the value of s so that $|\mu(q) - \widehat{\mu}(q)| \leq \varepsilon$ for all queries q, with probability at least $1-\delta$. For a fixed q and outcome A_j, let X_i be the random indicator variable, which is 1 if $q \in \text{CH}(A_j)$ and 0 otherwise. Since $\mathsf{E}[X_i] = \mu(q)$ and $X_i \in \{0, 1\}$, using a Chernoff-Hoeffding bound on $\widehat{\mu}(q) = k/s = (1/s) \sum_i X_i$, we observe that $\Pr[\, |\widehat{\mu}(q) - \mu(q)| \geq \varepsilon \,] \leq 2 \exp(-2\varepsilon^2 s) \leq \delta'$. By Lemma 2, we need to consider $O(n^{d^2})$ distinct queries. If we set $1/\delta' = O(n^{d^2}/\delta)$ and $s = O((1/\varepsilon^2) \log(n/\delta))$, we obtain the following theorem.

Theorem 5. *Let $\mathcal{P}$ be a set of uncertain points in $\mathbb{R}^d$ under the multipoint model with a total of n sites, and let $\varepsilon, \delta \in (0, 1)$ be parameters. For $d \geq 4$, $\mathcal{P}$ can be preprocessed, for any constant $\sigma > 0$, in $O((t^{1+\sigma}/\varepsilon^2) \log \frac{n}{\delta})$ time, into a data structure of size $O((t^{1+\sigma}/\varepsilon^2) \log \frac{n}{\delta})$, so that with probability at least $1-\delta$, for any query point $q \in \mathbb{R}^2$, $\widehat{\mu}(q)$ satisfying $|\mu(q) - \widehat{\mu}(q)| \leq \varepsilon$ and $\widehat{\mu}(q) > 0$ can be returned in $O(\frac{n}{t^{1/\lfloor d/2 \rfloor} \varepsilon^2} \log \frac{n}{\delta} \log^{2d+1} n)$ time, where t is a parameter and $n \leq t \leq n^{\lfloor d/2 \rfloor}$.*

For $d \leq 3$, the preprocessing time and space are $O(\frac{n}{\varepsilon^2}\log\log\frac{n}{\delta}\log n)$ and $O(\frac{n}{\varepsilon^2}\log\frac{n}{\delta})$, respectively. The query time is $O(\frac{1}{\varepsilon^2}\log(\frac{n}{\delta})\log n)$ (resp. $O(\frac{1}{\varepsilon^2}\log\frac{n}{\delta})$) for $d = 3$ (resp. $d = 2$).

4 Tukey Depth and Convex Hull

The membership probability is neither a convex nor a continuous function, as suggested by the example in the proof of Lemma 2. In this section, we establish a helpful structural property of this function, intuitively showing that the probability stabilizes once we go deep enough into the "region". Specifically, we show a connection between the Tukey depth of a point q with its membership probability; in two dimensions, this also results in an efficient data structure for approximating $\mu(q)$ quickly within a small absolute error.

Estimating $\mu(q)$**.** Let Q be a set of weighted points in $\mathbb{R}^d$. For a subset $A \subseteq Q$, let $w(A)$ be the total weight of points in A. Then the *Tukey depth* of a point $q \in \mathbb{R}^d$ with respect to Q, denoted by $\tau(q, Q)$, is $\min w(Q \cap H)$ where the minimum is taken over all halfspaces H that contain q.[2] If Q is obvious from the context, we use $\tau(q)$ to denote $\tau(q, Q)$. Before bounding $\mu(q)$ in terms of $\tau(q, Q)$, we prove the following lemma.

Lemma 3. *Let Q be a finite set of points in $\mathbb{R}^d$. For any $p \in \mathbb{R}^d$, there is a set $\mathcal{S} = \{S_1, \ldots, S_T\}$ of d-simplices formed by Q such that (i) each S_i contains p in its interior; (ii) no pair of them shares a vertex; and (iii) $T \geq \lceil \tau(p, Q)/d \rceil$.*

We now use Lemma 3 to bound $\mu(p)$ in terms of $\tau(p, P)$.

Theorem 6. *Let $\mathcal{P}$ be a set of n uncertain points in the uniform unipoint model, that is, each point is chosen with the same probability $\gamma > 0$. Let P be the set of sites in $\mathcal{P}$. There is a constant $c > 0$ such that for any point $p \in \mathbb{R}^d$ with $\tau(p, P) = t$, we have $(1-\gamma)^t \leq 1 - \mu(p) \leq d\exp\Big(-\frac{\gamma t}{cd^2}\Big)$.*

Proof. For the first inequality, fix a closed halfspace H that contains t points of P. If none of these t points is chosen then p does not appear in the convex hull of the outcome, so $1 - \mu(p) \geq (1-\gamma)^t$.

Next, let $\mathcal{S}$ be the set of simplices of Lemma 3, and let V be its set of vertices, where $T \geq \lceil t/d \rceil$. Let $n' = |V| = (d+1)T$. Set $\varepsilon = \frac{1}{d+1}$. A random subset of V of size $O(\frac{d}{\varepsilon}\log\frac{1}{\varepsilon\delta}) = O(d^2\log\frac{d}{\delta})$ is an ε-net for halfspaces, with probability at least $1 - \delta$.

In particular, any halfspace passing through p, contains at least T points of V. That is, all these halfspaces are ε-heavy and would be stabbed by an ε-net. Now, if we pick each point of V with probability γ, it is not hard to argue that the resulting sample R is an ε-net[3]. Indeed, the expected size (and

[2] If the points in Q are unweighted, then $\tau(q, Q)$ is simply the minimum number of points that lie in a closed halfspace that contains q.

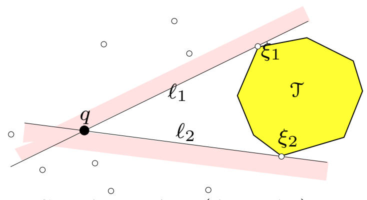
in with sufficiently large probability) of $R \cap V$ is $n'' = n'\gamma = (d+1)T\gamma \geq t\gamma$. As such, for some constant c, we need the minimal value of δ such that the inequality $t\gamma \geq cd^2 \ln \frac{d}{\delta}$ holds, which is equivalent to $\exp\left(\frac{t\gamma}{cd^2}\right) \geq \frac{d}{\delta}$. This in turn is equivalent to $\delta \geq d \exp\left(-\frac{t\gamma}{cd^2}\right)$. Thus, we set $\delta = d \exp\left(-\frac{t\gamma}{cd^2}\right)$.

Now, with probability at least $1-\delta$, for a point p in $\mathbb{R}^d$ with Tukey depth at least t, we have that p is in the convex-hull of the sample. $\square$

Theorem 6 can be extended to the case when each point p_i of $\mathcal{P}$ is chosen with different probability, say, γ_i. In order to apply Theorem 6, we convert $\mathcal{P}$ to a multiset $\mathcal{Q}$, as follows. We choose a parameter $\eta = \frac{\delta}{10n}$. For each point $p_i \in \mathcal{P}$, we make $w_i = \left\lceil \frac{\ln(1-\gamma_i)}{\ln(1-\eta)} \right\rceil$ copies of p_i, each of which is selected with probability η. We can apply Theorem 6 to $\mathcal{Q}$ and show that if $\tau(q,\mathcal{Q}) \geq \frac{d^2}{\eta}\ln(2d/\delta)$, then $\mu(q,\mathcal{Q}) \geq (1-\delta/2)$. Omitting the further details, we conclude the following.

Corollary 1. *Let $\mathcal{P} = \{(p_1,\gamma_1),\ldots,(p_n,\gamma_n)\}$ be a set of n uncertain points in $\mathbb{R}^d$ under the unipoint model. For $1 \leq i \leq n$, set $w_i = \left\lceil \frac{\ln(1-\gamma_i)}{\ln(1-\delta/10n)} \right\rceil$ to be the weight of point p_i. If the (weighted) Tukey depth of a point $q \in \mathbb{R}^d$ in $\{p_1,\ldots,p_n\}$ is at least $\frac{10d^2n}{\delta}\ln(2d/\delta)$, then $\mu(q,\mathcal{P}) \geq 1-\delta$.*

Data structure. Let $\mathcal{P}$ be a set of points in the uniform unipoint model in $\mathbb{R}^2$, i.e., each point appears with probability γ. We now describe a data structure to estimate $\mu(q)$ for a query point $q \in \mathbb{R}^2$, within additive error $1/n$. We fix a parameter $t_0 = \frac{c}{\gamma}\ln n$ for some constant $c > 0$. Let $\mathcal{T} = \left\{x \in \mathbb{R}^2 \mid \tau(x,\mathcal{P}) \geq t_0\right\}$ be the set of all points whose Tukey depth in P is at least t_0. $\mathcal{T}$ is a convex polygon with $O(n)$ vertices [12]. By Theorem 6, $\mu(q) \geq 1 - 1/n^2$ for all points $q \in \mathcal{T}$, provided that the constant c is chosen appropriately. We also preprocess P for halfspace range reporting queries [6]. $\mathcal{T}$ can be computed in time $O(n\log^3 n)$ [12], and constructing the half-plane range reporting data structure takes $O(n\log n)$ time [6]. So the total preprocessing time is $O(n\log^3 n)$, and the size of the data structure is linear.



A query is answered as follows. Given a query point $q \in \mathbb{R}^2$, we first test in $O(\log n)$ time whether $q \in \mathcal{T}$. If the answer is yes, we simply return 1 as $\mu(q)$. If not, we compute in $O(\log n)$ time the two tangents ℓ_1, ℓ_2 of $\mathcal{T}$ from q. For $i = 1,2$, let $\xi_i = \ell_i \cap \mathcal{T}$, and let ℓ_i^- be the half-plane bounded by ℓ_i that does not contain $\mathcal{T}$. Set $\mathcal{P}_q = \mathcal{P} \cap (\ell_1^- \cup \ell_2^-)$ and $n_q = |\mathcal{P}_q|$. Let R_q be the subset of $\mathcal{P}_q$ by choosing each point with probability γ.

By querying the half-plane range reporting data structure with each of these two tangent lines, we compute the set $\mathcal{P}_q$ in time $O(\log n + n_q)$. Let $\omega_q = \Pr[\, q \notin \text{CH}(R_q \cup \mathcal{T})\,]$. We compute ω_q, in $(n_q \log n_q)$ time, by adapting the algorithm for computing $\mu(q)$ described in Section 2.

[3] The standard argument uses slightly different sampling, but this is a minor technicality, and it is not hard to prove the ε-net theorem with this modified sampling model.

The correctness and efficiency of the algorithm follow from the following lemma, whose proof is omitted from this version.

Lemma 4. *For any point* $q \notin \mathcal{T}$, *(i)* $|\Pr[\, q \in \text{CH}(R_q \cup \mathcal{T})\,] - \mu(q)| \leq 1/n$; *(ii)* $n_q \leq 4t_0 = O(\gamma^{-1} \log n)$.

By Lemma 4, $n_q = O(\gamma^{-1} \log n)$, so the query takes $O(\gamma^{-1} \log(n) \log\log n)$ time. We thus obtain the following.

Theorem 7. *Let* $\mathcal{P}$ *be a set of* n *uncertain points in* $\mathbb{R}^2$ *in the unipoint model, where each point appears with probability* γ. $\mathcal{P}$ *can be preprocessed in* $O(n \log^3 n)$ *time into a linear-size data structure that, for any point* $q \in \mathbb{R}^2$, *returns a value* $\widetilde{\mu}(q)$ *in* $O(\gamma^{-1} \log(n) \log\log n)$ *time such that* $|\widetilde{\mu}(q) - \mu(q)\,| \leq 1/n$.

5 β-Hull

In this section, we consider the multipoint model, i.e., $\mathcal{P}$ is a set of m uncertain point defined by the pairs $\{(P_1, \Gamma_1), \dots, (P_m, \Gamma_m)\}$. A convex set $C \subseteq \mathbb{R}^2$ is called *β-dense* with respect to $\mathcal{P}$ if it contains β-fraction of each (P_i, Γ_i), i.e., $\sum_{p_i^j} \gamma_i^j \geq \beta$ for all $i \leq m$. The *β-hull* of $\mathcal{P}$, denoted by $\text{CH}_\beta(\mathcal{P})$, is the intersection of all convex β-dense sets with respect to $\mathcal{P}$. Note that for $m = 1$, $\text{CH}_\beta(\mathcal{P})$ is the set of points whose Tukey depth is at least $1 - \beta$. We first prove an $O(n)$ upper bound on the complexity of $\text{CH}_\beta(\mathcal{P})$ and then describe an algorithm for computing it.

Theorem 8. *Let* $\mathcal{P} = \{(P_1, \Gamma_1), \dots, (P_m, \Gamma_m)\}$ *be a set of* m *uncertain points in* $\mathbb{R}^2$ *under the multipoint model with* $P = \bigcup_{i=1}^m P_i$ *and* $|P| = n$. *For any* $\beta \in [0, 1]$, $\text{CH}_\beta(\mathcal{P})$ *has* $O(n)$ *vertices.*

Proof. We call a convex β-dense set C *minimal* if there is no convex β-dense set C' such that $C' \subset C$. A minimal convex β-dense set C is the convex hull of $P \cap C$. Therefore C is a convex polygon whose vertices are a subset of P. Obviously $\text{CH}_\beta(\mathcal{P})$ is the intersection of minimal convex β-dense sets. Therefore each edge of $\text{CH}_\beta(\mathcal{P})$ lies on a line passing through a pair of points of P, i.e., $\text{CH}_\beta(\mathcal{P})$ is the intersection of a set H of halfplanes, each bounded by a line passing through a pair of points of P. Next we argue that $|H| \leq 2n$.

Fix a point $p \in P$. We claim that H contains at most two halfplanes whose bounding lines pass through p. Indeed if $p \in \text{int}(\text{CH}_\beta(\mathcal{P}))$, then no bounding line of H passes through p; if $p \in \partial(\text{CH}_\beta(\mathcal{P}))$, then at most two bounding lines of H pass through p; and if $p \notin \text{CH}_\beta(\mathcal{P})$, then there are two tangents to $\text{CH}_\beta(\mathcal{P})$ from p. Hence at most two bounding lines of H pass through p, as claimed. $\square$

Algorithm. We describe the algorithm for computing the upper boundary $\mathcal{U}$ of $\text{CH}_\beta(\mathcal{P})$. The lower boundary of $\text{CH}_\beta(\mathcal{P})$ can be computed analogously. It will be easier to compute $\mathcal{U}$ in the dual plane. Let $\mathcal{U}^*$ denote the dual of $\mathcal{U}$. We call a line ℓ passing through a point $p \in P_i$ *β-tangent* of P_i at p if one of the open half-planes bounded by ℓ contains less than β-fraction of points of P_i but the corresponding closed half-plane contains at least β-fraction of points.

Recall that the dual of a point $p = (a, b)$ is the line $p^* : y = ax - b$, and the dual of a line $\ell : y = mx + c$ is the point $\ell^* = (m, -c)$. The point p lies above/below/on the line ℓ if and only if the dual point ℓ^* lies above/below/on the dual line p^*. Set $P_i^* = \left\{p_i^{j*} \mid p_i^j \in P_i\right\}$ and $P^* = \bigcup_{i=1}^m P_i^*$. For a point $q \in \mathbb{R}^2$ and for $i \leq m$, let $\kappa(q, i) = \sum \gamma_i^j$ where the summation is taken over all points $p_i^j \in P_i$ such that q lies below the dual line p_i^{j*}. We define the β-level Λ_i of P_i^* to be the upper boundary of the region $\left\{q \in \mathbb{R}^2 \mid \kappa(q, i) \geq \beta\right\}$. Λ_i is an x-monotone polygonal chain composed of the edges of the arrangement $\mathcal{A}(P_i^*)$; the dual line of a point on Λ_i is a β-tangent line of P_i. Let Λ be the lower envelope of $\Lambda_1, \ldots, \Lambda_m$.

Let ℓ be the line supporting an edge of $\mathcal{U}$. It can be proved that the dual point ℓ^* is a vertex of Λ. Next, let q be a vertex of $\mathcal{U}$, then q cannot lie above any β-tangent line of any P_i, which implies that the dual line q^* passes through a pair of vertices of Λ and does not lie below any vertex of Λ. Hence, each vertex of $\mathcal{U}$ corresponds to an edge of the upper boundary of the convex hull of Λ. This observation suggests that $\mathcal{U}^*$ can be computed by adapting an algorithm for computing the convex hull of a level in an arrangement of lines [4, 12]. We begin by describing a simple procedure, which will be used as a subroutine in the overall algorithm.

Lemma 5. *Given a line ℓ, the intersection points of ℓ and Λ can be computed in $O(n \log n)$ time.*

Proof. We sort the intersections of the lines of P^* with ℓ. Let $\langle q_1, \ldots, q_u\rangle$, $u \leq n$, be the sequence of these intersection points. For every $i \leq m$, $\kappa(q_1, i)$ can be computed in a total of $O(n)$ time. Given $\{\kappa(q_{j-1}, i) \mid 1 \leq i \leq m\}$, $\{\kappa(q_j, i) \mid 1 \leq i \leq m\}$ can be computed in $O(1)$ time. A point $q_j \in \Lambda$ if $q_j \in \Lambda_i$ for some i and lies below Λ_i' for all other i'. This completes the proof of the lemma. □

The following two procedures can be developed by plugging Lemma 5 into the parametric-search technique [4, 12].

(A) Given a point q, determine whether q lies above $\mathcal{U}^*$ or return the tangent lines of $\mathcal{U}^*$ from q. This can be done in $O(n \log^2 n)$ time.

(B) Given a line ℓ, compute the edges of $\mathcal{U}^*$ that intersect ℓ, in $O(n \log^3 n)$ time. (Procedure (B) uses (A) and parametric search.)

Given (B), we can now compute $\mathcal{U}^*$ as follows. We fix a parameter $r > 1$ and compute a $(1/r)$-cutting[4] $\Xi = \{\Delta_1, \ldots, \Delta_u\}$, where $u = O(r^2)$. For each Δ_i, we do the following. Using (B) we compute the edges of $\mathcal{U}^*$ that intersect $\partial\Delta_i$. We can then deduce whether Δ_i contains any vertex of $\mathcal{U}^*$. If the answer is yes, we solve the problem recursively in Δ_i with the subset of lines of P^* that cross Δ_i. We omit the details from here and conclude the following.

[4] A $(1/r)$-*cutting* of P^* is a triangulation Ξ of $\mathbb{R}^2$ such that each triangle of Ξ crosses at most n/r lines of P^*.

Theorem 9. *Given a set $\mathcal{P}$ of uncertain points in $\mathbb{R}^2$ under the multipoint model with a total of n sites, and a parameter $\beta \in [0, 1]$, the β-hull of $\mathcal{P}$ can be computed in $O(n \log^3 n)$ time.*

Acknowledgments. P. Agarwal and W. Zhang are supported by NSF under grants CCF-09-40671, CCF-10-12254, and CCF-11-61359, by ARO grants W911NF-07-1-0376 and W911NF-08-1-0452, and by an ERDC contract W9132V-11-C-0003. S. Har-Peled is supported by NSF grants CCF-09-15984 and CCF-12-17462. S. Suri and H. Yıldız are supported by NSF grants CCF-1161495 and CNS-1035917.

References

1. Agarwal, P.K., Aronov, B., Har-Peled, S., Phillips, J.M., Yi, K., Zhang, W.: Nearest neighbor searching under uncertainty II. In: Proc. 32nd ACM Sympos. Principles Database Syst., pp. 115–126 (2013)
2. Agarwal, P.K., Cheng, S., Tao, Y., Yi, K.: Indexing uncertain data. In: Proc. 28th ACM Sympos. Principles Database Syst., pp. 137–146 (2009)
3. Agarwal, P.K., Har-Peled, S., Suri, S., Yıldız, H., Zhang, W.: Convex hulls under uncertainty. CoRR abs/1406.6599 (2014), `http://arxiv.org/abs/1406.6599`
4. Agarwal, P.K., Sharir, M., Welzl, E.: Algorithms for center and Tverberg points. ACM Trans. Algo. 5(1), 5:1–5:20 (2008)
5. Chazelle, B.: Cutting hyperplanes for divide-and-conquer. Discrete Comput. Geom. 9(1), 145–158 (1993)
6. Chazelle, B., Guibas, L.J., Lee, D.T.: The power of geometric duality. BIT 25(1), 76–90 (1985)
7. Dalvi, N.N., Ré, C., Suciu, D.: Probabilistic databases: Diamonds in the dirt. Commun. ACM 52(7), 86–94 (2009)
8. Jørgensen, A., Löffler, M., Phillips, J.: Geometric computations on indecisive points. In: Proc. 12th Workshop Algorithms Data Struct., pp. 536–547 (2011)
9. Kamousi, P., Chan, T.M., Suri, S.: Closest pair and the post office problem for stochastic points. In: Proc. 12th Workshop Algorithms Data Struct., pp. 548–559 (2011)
10. Kamousi, P., Chan, T., Suri, S.: Stochastic minimum spanning trees in euclidean spaces. In: Proc. 27th Annu. Sympos. Comput. Geom., pp. 65–74 (2011)
11. Löffler, M.: Data Imprecision in Computational Geometry. Ph.D. thesis, Dept. Computer Sci. (2009)
12. Matoušek, J.: Computing the center of planar point sets. In: Goodman, J.E., Pollack, R., Steiger, W. (eds.) Computational Geometry: Papers from the DIMACS Special Year, pp. 221–230. Amer. Math. Soc. (1991)
13. Matoušek, J., Schwarzkopf, O.: Linear optimization queries. In: Proc. 8th Annu. Sympos. Comput. Geom, pp. 16–25 (1992)
14. Phillips, J.: Small and Stable Descriptors of Distributions for Geometric Statistical Problems. Ph.D. thesis, Dept. Computer Sci. (2009)
15. Seidel, R.: Convex hull computations. In: Goodman, J.E., O'Rourke, J. (eds.) Handbook of Discrete and Computational Geometry, pp. 495–512. CRC Press (2004)
16. Suri, S., Verbeek, K., Yıldız, H.: On the most likely convex hull of uncertain points. In: Proc. 21st Annu. European Sympos. Algorithms, pp. 791–802 (2013)
17. Zhao, Z., Yan, D., Ng, W.: A probabilistic convex hull query tool. In: Proc. 15th Int. Conf. on Ext. Database Tech., pp. 570–573 (2012)

The Space-Stretch-Time Tradeoff in Distance Oracles

Rachit Agarwal

University of California at Berkeley, CA, USA
ragarwal@berkeley.edu

Abstract. We present new distance oracles for computing distances of stretch less than 2 on general weighted undirected graphs. For the realistic case of sparse graphs and for any integer k, the new oracles return paths of stretch $1+1/k$ and exhibit a smooth three-way tradeoff of $S \times T^{1/k} = O(n^2)$ between space S, stretch and query time T. This significantly improves the state-of-the-art *for each point in the space-stretch-time tradeoff space*, and matches the known space-time curve for stretch 2 and larger. We also present new oracles for stretch $1 + 1/(k + 0.5)$. A particularly interesting case is of stretch $5/3$, where improving the query time of our oracles from T to $T^{1-\varepsilon}$ for any $\varepsilon > 0$ would lead to the first purely $o(mn)$-time combinatorial algorithm for Boolean Matrix Multiplication, a longstanding open problem.

1 Introduction

A distance oracle is a compact representation of all-pair shortest path matrix of a graph. A stretch-c oracle for a weighted undirected graph $G = (V, E)$ returns, for any pair of vertices $s, t \in V$ at distance $d(s, t)$, a distance estimate $\delta(s, t)$ that satisfies $d(s, t) \leq \delta(s, t) \leq c \cdot d(s, t)$. Let $n = |V|$ and $m = |E|$. For general graphs, Thorup and Zwick [31] showed a fundamental space-stretch tradeoff — for any integer $k \geq 2$, they designed an oracle of size $O(kn^{1+1/k})$ that returned distances of stretch $(2k - 1)$ in $O(k)$ time; the construction time of their oracle was $\widetilde{O}(kmn^{1/k})$, in expectation. The Thorup-Zwick (TZ) oracle was a significant improvement over previous constructions that had much higher stretch and/or query time [8,15,21].

Improvements in Construction and Query Time. Much of the early research following the TZ result focused on improving the construction time. Roditty, Thorup and Zwick [27] derandomized the TZ construction. Baswana and Sen [11] improved the construction time to $O(n^2)$ for unweighted graphs. Their result was extended to weighted graphs by Baswana and Kavitha [10]. Finally, Wulff-Nilsen [33] achieved subquadratic construction time for weighted graphs with $m = o(n^2)$ edges.

The query time of the TZ oracle is not constant for super-constant stretch. Wulff-Nilsen [32] reduced the query time of the TZ oracle to $O(\log k)$ using a new query algorithm that incorporates binary search within the TZ oracle. Mendel and Naor [22] reduced the query time to $O(1)$ at the expense of increasing the stretch to $O(k)$ and the construction time to $\widetilde{O}(n^{2+1/k})$. Interestingly, Chechik [13] showed that it is possible to reduce the query time of TZ oracle to an absolute constant, without increasing the stretch or space of the original TZ construction.

Improvements in Space-Stretch Tradeoff. Thorup and Zwick showed that, assuming a girth conjecture of Erdős, any oracle that returns distances of stretch less than $(2k+1)$ must have size $\Omega(n^{1+1/k})$. However, the hard instances used to prove this lower bound are extremely dense graphs; for instance, their construction uses a graph with $m = \Omega(n^2)$ edges to prove the space lower bound for stretch less than 3. For graphs with $m = o(n^2)$ edges, it may in fact be possible to get a better space-stretch tradeoff — their result merely implies a trivial space lower bound of $\Omega(m)$, that is, compression is impossible.

However, improving the space-stretch tradeoff turned out to be a much harder problem. Until 2010, a better tradeoff was known only for special graph classes such as planar graphs [20, 30], bounded-genus and minor-free graphs [19], power-law graphs [14] and random graphs [17]. Pătraşcu and Roditty [23] achieved the first breakthrough, constructing a stretch-2 constant-time oracle of size $\widetilde{O}(n^{4/3}m^{1/3})$. Their result was generalized for larger stretch values by Abraham and Gavoille [1] and by Pătraşcu, Roditty and Thorup [24]. The original construction of Pătraşcu and Roditty was rather complex; simpler construction and analysis are now known [3].

Lower Bounds. Sommer *et al.* [28] proved in the cell-probe model that the size of stretch-s time-t distance oracles is lower bounded by $n^{1+\Omega(1/st)}$. That is, for graphs with $m = \widetilde{O}(n)$ edges, computing distances of constant stretch in constant time requires super-linear space. Conditioned on hardness of set intersection, Pătraşcu and Roditty [23] strengthened their result by proving an $\Omega(n^2)$ space lower bound for constant-time stretch-less-than-2 oracles. Pătraşcu, Roditty and Thorup [24] proved, among other results, a conditional space lower bound of $\Omega(m^{5/3})$ for constant-time stretch-2 oracles. Due to several upper bounds matching these lower bounds [23, 24], these results are believed to provide a complete understanding of the space-stretch tradeoff *for constant-time oracles*.

Distance Oracles with Super-Constant Query Time. The problem of improving the space-stretch tradeoff of the TZ oracle is wide open if one allows super-constant query time. No non-trivial lower bounds are known for this regime and it is possible that there exist constant-stretch oracles of size $\widetilde{O}(m)$ with polylog(n) query time!

Agarwal, Godfrey and Har-Peled [5, 6] constructed oracles with super-constant query time for stretch 2 and larger. Their stretch-2 and stretch-3 oracles achieve a space-time tradeoff of $S \times T = O(n^2)$ and $S \times T^2 = O(n^2)$, respectively, for sparse graphs. For instance, stretch-2 and stretch-3 distances can be computed using $\widetilde{O}(n^{3/2})$ and $\widetilde{O}(n)$ space, respectively, if one allows $O(\sqrt{n})$ query time. Even on graphs with millions of nodes and edges, a query time of $O(\sqrt{n})$ time can be engineered to return results in less than a millisecond [2], an extremely acceptable latency for most real-world applications [2, 5, 18, 26].

Porat and Roditty [25] showed existence of $o(n^2)$-size stretch-less-than-2 oracles for unweighted graphs given super-constant query time. Agarwal and Godfrey [4] explored a general space-stretch-time tradeoff for oracles with stretch less than 2 — their oracle is for general weighted graphs and significantly reduces both the space and query time of the Porat-Roditty oracle for any fixed stretch. For space, stretch, query time and construction time bounds for these oracles, see Table 1.

Table 1. Summary of results and comparison with oracles in [25] and in [4]

Stretch	Space	Query time	Construction time	Remarks	Ref.
$1+\frac{1}{k}$	$\widetilde{O}\left(\frac{nm}{m^{1/(4k+2)}}\right)$	$O\left(\frac{m}{m^{1/(4k+2)}}\right)$	$\widetilde{O}(mn)$	$1 \le \alpha \le n$	[25]
	$\widetilde{O}(m+n^2/\alpha)$	$O((\alpha\mu)^{2k-1})$	$\widetilde{O}(mn/\alpha)$	$1 \le \alpha \le n$	[4]
	$\widetilde{O}(m+n^2/\alpha)$	$O((\alpha\mu)^{k})$	$\widetilde{O}(mn/\alpha)$	$1 \le \alpha \le n$	§3
$1+\frac{1}{k+0.5}$	$\widetilde{O}(m+n^2/\alpha)$	$O((\alpha\mu)^{2k})$	$\widetilde{O}(mn/\alpha)$	$1 \le \alpha \le n$	[4]
	$\widetilde{O}(m+n^2/\alpha)$	$O(\alpha(\alpha\mu)^{k})$	$\widetilde{O}(mn/\alpha)$	$1 \le \alpha \le n$	§4
$1+\frac{2}{3}$	$\widetilde{O}(m+n^2/\alpha)$	$O((\alpha\mu)^{2})$	$\widetilde{O}(mn/\alpha)$	$1 \le \alpha \le (n^2/m)^{1/3}$	[4]
	$\widetilde{O}(m+n^2/\alpha)$	$O(\alpha\mu)$	$\widetilde{O}(mn/\alpha)$	$1 \le \alpha \le (n^2/m)^{1/3}$	§5

1.1 Our Contributions

This paper makes two contributions. Our first contribution is a new space-stretch-time tradeoff for distance oracles for stretch less than 2:

Theorem 1. *Let G be a non-negatively weighted undirected graph with n vertices, m edges and average degree $\mu = 2m/n$. Then, for any fixed $1 \le \alpha \le n$ and for any integer $k \ge 1$, there exist distance oracles of size $\widetilde{O}(m + n^2/\alpha)$ that return distances of stretch $(1+\frac{1}{k})$ in $O((\alpha\mu)^k)$ time and of stretch $(1+\frac{1}{k+0.5})$ in $O(\alpha(\alpha\mu)^k)$ time. For $1 \le \alpha \le n^{2/3}m^{-1/3}$, there also exist oracles of size $\widetilde{O}(m+n^2/\alpha)$ that return distances of stretch $5/3$ in $O(\alpha\mu)$ time. All these oracles can be constructed in time $\widetilde{O}(mn/\alpha)$.*

The first oracle of Theorem 1, for sparse graphs, achieves a space-stretch-time tradeoff of $S \times T^{1/k} = O(n^2)$ for stretch $(1 + 1/k)$. For any fixed space and stretch, the oracle reduces the query time in [4] from T^{2k-1} to T^k (or alternatively, reduces space for any fixed stretch and query time). Interestingly, the space-stretch-time tradeoff achieved by this oracle matches the known space-time tradeoff space for stretch 2 and larger. For instance, setting $k = 1$, we get $S \times T = O(n^2)$ for stretch 2 and setting $1/k = 2$, we get $S \times T^2 = O(n^2)$ for stretch 3, precisely as in [5].

The second oracle reduces the query time from T^{2k-2} in [4] to T^k for any fixed stretch and space. Note that both the first and the second construction also enable non-trivial constructions that were not possible using the results in [4]. For instance, the new results make it possible to compute stretch-1.5 distances using oracles of size $\widetilde{O}(n^{5/3})$ in sub-linear time.

The third oracle of Theorem 1 is particularly interesting. For any space $S = \Omega(n^{5/3})$, this oracle reduces the query time of the stretch-5/3 oracle of [4] from T to $\sqrt{T}$. In particular, this oracle achieves a space-time tradeoff of $S \times T = O(n^2)$, which is same as the stretch-2 oracle of [5]. Essentially, compared to the stretch-2 oracle of [5], this oracle reduces the stretch from 2 to 5/3 without any increase in space or query time for the regime of $S = \Omega(n^{5/3})$.

We argue that the query time of the second and the third oracles may be close to optimal. Specifically, the problem of computing all-pair stretch-less-than-2 distances in undirected graphs is equivalent to combinatorial[1] boolean matrix multiplication (BMM) over the (OR, AND) semiring [16]. Hence, for $k = 1$, if the query time of the second oracle of Theorem 1 can be reduced to $O((\alpha^2\mu)^{1-\varepsilon})$ for any $\varepsilon > 0$, it would be possible to multiply two boolean matrices in time $\widetilde{O}(mn/\alpha + n^2(\alpha^2\mu)^{1-\varepsilon})$. By setting $\alpha = o((m/n)^{\beta})$ for $\beta = \frac{\varepsilon}{2(1-\varepsilon)}$, we get that the time would be $o(mn)$. Hence, improving the query time from T to $T^{1-\varepsilon}$ for any $\varepsilon > 0$ would lead to a purely $o(mn)$ time combinatorial algorithm for BMM, a long standing open problem [9,12].

New Query Algorithms. In contrast to the elegant and compact data structures used in constant-time oracles [1,23,24,31], the data structures for super-constant time oracles [4,5] are usually relatively simpler — in addition to the graph, distance from a few sampled vertices to each vertex in the graph is stored. The main technique used in super-constant time oracles is more sophisticated query algorithms that allow exploring a tradeoff between space, stretch and query time (cf. [4]). Our second contribution is, indeed, such new query algorithms.

Our query algorithms perform a *bidirectional recursion* to compute (not necessarily shortest) distances to vertices in carefully defined neighborhoods of both the source s and the destination t. Specifically, the algorithm explores recursively larger neighborhoods of both s and t in each step, and computes distances from s and t to vertices in the respective neighborhoods. The neighborhoods are defined in a manner that once the recursion depth is reached, the explored neighborhoods either intersect along the shortest path or we are able to prove a non-trivial lower bound on the exact distance between s and t. Intuitively, these neighborhood definitions ensure that two new "subpaths" of the shortest path between s and t are explored in each recursive step (one closer to s and one closer to t). When neighborhoods do not intersect, the length of the shortest of these subpaths times twice the recursion depth is a lower bound on the exact distance between s and t. Moreover, the neighborhood definitions also ensure that the neighborhoods explored in each recursive step also contain at least one of the "landmark" vertices that store distances to each vertex in the graph (computed and stored during graph preprocessing). The path via the landmark vertex in the neighborhood containing the shortest of the subpaths gives us a path with desired stretch.

Our new query algorithms are simpler, faster and compute paths of smaller stretch than the ones in [4]. In contrast to the algorithm in [4] that explores the neighborhood of only either the source or the destination in each recursive step, our new definition of neighborhoods allow us to perform bidirectional recursion. This, in turn, leads to significantly stronger lower bound on the exact distance between the source and the destination without any asymptotic increase in the query time.

[1] Although not defined precisely, we say that an algorithm is "combinatorial" in nature if it does not use algebraic techniques of fast matrix multiplication.

2 Preliminaries

This section sets up the notation and basic results [4, 5, 31] used throughout the paper. We assume that the graph $G = (V, E)$ is a weighted undirected graph with n vertices and m edges with non-negative edge weights.

2.1 Reducing the Problem to Degree-Bounded Graphs

The following lemma shows that the problem of designing oracles and algorithms for computing low stretch distances on weighted graphs with n vertices and m edges is no harder than designing oracles for $O(m/n)$-degree bounded graphs.

Claim 1 ([4–6]). *Let $G = (V, E)$ be a weighted undirected graph with n vertices, m edges with non-negative edge weights, and average degree $\mu = 2m/n$. Then, it is possible to construct an equivalent graph with maximum degree $\Delta = \lceil \mu + 2 \rceil$, such that the new graph has $2n$ vertices, $m + n$ edges, and has the same distances between any pair of vertices as the distance in the original graph between the corresponding vertices. The new graph can be computed in $O(n + m)$ time.*

2.2 Balls and Vicinities, Shortest Distances and Candidate Distances

Let $d(s, t)$ denote the exact distance between any vertex pair $s, t \in V$. For any $V' \subset V$, we denote by $N(V')$ the set of neighbors of vertices in V'. Given G, a vertex v and a subset of vertices $L \subset V$, we use the following definitions:

- **Nearest vertex in set** L **—** $\ell(v)$**:** the vertex $a \in L$ that minimizes $d(v, a)$, ties broken arbitrarily.
- **Ball radius** r_v**:** the distance from v to its nearest neighbor in L, that is, $d(v, \ell(v))$.
- **Ball of a vertex** $B(v)$**:** the set of vertices $w \in V$ for which $d(v, w) < d(v, \ell(v))$.
- **Vicinity of a vertex** $B^\star(v)$**:** the set of vertices in $B(v) \cup N(B(v))$.
- **Candidate distance from** v **to** w **—** $d'_v(w)$**:** cost of the least-cost path from v to w such that all intermediate vertices on this path are contained in $B(v)$; that is:

$$d'_v(w) = \min_{x \in N(w) \cap B(v)} \{d(v, x) + \text{weight of edge}(x, w)\}$$

  If $N(w) \cap B(v) = \emptyset$, we let $d'_v(w) = \infty$.

The following lemma gives an efficient way of sampling vertices for set L such that the ball of each vertex is of bounded size (for degree-bounded graphs, we also get a bound on the size of the vicinity of each vertex):

Lemma 1 ([7, 31]). *Let $G = (V, E)$ be a weighted undirected graph with n vertices, m edges with non-negative weights and maximum degree $\mu = O(m/n)$. For any fixed $1 \leq \alpha \leq n$, there exists a subset of vertices L of size $\widetilde{O}(n/\alpha)$ such that for each vertex $v \in V$, we have that $|B(v)| = O(\alpha)$ and $|B^\star(v)| = O(\alpha\mu)$ with high probability. Moreover, such a set L can be computed in time $\widetilde{O}(m)$.*

For a $\mu = O(m/n)$-degree bounded graph, it is not very hard to construct a set L in time $\widetilde{O}(m\alpha)$ that deterministically guarantees the above bound. The following claim, which settles a sufficient condition for the candidate distance to be equal to the exact shortest distance, will play a crucial role in our proofs:

Claim 2. *Let s, t be a vertex pair such that $t \notin B(s)$. Let $P = (s, x_1, x_2, \ldots, t)$ be a shortest path between s and t. Let x_{i_0} be the first vertex from s along P that does not lie in $B(s)$; that is, let $i_0 = \max\{i \ : \ x_j \in B(s) \cap P, \forall j < i\}$. Then, $d'_s(x_{i_0}) = d(s, x_{i_0})$.*

3 Stretch $\left(1 + \frac{1}{k}\right)$ Oracle

In this section, we prove the first part of Theorem 1: for a weighted undirected graph with n vertices, m edges with non-negative weights, and for any $1 \le \alpha \le n$, there exists an oracle of size $\widetilde{O}(m + n^2/\alpha)$ that returns distances of stretch $1 + 1/k$ in time $O((\alpha\mu)^k)$. We need some notation to succinctly describe the construction.

3.1 i-Balls and i-Vicinities

We will generalize the idea of balls and vicinities from §2.2. In particular, we define the i-vicinity of a vertex $v \in V$, denoted as $\Gamma_i^\star(v)$ as follows:

$$\Gamma_0^\star(v) = \{v\}; \qquad \text{and} \qquad \Gamma_i^\star(v) = \bigcup_{w \in \Gamma_{i-1}^\star(v)} B^\star(w) \tag{1}$$

For instance, the 1-vicinity of any vertex includes all the vertices in its vicinity and the 2-vicinity of any vertex v is the union of all the vicinities of vertices in $B^\star(v)$. Given the definition of i-vicinities, we can now define the i-ball of a vertex v:

$$\Gamma_0(v) = \emptyset; \qquad \text{and} \qquad \Gamma_i(v) = \bigcup_{w \in \Gamma_{i-1}^\star(v)} B(w) \tag{2}$$

Note that $\Gamma_i(v) \subseteq \Gamma_i^\star(v)$ for any vertex v. We will also need a generalization for the definition of the *candidate* distance. Given a vertex v and a vertex w in the i-vicinity of v, the candidate distance from v to w is given by the cost of the least-cost path from v to w such that all intermediate vertices are contained in the i-ball of v. We will slightly abuse the notation and use $d'_v(w)$ to denote this candidate distance.

3.2 Oracle and Query Algorithm

Our oracle is similar to the one used in [4]. Fix some $1 \le \alpha \le n$. The preprocessing algorithm first replaces the original graph with a degree-bounded graph using Claim 1. The algorithm then samples a set L of vertices of size $\widetilde{O}(n/\alpha)$ using the result of Lemma 1. The oracle stores, for each $v \in V$: (1) a hash table storing the shortest distance to each vertex in L; and (2) the nearest neighbor $\ell(v)$ and the ball radius r_v. In addition, the oracle also stores the degree-bounded graph computed in the first step of the preprocessing algorithm.

We now describe our query algorithm (see Algorithm 1). In the first two steps, the query algorithm computes candidate distance from s and from t to each vertex in their respective k-vicinities; these distances are temporarily stored in a hash table. Then, the algorithm computes three sets of paths between s and t. The first set of paths are of the form $s \rightsquigarrow w \rightsquigarrow t$ via vertices w in $\Gamma_k^\star(s) \cap \Gamma_k^\star(t)$. The second set of paths are of the form $s \rightsquigarrow w \rightsquigarrow \ell(w) \rightsquigarrow t$ via vertices $w \in \Gamma_k^\star(s)$. The third set of paths are of the form $t \rightsquigarrow w \rightsquigarrow \ell(w) \rightsquigarrow s$ via vertices $w \in \Gamma_k^\star(t)$. Finally, the least-cost path among all the above three sets of paths is returned.

Algorithm 1. Query algorithm for the stretch-$(1+1/k)$ oracle

```
1: Compute candidate distance from $s$ to each vertex in $\Gamma_k^\star(s)$
2: Compute candidate distance from $t$ to each vertex in $\Gamma_k^\star(t)$
3: $\gamma_1 \leftarrow \infty, \gamma_2 \leftarrow \infty, \gamma_3 \leftarrow \infty$
4: $\gamma_1 \leftarrow \min_{w \in \Gamma_k^\star(s) \cap \Gamma_k^\star(t)} \left\{ d_s'(w) + d_t'(w) \right\}$
5: $\gamma_2 \leftarrow \min_{w \in \Gamma_k^\star(s)} \left\{ d_s'(w) + d(w, \ell(w)) + d(\ell(w), t) \right\}$
6: $\gamma_3 \leftarrow \min_{w \in \Gamma_k^\star(t)} \left\{ d_t'(w) + d(w, \ell(w)) + d(\ell(w), s) \right\}$
7: return $\min\{\gamma_1, \gamma_2, \gamma_3\}$
```

3.3 Analysis

For any pair of vertices $s, t \in V$, let $P(s,t) = (s, x_1, x_2, \ldots, t)$ denote the shortest path between s and t. Let

$$w_i^s(t) = x_{i_0}, \quad \text{where } i_0 = \max\{i \ : \ x_j \in B(w_{i-1}^s(t)) \cap P(s,t), \forall j < i\}; \qquad w_0^s(t) = t$$

Intuitively, $w_i^s(t)$ is the first vertex from $w_{i-1}^s(t)$ along $P(s,t)$ that is not contained in the ball of $w_{i-1}^s(t)$. Let

$$r_i^s(t) = \min_{j \le i}\{d(w_j^s(t), \ell(w_j^s(t)))\}$$

that is, $r_i^s(t)$ is the smallest ball radius among all vertices $w_j^s(t)$ for $j \le i$. When the context is clear, we will denote $w_i^s(t)$ and $r_i^s(t)$ simply as w_i^s and r_i^s. We will need the following claims to prove our main result:

Claim 3. *Let $P(s,t) = (s, x_1, x_2, \ldots, t)$ be the shortest path between a pair of vertices s and t. Let i_0 and j_0 be such that $w_k^s = x_{i_0}$ and $w_k^t = x_{j_0}$. Then, for all $i \le i_0$, $d_s'(x_i) = d(s, x_i)$ and for all $j \ge j_0$, $d_t'(x_j) = d(t, x_j)$.*

Claim 4. *For any vertex pair s, t, we have that $d(s, w_i^s) \ge i \cdot r_{i-1}^s$ and $d(t, w_i^t) \ge i \cdot r_{i-1}^t$.*

Claim 5. *For any pair of vertices $s, t \in V$, if $w_k^s \notin \Gamma_k^\star(t)$, then we have that $d(s,t) \ge 2k \min\{r_{k-1}^s, r_{k-1}^t\}$.*

Claim 6. *For any pair of vertices $s, t \in V$, the query algorithm returns a distance estimate of at most $d(s,t) + 2\min\{r_{k-1}^s, r_{k-1}^t\}$.*

Proof of First Oracle of Theorem 1. The oracle stores the input graph and the distance from each vertex in the graph to each vertex in a set L of size $\widetilde{O}(n/\alpha)$; hence, the size of the oracle is $\widetilde{O}(m + n^2/\alpha)$. Constructing the oracle requires computing a shortest path tree from each vertex in set L, and hence, requires time $\widetilde{O}(mn/\alpha)$.

Next, we bound the query time of the query algorithm. We first claim that the size of the k-vicinity of each vertex is bounded by $O((\alpha\mu)^k)$. This follows from the definition of the i-vicinity and from the fact that the size of the vicinity of each vertex is bounded by $O(\alpha\mu)$. Furthermore, the candidate distance from any vertex v to vertices in $B^\star(v)$ can be computed in $O(\alpha\mu)$ time. Hence, by definition of i-vicinity, it takes time $O((\alpha\mu)^k)$ to compute the candidate distance from s to vertices in $\Gamma^\star_k(s)$. Finally, lines (4), (5) and (6) of Algorithm 1 take time linear in the size of the i-vicinities of s and t, leading to the desired bound of $O((\alpha\mu)^k)$ on query time.

Finally, we prove a bound on stretch. If $w^s_k \in \Gamma^\star_k(t)$, then $\gamma_1 \leq d'_s(w^s_k) + d'_t(w^s_k) = d(s, w^s_k) + d(t, w^s_k) = d(s,t)$; hence, the exact distance is returned. Consider the case when $w^s_k \notin \Gamma^\star_k(t)$. Then, by Claim 5, we have that the distance between s and t is lower bounded by $d(s,t) \geq 2k \min\{r^s_{k-1}, r^t_{k-1}\}$. On the other hand, from Claim 6, the distance returned by the query algorithm is at most $d(s,t) + 2\min\{r^s_{k-1}, r^t_{k-1}\} \leq d(s,t) + 2d(s,t)/(2k)$, leading to the desired bound on stretch. □

4 Stretch $\left(1 + \frac{1}{k+0.5}\right)$ Oracle

We now prove the second part of Theorem 1: for a weighted undirected graph with n vertices, m edges with non-negative weights, and for any $1 \leq \alpha \leq n$, there exists an oracle of size $\widetilde{O}(m + n^2/\alpha)$ that returns distances of stretch $1 + 1/(k + 0.5)$ in time $O(\alpha(\alpha\mu)^k)$. See notation in §2.2 and §3.1.

4.1 Oracle and Query Algorithm

We will use the oracle of §3.2 with the addition that the exact distance from each vertex v to each vertex in $B(v)$ will be stored within the oracle. The query algorithm for this oracle (see Algorithm 2) is similar to that of Algorithm 1 with the only difference that the k-vicinities $\Gamma^\star_k(s)$ and $\Gamma^\star_k(t)$ are now replaced by $(k+1)$-balls $\Gamma_{k+1}(s)$ and $\Gamma_{k+1}(t)$, respectively (and γ_1, γ_2 and γ_3 modified accordingly).

4.2 Analysis

The proof is facilitated by the following two claims that are used to bound the stretch of the oracle:

Claim 7. *For any vertex pair s, t, if $w^s_k \notin \Gamma_{k+1}(t)$ then $d(s,t) \geq (2k+1)\min\{r^s_{k-1}, r^t_k\}$.*

Claim 8. *For any pair of vertices s, t, the query algorithm returns a distance estimate of at most $d(s,t) + 2\min\{r^s_{k-1}, r^t_k\}$.*

The above two claims directly lead to the stretch bound claimed in Theorem 1. The proof on the size, construction time, and query time follow using straightforward changes in the proof for the first oracle.

Algorithm 2. The query algorithm for stretch-$(1+1/(k+0.5))$ oracle

```
1: Compute candidate distance from $s$ to each vertex in $\Gamma_{k+1}(s)$
2: Compute candidate distance from $t$ to each vertex in $\Gamma_{k+1}(t)$
3: $\gamma_1 \leftarrow \infty, \gamma_2 \leftarrow \infty, \gamma_3 \leftarrow \infty$
4: $\gamma_1 \leftarrow \min_{w \in \Gamma_{k+1}(s) \cap \Gamma_{k+1}(t)} \left\{ d'_s(w) + d'_t(w) \right\}$
5: $\gamma_2 \leftarrow \min_{w \in \Gamma_{k+1}(s)} \left\{ d'_s(w) + d(w, \ell(w)) + d(\ell(w), t) \right\}$
6: $\gamma_3 \leftarrow \min_{w \in \Gamma_{k+1}(t)} \left\{ d'_t(w) + d(w, \ell(w)) + d(\ell(w), s) \right\}$
7: return $\min\{\gamma_1, \gamma_2, \gamma_3\}$
```

5 Stretch $\left(1+\frac{2}{3}\right)$ Oracle

Finally, we prove the third part of Theorem 1: for a weighted undirected graph with n vertices, m edges with non-negative weights and for any $1 \le \alpha \le n$, an oracle of size $\widetilde{O}(m + n^2/\alpha)$ that returns distances of stretch 5/3 in time $O(\alpha\mu)$.

5.1 Inverse-Ball and Inverse-Vicinities

The **inverse-ball of a vertex, denoted by** $\bar{B}(v)$**,** is the set of vertices w that contain v in their ball. Similar, the **inverse-vicinity of a vertex, denoted by** $\bar{B}^\star(v)$**,** is the set of vertices w for which $v \in B^\star(w)$. For constructing this oracle, we will use a different sampling technique given by the following lemma:

Lemma 2 ([29,31]). *Let $G = (V, E)$ be a weighted undirected graph with n vertices, m edges and maximum degree $\mu = 2m/n$. For any fixed $1 \le \alpha \le n$, there exists a subset of vertices L of expected size $\widetilde{O}(n/\alpha)$ such that for each vertex $v \in V$, we have that $|B(v)| = O(\alpha)$, $|\bar{B}(v)| = O(\alpha)$, $|B^\star(v)| = O(\alpha\mu)$ and $|\bar{B}^\star(v)| = O(\alpha\mu)$. Moreover, such a set L can be computed in expected time $\widetilde{O}(m\alpha)$.*

5.2 Oracle and Query Algorithm

Fix some $1 \le \alpha \le n$. The preprocessing algorithm first replaces the original graph with a degree-bounded graph using the result of Corollary 1. The algorithm then samples a set L of vertices of size $\widetilde{O}(n/\alpha)$ using the result of Lemma 2. The algorithm then constructs a data structure that stores, for each $v \in V$:

– a hash table storing the shortest distance to each vertex in L;
– the nearest neighbor $\ell(v)$ and the ball radius r_v;
– a hash table storing the distance $d'_s(w) = \min_{x \in B^\star(v) \cap B(w)} d'_s(x) + d(x, w)$ to each vertex w in the set $S_v = \{w \ : \ B^\star(v) \cap B(w) \neq \emptyset\}$, that is, to all vertices w whose ball intersects with the *vicinity* of v.

The oracle also stores the degree-bounded graph computed in the first step of the preprocessing algorithm.

We now describe our query algorithm (see Algorithm 3). In the first and the second step, the query algorithm computes candidate distances from s and t to vertices in their respective vicinities; these distances are temporarily stored in a hash table. The algorithm then computes three set of paths. The first set of paths is of the form $s \rightsquigarrow w \rightsquigarrow w' \rightsquigarrow t$ for some $w \in B^{\star}(s)$ and $w' \in S_s \cap B^{\star}(t)$. The second set of paths are of the form $s \rightsquigarrow w \rightsquigarrow \ell(w) \rightsquigarrow t$ for vertices $w \in B^{\star}(s)$ and the final set of paths are of the form $t \rightsquigarrow w \rightsquigarrow \ell(w) \rightsquigarrow s$ for vertices $w \in B^{\star}(t)$. The least-cost path among these paths is returned by the algorithm.

Algorithm 3. The query algorithm for the third oracle of Theorem 1

```
1: Compute candidate distance from $s$ to each vertex in $B^{\star}(s)$
2: Compute candidate distance from $t$ to each vertex in $B^{\star}(t)$
3: $\gamma_1 \leftarrow \infty, \gamma_2 \leftarrow \infty, \gamma_3 \leftarrow \infty$
4: $\gamma_1 \leftarrow \min_{w \in S_s \cap B^{\star}(t)} \left\{ d(s,w) + d'_t(w) \right\}$
5: $\gamma_2 \leftarrow \min_{w \in B^{\star}(s)} \left\{ d'_s(w) + d(w,\ell(w)) + d(\ell(w),t) \right\}$
6: $\gamma_3 \leftarrow \min_{w \in B^{\star}(t)} \left\{ d'_t(w) + d(w,\ell(w)) + d(\ell(w),s) \right\}$
7: return $\min\{\gamma_1, \gamma_2, \gamma_3\}$
```

5.3 Analysis

Claim 9. *Let $P = (s, x_1, x_2, \ldots, t)$ be the shortest path between any pair of vertices s and t. Let $i_0 = \max\{i | x_i \notin P \cap B^{\star}(t)\}$ and $w = x_{i_0+1}$. If $w \notin B(s)$, then $d(s,t) \geq r_s + r_t$.*

Lemma 3. *Let $G = (V, E)$ be a weighted undirected graph with n vertices, m edges and maximum degree $\mu = O(m/n)$. For any fixed $1 \leq \alpha \leq n$, let L be the set of vertices sampled using the algorithm of Lemma 2. Then, $\sum_{v \in V} |S_v| \leq O(m\alpha^2)$.*

Claim 10. *Let $G = (V, E)$ be a weighted undirected graph with n vertices, m edges and maximum degree $\mu = O(m/n)$. For any fixed $1 \leq \alpha \leq n$, let L be the set of vertices sampled using Lemma 2. Then, constructing a hash table that contains, for each vertex $v \in V$, distance to each vertex in S_v can be constructed in time $O(m\alpha^2)$.*

Proof of the Third Oracle of Theorem 1. The oracle stores, in addition to the oracle of [4], a distance from each vertex v to vertices in set S_v. Using Lemma 3, it follows that the size of the oracle if $\widetilde{O}(m\alpha^2 + m + n^2/\alpha)$; for $1 \leq \alpha \leq n^{2/3}m^{-1/3}$, the size is $\widetilde{O}(m + n^2/\alpha)$ as desired. The construction of the oracle requires running a shortest path algorithm from each vertex in L and computing distances to vertices in set S_v for each vertex v. Using Lemma 2 and Claim 10, it follows that the oracle can be constructed in time $\widetilde{O}(m\alpha^2 + n^2/\alpha)$. Finally, to bound the query time, recall that the size of the vicinity of each vertex is bounded by $O(\alpha\mu)$ and a candidate distance to each vertex in the vicinity can be computed in time $O(\alpha\mu)$; the bound follows.

Let $P = (s, x_1, x_2, \ldots, t)$ be the shortest path between s and t. Let $i_0 = \max\{i | x_i \notin P \cap B^{\star}(t)\}$ and $w = x_{i_0+1}$; note that $x_{i_0} \notin B^{\star}(t)$ and hence, $w \in B^{\star}(t) \setminus B(t)$. If $w \in S_s$, we get that $\gamma_1 \leq d(s,w) + d'_t(w) = d(s,w) + d(t,w) = d(s,t)$, since w

lies along P; hence, the algorithm returns the exact distance. Consider the case when $w \notin S_s$. In this case, using Lemma 9, we get that $d(s,w) \geq 2\min\{r_s, r_w\}$; also $d(t,w) \geq r_t$. Since w lies along the shortest path between s and t, we get that $d(s,t) \geq 2\min\{r_s, r_w\} + r_t \geq 3\min\{r_s, r_w, r_t\}$. We now give an upper bound on the distance returned by the query algorithm. Note that $s \in B^\star(s)$ and $t \in B^\star(t)$; it follows that $\gamma_2 \leq d(s,\ell(s)) + d(\ell(s),t) \leq 2d(s,\ell(s)) + d(s,t) = 2r_s + d(s,t)$. Similarly, we get that $\gamma_3 \leq 2r_t + d(s,t)$. Finally, since $w \in B^\star(t)$, we get that $\gamma_3 \leq d'_t(w) + d(w,\ell(w)) + d(\ell(w),s)$. Since w lies along the shortest path between s and t, we get that $d'_t(w) = d(t,w)$; using this along with triangle inequality, we get that $\gamma_3 \leq d(t,w) + 2d(w,\ell(w)) + d(w,s) = 2r_w + d(s,t)$. Hence, $\gamma_3 \leq 2\min\{r_w, r_t\} + d(s,t)$. Since the algorithm returns $\min\{\gamma_2, \gamma_3\}$, the returned distance is at most $2\min\{r_s, r_t, r_w\} + d(s,t)$. The proof follows using the upper bound established above, which says that $\min\{r_s, r_t, r_w\} \leq d(s,t)/3$. □

6 Open Problems

We close the discussion with some of the most interesting open problems:

- Is it possible to prove or disprove a separation between oracles with stretch-k and stretch-less-than-k for $1 < k < 2$? In particular, do stretch-4/3 oracles require more space or time compared to stretch-3/2 oracles?
- There is an interesting problem related to improving the lower order terms in our results. Specifically, if one can reduce the query time of our algorithm of Theorem 1 (for $k = 1$, stretch-$(1 + 1/(k + 0.5))$) by $\log^c(n)$ for some large enough c, we would get a combinatorial algorithm for BMM that is asymptotically faster than the state-of-the-art [12]. Is it possible?

Finally, the most interesting open problem is to prove or disprove the existence of near-linear size oracles that compute distances of $O(1)$ stretch in polylog(n) time.

Acknowledgments. The author would like to thank Philip Brighten Godfrey and Mikkel Thorup for many helpful discussions.

References

1. Abraham, I., Gavoille, C.: On approximate distance labels and routing schemes with affine stretch. In: Peleg, D. (ed.) Distributed Computing. LNCS, vol. 6950, pp. 404–415. Springer, Heidelberg (2011)
2. Agarwal, R., Caesar, M., Godfrey, P.B., Zhao, B.Y.: Shortest paths in less than a millisecond. In: SIGCOMM WOSN (2012)
3. Agarwal, R., Godfrey, P.B.: Brief announcement: A simple stretch 2 distance oracle. In: PODC (2013)
4. Agarwal, R., Godfrey, P.B.: Distance oracles for stretch less than 2. In: SODA (2013)
5. Agarwal, R., Godfrey, P.B., Har-Peled, S.: Approximate distance queries and compact routing in sparse graphs. In: INFOCOM (2011)
6. Agarwal, R., Godfrey, P.B., Har-Peled, S.: Faster approximate distance queries and compact routing in sparse graphs (2012)
7. Alon, N., Spencer, J.H.: The probabilistic method, vol. 57. Wiley Interscience (1992)
8. Awerbuch, B., Berger, B., Cowen, L., Peleg, D.: Near-linear time construction of sparse neighborhood covers. SIAM Journal on Computing 28(1), 263–277 (1998)

9. Bansal, N., Williams, R.: Regularity lemmas and combinatorial algorithms. In: FOCS (2009)
10. Baswana, S., Kavitha, T.: Faster algorithms for approximate distance oracles and all-pair small stretch paths. In: FOCS (2006)
11. Baswana, S., Sen, S.: Approximate distance oracles for unweighted graphs in expected $O(n^2)$ time. ACM Transactions on Algorithms 2(4), 557–577 (2006)
12. Blelloch, G.E., Vassilevska, V., Williams, R.: A new combinatorial approach for sparse graph problems. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part I. LNCS, vol. 5125, pp. 108–120. Springer, Heidelberg (2008)
13. Chechik, S.: Approximate distance oracles with constant query time. In: STOC (2014)
14. Chen, W., Sommer, C., Teng, S.-H., Wang, Y.: A compact routing scheme and approximate distance oracle for power-law graphs. ACM Transactions on Algorithms 9(1), 4:1–4:26 (2012)
15. Cohen, E.: Fast algorithms for constructing t-spanners and paths with stretch t. SIAM Journal on Computing 28(1), 210–236 (1998)
16. Dor, D., Halperin, S., Zwick, U.: All pairs almost shortest paths. In: FOCS (1996)
17. Enachescu, M., Wang, M., Goel, A.: Reducing maximum stretch in compact routing. In: INFOCOM (2008)
18. Gubichev, A., Bedathur, S., Seufert, S., Weikum, G.: Fast and accurate estimation of shortest paths in large graphs. In: CIKM (2010)
19. Kawarabayashi, K.-I., Klein, P.N., Sommer, C.: Linear-space approximate distance oracles for planar, bounded-genus and minor-free graphs. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part I. LNCS, vol. 6755, pp. 135–146. Springer, Heidelberg (2011)
20. Kawarabayashi, K.-I., Sommer, C., Thorup, M.: More compact oracles for approximate distances in undirected planar graphs. In: SODA (2013)
21. Matoušek, J.: On the distortion required for embedding finite metric spaces into normed spaces. Israel Journal of Mathematics 93(1), 333–344 (1996)
22. Mendel, M., Naor, A.: Ramsey partitions and proximity data structures. Journal of European Mathematical Society 2(9), 253–275 (2007)
23. Pătraşcu, M., Roditty, L.: Distance oracles beyond the Thorup-Zwick bound. In: FOCS (2010)
24. Pătraşcu, M., Roditty, L., Thorup, M.: A new infinity of distance oracles for sparse graphs. In: FOCS (2012)
25. Porat, E., Roditty, L.: Preprocess, set, *query!*. In: Demetrescu, C., Halldórsson, M.M. (eds.) ESA 2011. LNCS, vol. 6942, pp. 603–614. Springer, Heidelberg (2011)
26. Potamias, M., Bonchi, F., Castillo, C., Gionis, A.: Fast shortest path distance estimation in large networks. In: CIKM (2009)
27. Roditty, L., Zwick, U.: Replacement paths and k simple shortest paths in unweighted directed graphs. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 249–260. Springer, Heidelberg (2005)
28. Sommer, C., Verbin, E., Yu, W.: Distance oracles for sparse graphs. In: FOCS (2009)
29. Thorup, M., Zwick, U.: Compact routing schemes. In: SPAA (2001)
30. Thorup, M.: Compact oracles for reachability and approximate distances in planar digraphs. Journal of the ACM 51(6), 993–1024 (2004)
31. Thorup, M., Zwick, U.: Approximate distance oracles. Journal of the ACM 52(1), 1–24 (2005)
32. Wulff-Nilsen, C.: Approximate distance oracles with improved query time. In: SODA (2013)
33. Wulff-Nilsen, C.: Approximate distance oracles with improved preprocessing time. In: SODA (2012)

Distribution-Sensitive Construction of the Greedy Spanner

Sander P.A. Alewijnse, Quirijn W. Bouts[*], and Alex P. Ten Brink

Eindhoven University of Technology, The Netherlands
q.w.bouts@tue.nl

Abstract. The greedy spanner is the highest quality geometric spanner (in e.g. edge count and weight, both in theory and practice) known to be computable in polynomial time. Unfortunately, all known algorithms for computing it on n points take $\Omega(n^2)$ time, limiting its use on large data sets.

We observe that for many point sets, the greedy spanner has many 'short' edges that can be determined locally and usually quickly, and few or no 'long' edges that can usually be determined quickly using local information and the well-separated pair decomposition. We give experimental results showing large to massive performance increases over the state-of-the-art on nearly all tests and real-life data sets. On the theoretical side we prove a near-linear expected time bound on uniform point sets and a near-quadratic worst-case bound.

Our bound for point sets drawn uniformly and independently at random in a square follows from a local characterization of t-spanners we give on such point sets: we give a geometric property that holds with high probability on such point sets. This property implies that if an edge set on these points has t-paths between pairs of points 'close' to each other, then it has t-paths between all pairs of points.

This characterization gives a $O(n \log^2 n \log^2 \log n)$ expected time bound on our greedy spanner algorithm, making it the first subquadratic time algorithm for this problem on any interesting class of points. We also use this characterization to give a $O((n + |E|) \log^2 n \log \log n)$ expected time algorithm on uniformly distributed points that determines if E is a t-spanner, making it the first subquadratic time algorithm for this problem that does not make assumptions on E.

1 Introduction

A *Euclidean graph* on a set of n points in the Euclidean plane is a weighted graph with geometric distances as edge weights. If a shortest route in the graph is at most t times longer than the direct geometric distance between its endpoints, we say these endpoints *have a t-path*: a Euclidean graph is a t-spanner if all pairs of points have t-paths. For any $t > 1$, we can efficiently find a t-spanner with $O\left(\frac{n}{t-1}\right)$ edges in the Euclidean plane [17]. These 'approximations' have

[*] Q.W. Bouts is supported by the Netherlands Organisation for Scientific Research (NWO) under project no. 639.023.208

few edges compared to the complete graph, while approximately maintaining distances, making them a useful tool in many areas.

Bounded degree spanners are used in wireless network design [13], where for example points of high degree tend to have problems with interference. By using such a bounded degree spanner the problem of interference is minimized while the connectivity is maintained. A considerable amount of research has been done on spanners [14, 17] since they were introduced in network design [18] and in geometry [10]. Spanners have been used as components in various geometric and distributed algorithms.

Many different construction methods exist for t-spanners, where t can be parameterized to an arbitrary value greater than 1, each having different advantages and disadvantages. An in-depth treatise of these spanners can be found in the book [17]. We focus on the greedy spanner, which is defined as the graph resulting from repeatedly adding the edge between the closest pair of points which do not have a t-path yet. The result is a very sparse graph with assymptotically optimal edge count, degree and weight. On uniform point sets and for $t = 2$, one of its closest well-known competitors with respect to these three properties is the Θ-graph. It has about ten times as many edges, twenty times higher total weight and six times higher maximum degree. Figure 1 clearly shows the contrast between these two spanners. Unfortunately, all known algorithms computing the greedy spanner use $\Omega(n^2)$ time [3], making the spanner impractical to compute.

We observed that on real-world examples, the greedy spanner contains mostly short edges with at most a few longer edges. Whether an edge is placed depends only on the points and edges in an ellipse with its endpoints as foci and with eccentricity $1/t$, which is a small area for short potential edges, hopefully containing few points. We can therefore find these short edges using a bucketing scheme, giving a speedup on such point sets.

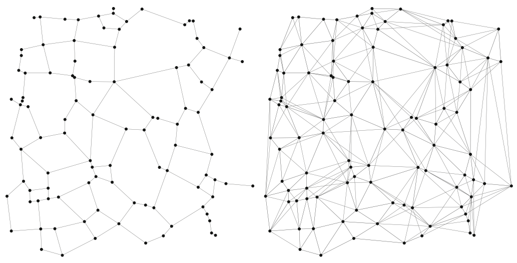


Fig. 1. The left rendering shows the greedy spanner on 100 points distributed uniformly in a square with $t = 2$. The right rendering shows the Θ-graph on the same points with $k = 6$ for which it was recently proven it achieves a dilation of 2.

For the 'long' edges, we consider the 'long' well-separated pairs from a Well-separated pair decomposition (WSPD) [9]. We first compute information from the 'short' edges, attempting to find witnesses that show that certain 'long' well-separated pairs will not contain greedy spanner edges. This information is represented by *path-hyperbola*. We then perform a standard algorithm [3] on the (hopefully only few) well-separated pairs for which we cannot find such a witness.

We present experimental results showing that the above algorithm works very well on many data sets, ranging from real-world data sets to sets which are generated according to different distributions. Speedups vary from an (apparently) linear factor to a constant factor. In particular, on a uniformly distributed point set with 300,000 points, our new algorithm needs 19 minutes to compute the greedy spanner for $t = 2$, while the only other algorithm that can handle point sets of this size [3] (other algorithms need quadratic space, which is prohibitive) needs 17 hours on the same set.

We show that our algorithm has a near-quadratic worst-case time bound. We give formal evidence for the algorithm's good behavior observed in experiments on realistic point sets (which are often reasonably spread out) by analyzing its performance on point sets distributed uniformly and independently at random in a square (or 'uniformly distributed points' for short).

Euclidean graphs are frequently analyzed on uniformly distributed points, both concerning theoretical properties and experimental evaluation of structures and algorithms. One can find examples in computational geometry [8], combinatorial optimization [21] and the analysis of ad-hoc networks [19].

Various spanner constructions have been analyzed on uniformly distributed point sets [1, 7]. Some of these constructions are a t-spanner for fixed t, others are parameterizable with arbitrary $t > 1$. Relatively sharp bounds have been obtained on various qualities of these spanners. This gives insight into the behavior of these constructions in situations arguably closer to realistic point sets than worst case situations.

The spanner constructions studied in these analyses have a 'local' characterization: for example, Gabriel graphs connect u, v if the circle having uv as its diameter contains no points other than u and v. For graphs with such a local characterization there are well-developed techniques to analyze them on uniformly distributed points [11]. In this paper, however, we look at the 'global' property t-spannerness and the greedy spanner, a graph for which the existence of an edge may depend on all other points. Previous analysis techniques do not directly apply on such properties. However, one of our main contributions is to show that with high probability, greedy spanners do admit a local characterization on uniform point sets.

We consider points distributed uniformly and independently at random in a $\sqrt{n} \times \sqrt{n}$ square. We use this square so that if we have an area A, then $O(A)$ points lie in it in expectation. We only consider the case of the Euclidean plane – our results may generalize to higher dimensions, but we did not explore this. In this introduction, when stating bounds, we assume t is a constant.

We prove that such point sets are, with high probability, configured in such a way that for any edge set E, if there are t-paths between points at most $O(\log n)$ away from each other, then there are t-paths between all points. In particular, we show that we can construct a 'witness' of this configuration in $O(n \log^2 n \log\log n)$ expected time if it exists, thus allowing our algorithms to always give the correct answer.

This result easily implies that with high probability the greedy spanner has no long edges (longer than $O(\log n)$) and furthermore that the 'proof' phase of our algorithm will find the witnesses for this if it exists. As the grid strategy works well on uniformly distributed point sets, we obtain a $O(n \log^2 n \log^2 \log n)$ expected time bound on our algorithm. To the best of our knowledge, this algorithm is the first subquadratic algorithm to compute the greedy spanner on any interesting class of point sets.

Another application of our result is a method to test whether a Euclidean graph $G = (P, E)$ is a t-spanner on uniformly distributed points in $O((n + |E|) \log^2 n \log \log n)$ expected time. Various algorithms are known for specific graphs on arbitrary points, but not for arbitrary graphs on specific sets of points. For specific graph classes the minimum t can be computed [2,12], and for general graphs this t can be approximated [16].

The rest of the paper is organized as follows. In Section 2 we introduce *bridgedness* and give a geometric lemma that will help us obtain our results. In Section 3 we show uniform point sets are locally-$O(\log n)$-bridged with high probability. In Section 4 we give several fast algorithms that use this result. Finally, in Section 5 we present experimental results for our algorithm that computes the greedy spanner. Full proofs and additional experimental results can be found in [4].

2 Bridging Points

In this section we will introduce the concept of λ-bridgedness for point sets. We will later use this concept in our characterization of t-spanners on uniformly distributed point sets. We prove two geometric lemmas that will help us with the result of Section 3.

Let P be a finite set of points in $\mathbb{R}^2$, let $n = |P|$, and let $t \in \mathbb{R}$ be the intended dilation ($t > 1$). Let $G = (P, E)$ be a graph on P whose edges are weighted with the Euclidean distance between its endpoints. For two points $u, v \in P$, we denote the Euclidean distance between u and v by $|uv|$, and the network distance in G by $\delta_G(u, v)$ (or $\delta(u, v)$ if G is clear from the context). We say a pair of points (u, v) *has a t-path* if $\delta(u, v) \leq t \cdot |uv|$. If all pairs of points have a t-path, the graph is called a *t-spanner*.

Let $a, b, p, q \in P$ be pairwise different points. We say that the pair (p, q) *bridges* the pair (a, b) if $t \cdot |ap| + |pq| + t \cdot |qb| \leq t \cdot |ab|$. Bridging points guarantee a t-path for (a, b) if (p, q) is an edge and the pairs (a, p) and (q, b) already have t-paths. Note that $|ap|, |qb| < |ab|$ as a consequence.

We say that (p, q) is *mandatory* if the ellipse with foci p and q and eccentricity $1/t$ contains no points in P other than p and q. Any t-path between p and q must fully lie within this ellipse, so a mandatory (p, q) will be in E for any t-spanner.

Let $\lambda \in \mathbb{R}$. We say that a point $a \in P$ is *λ-bridged* if for all $b \in P$ with $|ab| > \lambda$, there exist some mandatory pair of points (p, q), $p, q \in P$, bridging (a, b). We say that the point set P is *λ-bridged* if all points in P are λ-bridged. We say a point $a \in P$ is *locally-λ-bridged* if it is λ-bridged using only mandatory bridging pairs of points at with distance most λ from a. A point set P is *locally-λ-bridged* if all points in P are locally-λ-bridged. Lemma 1 shows the usefulness of this concept. In Lemma 2 we give a sufficient geometric condition for bridging pairs of points.

Lemma 1. *Let P be a set of points that is λ-bridged. For any Euclidean graph $G = (P, E)$ it holds that G is a t-spanner if and only if all pairs of points (a, b), $a, b \in P$, with $|ab| \leq \lambda$ have a t-path in G.*

Lemma 2. *Suppose we are given points $a, b \in P$, rectangles R_1 and R_2 and $t > 1$, such that (as per Fig. 2): R_1 and R_2 lie in between a and b, have a side parallel to ab, have their centers on line segment ab, both have width w and height h, are separated by $s \geq \frac{t+1}{t-1}h$ and R_1 lies closer to a than R_2.*

Then, for any $p, q \in P$ with p lying in R_1 and q in R_2, (p, q) bridges (a, b).

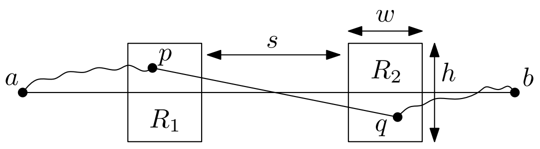


Fig. 2. (p, q) bridges (a, b)

We now use Lemma 2 to prove a stronger statement that we will use to prove the full version of Theorem 4. Let $a, p, q \in P$ be pairwise different points and let region $A \subseteq \mathbb{R}^2$ with $a, p, q \notin A$. We say that the pair (p, q) *bridges* (a, A) if for every point $b \in P$ with $b \in A$ we have that (p, q) bridges (a, b).

Lemma 3. *Assume we are given $a \in P$, a line ℓ through a, an angle $\alpha \leq \pi/4$, a constant c_{max}, rectangles R_1 and R_2 and $t > 1$, such that (as per Fig. 3): R_1 and R_2 have width w and height h, are separated by s, have a side parallel to ℓ, have their centers on ℓ, R_1 lies between a and R_2, R_2 lies at most c_{max} away from a, R_1 lies at least $h/2$ away from a and $s \geq \sqrt{2}\frac{t+1}{t-1}(2\sin(\alpha)c_{max} + h) + h$.*

For the cone with apex a, angle 2α and bisector ℓ, we define A as the area that is at least $c_{cone} = c_{max} + h/2$ away from a. Then for any $p, q \in P$ with p lying in R_1 and q lying in R_2, (p, q) bridges (a, A).

3 Uniform Point Sets

We will now give a sketch of the proof of the following result. The full proof can be found in [4].

Theorem 4. *There exists c_t dependent only on t such that for every $c > 0$, if P is a set of points uniformly and independently distributed at random in a $\sqrt{n} \times \sqrt{n}$ square and n is large enough, then with probability at least $1 - n^{-c}$, P is locally-$(c \cdot c_t \log n)$-bridged.*

We need to prove that every point in P is locally-$(c \cdot c_t \log n)$-bridged simultaneously with high probability. We show that every point is locally-$(c \cdot c_t \log n)$-bridged with sufficiently high probability that a simple union bound shows that it will happen to all points simultaneously with high probability. We use Lemma 3 to achieve this. For ease of presentation, we assume t is constant.

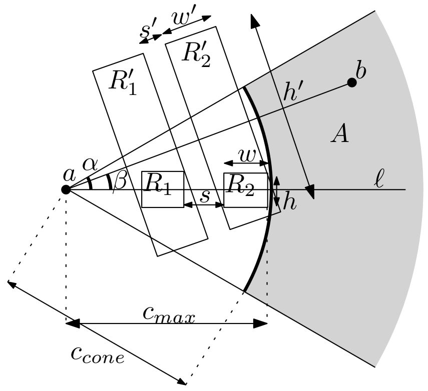


Fig. 3. R_1 and R_2 are covered by R_1' and R_2', according to Lemma 2

The rectangles in Lemma 3 can be chosen to have a roughly constant chance of containing a point, and if we can fulfill the other requirements, the resulting pair of points bridges a relatively large part of $\mathbb{R}^2$. In fact, we need only $\lceil \pi/\alpha \rceil$ cones (we will end up picking $\alpha = O(1/\log n)$) to cover the area we wish to cover, as depicted in Fig. 4. We show the likely existence of a pair of mandatory points that bridges a single cone and use a union bound to show such pairs are likely to exist for all cones simultaneously.

We will place $O(\log n)$ pairs of rectangles in every cone as depicted in Fig. 4. If any pair of boxes ends up containing a point per box, these two points will satisfy the requirements for Lemma 3. We just need this pair of points to be mandatory, and therefore consider an ellipse around such a pair of boxes (defined in terms of the boxes, not the points, for easy analysis), such that if this ellipse is empty apart from these two points, these points must be mandatory. Using a careful analysis, the chance that a pair of boxes contains one point per box and the ellipse contains no more points (an event we will call a 'success') is at least some constant p (dependent only on t). We need only one success per cone and the events are nearly independent (the ellipses do not overlap), so the chance that we get at least one success is at least (roughly) $1 - p^{O(\log n)} = 1 - n^{-O(f(t))}$, which then shows the theorem.

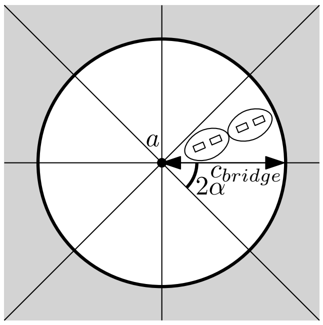


Fig. 4. Covering the plane with cones

4 Algorithms

We first introduce three tools used in the results below. Let c and c_t be as in Theorem 4 throughout this section. The first is that we can divide the input into a $\frac{\sqrt{n}}{c \cdot c_t \log n} \times \frac{\sqrt{n}}{c \cdot c_t \log n}$ grid in $O(n \log n)$ time, with every cell containing in expectation $O((c \cdot c_t \log n)^2)$ points.

The second tool is the 'local' Dijkstra algorithm. It determines for all points at most λ away from a source point s whether it has a t-path to s and if so, their network distance. It differs from the standard Dijkstra algorithm in that it only adds the points to the queue at most λt away from the source s by considering the points lying in cells at most λt away from s, and only considers the edges E_s that have such a point as either endpoint. Using the grid this can be done in $O((\lambda^2 + |E_s|) \log \lambda)$ expected time.

The third tool is called *path-hyperbola.* It is an area given by an origin point $u \in P$, a focus $v \in P$ and an edge set E, and is defined as $PH(u, v, E) = \{a \in \mathbb{R}^2 \mid \delta_{(P,E)}(u, v) + t \cdot |va| \leq t \cdot |ua|\}$. Obviously, if (p, q) bridges (a, b), then $b \in PH(a, q, E)$ for every edge set E with t-paths for pairs of points (u, v) with $|uv| \leq |ab|$, making path-hyperbola at least as powerful as bridging points for guaranteeing t-paths.

If we perform a local Dijkstra on s, we find a set of network distances that induce a set of path-hyperbola. If s is locally-λ-bridged, the union of path-hyperbola will be a superset of the area more than λ away from s, guaranteeing t-paths to all other points. This union can be computed in $O(\lambda^2 \log \lambda)$ expected time: using polar coordinates, the union corresponds to a lower envelope. Since the hyperbolas pairwise intersect at most twice, this envelope has linear complexity and can be computed in $O(n \log n)$ time [5,20]. We can therefore use this to test in $O(\lambda^2 \log \lambda)$ expected time whether s has a t-path to all other points: if the local Dijkstra finds only t-paths but s is not locally-λ-bridged, we can perform a normal Dijkstra without affecting the expected running time.

4.1 Testing t-Spanners

The first application of Theorem 4 and our tools is a faster algorithm to test if a Euclidean graph is a t-spanner on uniformly distributed point sets: we simply run the procedure from the previous section on every point. To the best of our knowledge, this leads to the first subquadratic algorithm for this problem on any interesting class of point sets not making assumptions on E.

Theorem 5. *There is an algorithm that, given a point set P whose points are uniformly distributed in a $\sqrt{n} \times \sqrt{n}$ square and a Euclidean graph E on P, checks if E is a t-spanner using $O((n + |E|)(c_t \log n)^2 \log(c_t \log n))$ expected time, where c_t is a constant dependent only on t.*

4.2 Greedy Spanner

Algorithm. *GreedySpannerOriginal*(V, t)

```
1.  $E \leftarrow \emptyset$
2.  for every pair of distinct points $(u, v)$ in ascending order of $|uv|$
3.      do if $\delta_{(V,E)}(u, v) > t \cdot |uv|$
4.          then add $(u, v)$ to $E$
5.  return $E$
```

Consider the original algorithm above as introduced in [15]. The graph returned by this algorithm is called the *greedy spanner* on V for t and it is obviously a t-spanner, but the algorithm has a $O(n^3 \log n)$ running time.

Lemma 6. *If P is λ-bridged, then the greedy spanner on P does not have edges longer than λ.*

We can combine Lemma 6 with Theorem 4 to quickly compute the greedy spanner on uniform point sets. We first give a preliminary algorithm which we then employ in two greedy spanner algorithms.

Theorem 7. *For every $\lambda > 0$, there is an algorithm that, given a point set P whose points are uniformly distributed in a $\sqrt{n} \times \sqrt{n}$ square, computes in $O(n \log n + n\lambda^2 \log^2 \lambda)$ expected time the edges of the greedy spanner on P for t of length at most λ.*

Proof. We use the algorithm introduced in [3] (we omit an explanation of the machinery introduced there), except we keep Lemma 6 in mind and use our local Dijkstra instead of a normal Dijkstra and only consider well-separated pairs $\{A_i, B_i\}$ with $\min(A_i, B_i) \le \lambda$.

Using the analysis in [3] and using that the greedy spanner has degree $O(1)$, we conclude that if m is the number of considered well-separated pairs, the running time of our modified algorithm is $O(n \log n + \lambda^2 \log \lambda \sum_{i=1}^{m} \min(|A_i|, |B_i|))$. We therefore need to bound
$\sum_{i=1}^{m} \min(|A_i|, |B_i|) \le \sum_{i=1}^{m}(|A_i| + |B_i|) = \sum_{a \in P} |\{\{A_i, B_i\} \mid a \in A_i \vee a \in B_i\}|$.

For any $l \in \mathbb{R}$, a point p can only be in $O(1)$ well-separated pairs of length at most a constant factor higher or lower than l [9, Lemma 4.6.1]. We can therefore partition the well-separated pairs containing p into $O(1)$-sized sets of similar length. As the minimal length per set differs by at least a constant factor, we conclude $|\{\{A_i, B_i\} \mid a \in A_i \vee a \in B_i\}| = O\left(\log \frac{\max_i\{l(\{A_i,B_i\})\}}{\min_i\{l(\{A_i,B_i\})\}}\right)$. This last expression is $O(\log \lambda)$ in expectation on uniform point sets, giving an expected running time of $O(n \log n + n\lambda^2 \log^2 \lambda)$. □

Note that we could have adapted the algorithm from [6], but this algorithm sorts all potential edges, resulting in an expected $O(n \log n\lambda^2 \log \lambda)$ running time, which is slower when filling in $\lambda = O(\log n)$.

Combining Lemma 6, Theorem 4 and Theorem 7 (with $\lambda = c \cdot c_t \log n$) gives:

Corollary 8. *There is an algorithm that, given a point set P whose points are uniformly distributed in a $\sqrt{n} \times \sqrt{n}$ square, computes in $O(n(c_t \log n)^2 \log^2(c_t \log n))$ expected time a graph on P which is with high probability the greedy t-spanner (with c_t is a constant dependent only on t).*

4.3 The Full Distribution-Sensitive Algorithm

The algorithm from Theorem 7 is the first phase of our distribution sensitive algorithm. We now present the second and third phase that ensure that all long edges are also computed.

The second phase gathers path-hyperbola as described at the start of this section. We then consider the well-separated pairs that did not get considered in the first stage of the algorithm and try to prove for them that they will not produce a greedy spanner edge. For the remaining pairs, we employ the algorithm of [3] in the third phase of our algorithm to find the remaining spanner edges.

If for a point $u \in A_i$, the bounding box B_i is covered by the union of path-hyperbola computed for u (testing this takes $O(\log n)$ time), then we say u is *discounted* with respect to $\{A_i, B_i\}$. If all $u \in A_i$ are discounted, then $\{A_i, B_i\}$ will not contain a greedy spanner edge and we say $\{A_i, B_i\}$ is *discounted.* This can be computed in $O(\log n \sum_{i=1}^{m}(|A_i| + |B_i|)) = O(n \log n \log \lambda)$ expected time by an earlier argument.

We then perform the algorithm from [3], with small differences. We ignore pairs that have been discounted in the previous phase, and we do not perform a Dijkstra operation on points which have been discounted with respect to that pair as well. By Theorem 4, all pairs are discounted with high probability and hence this phase takes constant time in expectation on uniform point sets.

In practice, using a λ lower than predicted by Theorem 4 will suffice and be faster. From experiments we observe that $\lambda = \frac{\log n}{\sqrt[4]{t-1} \log \log n}$ is the 'right' bound for the length of the longest edge in the greedy spanner. Using $1.1 \cdot \lambda$ the initial phase nearly always finds all edges, with the second phase usually discounting 99.7% of the pairs and 95% of the points in undiscounted pairs, with the second phase taking about 20% of the time of the first. Using $1.5 \cdot \lambda$, all pairs are typically discounted.

Theorem 9. *There is an algorithm that, given t and a point set P whose points are uniformly distributed in a $\sqrt{n} \times \sqrt{n}$ square, computes in $O(n(c_t \log n)^2 \log^2(c_t \log n))$ expected time its greedy spanner, with c_t a constant dependent only on t. The algorithm uses $O(n^2 \log^2 n)$ time on arbitrary P.*

5 Experimental Results

We have run our algorithm and WSPD-Greedy from [3] on point sets whose size ranged from 500 to 128,000 points. The WSPD-Greedy algorithm has a running time comparable to the other (quadratic space) algorithms. Since running these on more then 10,000 points quickly becomes infeasible we did not include them in our experiments. For a detailed comparison between the major quadratic space algorithms and WSPD-Greedy we refer to [3]. Note that we have verified that all our implemented algorithms give the same output.

Throughout this section we will refer to our algorithm as "Bucketing" in the graphs. We generated point sets according to several distributions. We have recorded space usage and running time (wall clock time). The results are averages over several runs where new point sets were generated each time. We included graphs for the uniform point set and for a clustered point set as these represent the best and worst cases respectively for our algorithm (with respect to our set of tests). To generate the clustered point set we used the same method as [3], that is, for n points, it consists of $\sqrt{n}$ uniformly distributed point sets of $\sqrt{n}$ uniformly distributed points.

5.1 Environment

The algorithms have been implemented in C++. The random generator used was the Mersenne Twister PRNG – we have used a C++ port by J. Bedaux of the C code by the designers of the algorithm, M. Matsumoto and T. Nishimura. We have implemented all other necessary data structures and algorithms not already in the `std` ourselves. The implementations do not use parallelism.

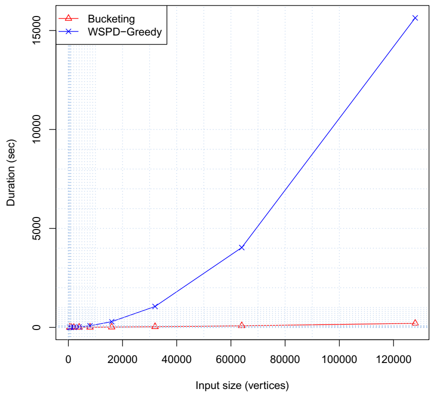
Our experiments have been run on a server using an Intel Xeon E5530 CPU (2.40GHz) and 8GB (1600 MHz) RAM. It runs the Debian 7 OS and we compiled for 64 bits using G++ 4.7.2 with the -O3 option.

5.2 Dependence on Instance Size

We have compared running time and space usage of WSPD-Greedy and our algorithm for different values of n. We plotted the running time for $t = 2$ on uniform and clustered points in Fig. 5. The space usage for both algorithms is linear but our algorithm uses a constant factor less space in practice.

The running time of our algorithm on uniformly distributed points is (nearly) linear making it a massive improvement over WSPD-Greedy. This allows us to calculate greedy spanners on such point sets in a matter of minutes where WSPD-Greedy would need hours or even days for bigger instances.

The clustered point set is a bad case for our algorithm since the greedy spanner will contain a considerable amount of really large edges between clusters. Nevertheless, the algorithm still outperforms WSPD-Greedy by quite a margin. Our experiments on clustered data with smaller t values (up to $t = 1.1$) show that the performance of the algorithms gets more similar as t decreases. On point sets drawn using a uniform or normal distribution our algorithm massively outperforms WSPD-Greedy for both small and large t.



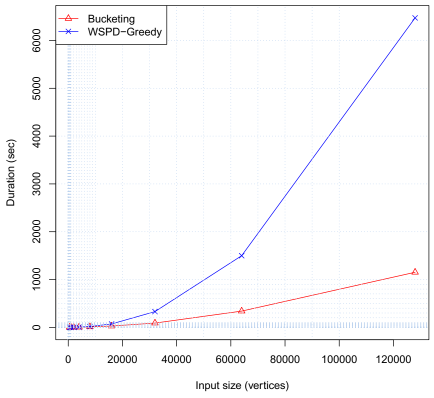


Fig. 5. The left plot shows the running time of our algorithm (Bucketing) and WSPD-Greedy for $t = 2$ on variously sized uniformly distributed instances. The right plot shows the same for clustered instances.

5.3 Real Data

Aside from generated instances we also experimented on some real point sets from the TSPLIB[1]. The performance of our algorithm on these sets seems to be close to the uniform point sets. Figure 6 shows two point sets and their greedy spanners. For the PCB the computation took on average about 2 seconds for $t = 2$ and 11 seconds for $t = 1.1$. The same computations using WSPD-Greedy took 12 and 203 seconds respectively. The bigger Germany instance took 21 and 147 seconds to compute using our algorithm while WSPD-Greedy needed 274 and 7,486 seconds for $t = 2$ and $t = 1.1$. This is a factor 50 improvement for the low t case which reduces the computation time from hours to minutes.

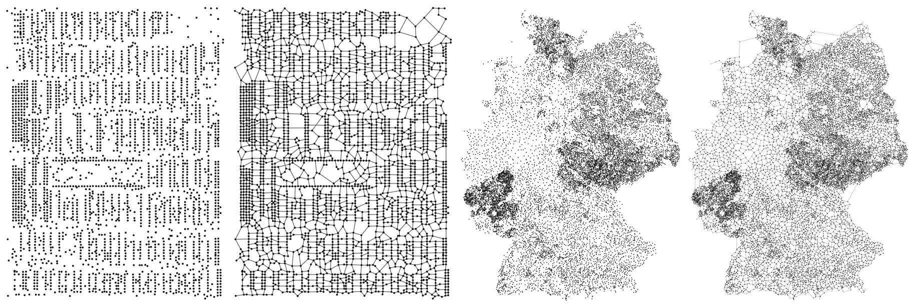


Fig. 6. Real point sets from the TSPLIB and their greedy spanners using $t = 2$. Left: A PCB instance of 3,038 points. Right: Cities in Germany, 15,112 points.

[1] `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/`

6 Conclusion

We have introduced a distribution sensitive algorithm for computing the greedy spanner. Experiments show large improvements in both time and space for most data sets, while results are never worse than the state-of-the-art. The performance gap in many cases becomes even larger for lower t. To explain these results, we have analyzed the algorithm on uniformly distributed point sets.

To this end, we have introduced the concept of *bridgedness* and have shown that point sets that are uniformly distributed in a $\sqrt{n} \times \sqrt{n}$ square are $O(\log n)$-bridged with high probability. This implies that 't-spannerness' is a 'local' property on these point sets: a Euclidean graph is a t-spanner if and only if all pairs of 'close-by' points have t-paths. This locality shows that our algorithm is near-linear on these point sets and yields a near-linear time algorithm for testing whether an edge set is a t-spanner on these point sets.

We leave open several questions that may be answered in future work. First, in our experiments, we have observed that the length of the longest edge of the greedy spanner on uniform point sets tends towards $\frac{\log n}{\sqrt[4]{t-1} \log \log n}$, leaving a gap with our upper bound; similarly, our bridgedness bound may also be improvable. Secondly, it would be interesting to see if our results generalize to higher dimensions. Lastly, there is still no general subquadratic time algorithm for the greedy spanner. Our algorithm could be considered a divide and conquer algorithm where the conquer step may be very slow, possibly susceptible to improvement.

References

[1] Abam, M.A., de Berg, M., Farshi, M., Gudmundsson, J.: Region-fault tolerant geometric spanners. Discr. Comp. Geom. 41(4), 556–582 (2009)

[2] Agarwal, P.K., Klein, R., Knauer, C., Langerman, S., Morin, P., Sharir, M., Soss, M.: Computing the Detour and Spanning Ratio of Paths, Trees, and Cycles in 2D and 3D. Discrete Comput. Geom. 39(1), 17–37 (2008)

[3] Alewijnse, S.P.A., Bouts, Q.W., ten Brink, A.P., Buchin, K.: Computing the greedy spanner in linear space. In: Bodlaender, H.L., Italiano, G.F. (eds.) ESA 2013. LNCS, vol. 8125, pp. 37–48. Springer, Heidelberg (2013)

[4] Alewijnse, S.P.A., Bouts, Q.W., ten Brink, A.P., Buchin, K.: Distribution-sensitive construction of the greedy spanner. CoRR, arXiv:1401.1085 (2014)

[5] Atallah, M.: Some dynamic computational geometry problems. Computers and Mathematics with Applications 11, 1171–1181 (1985)

[6] Bose, P., Carmi, P., Farshi, M., Maheshwari, A., Smid, M.: Computing the greedy spanner in near-quadratic time. Algorithmica 58(3), 711–729 (2010)

[7] Bose, P., Devroye, L., Evans, W., Kirkpatrick, D.: On the spanning ratio of Gabriel graphs and beta-skeletons. SIAM Journal on Discrete Mathematics 20(2), 412–427 (2006)

[8] Buchin, K.: Constructing Delaunay triangulations along space-filling curves. In: Fiat, A., Sanders, P. (eds.) ESA 2009. LNCS, vol. 5757, pp. 119–130. Springer, Heidelberg (2009)

[9] Callahan, P.B.: Dealing with Higher Dimensions: The Well-Separated Pair Decomposition and Its Applications. PhD thesis, Johns Hopkins University, Baltimore, Maryland (1995)

[10] Chew, L.P.: There are planar graphs almost as good as the complete graph. J. Comput. System Sci. 39(2), 205–219 (1989)

[11] Devroye, L.: On the expected size of some graphs in computational geometry. Comput. Math. Appl. 15, 53–64 (1988)

[12] Eppstein, D., Wortman, K.A.: Minimum dilation stars. Comput. Geom. 37(1), 27–37 (2007)

[13] Gao, J., Guibas, L.J., Hershberger, J., Zhang, L., Zhu, A.: Geometric spanners for routing in mobile networks. IEEE J. Selected Areas in Communications 23(1), 174–185 (2005)

[14] Gudmundsson, J., Knauer, C.: Dilation and detours in geometric networks. In: Gonzales, T. (ed.) Handbook on Approximation Algorithms and Metaheuristics, pp. 52-1– 52-16. Chapman and Hall/CRC, Boca Raton (2006)

[15] Keil, J.M.: Approximating the complete Euclidean graph. In: Karlsson, R., Lingas, A. (eds.) SWAT 1988. LNCS, vol. 318, pp. 208–213. Springer, Heidelberg (1988)

[16] Narasimhan, G., Smid, M.: Approximating the stretch factor of Euclidean graphs. SIAM J. Comput. 30(3), 978–989 (2000)

[17] Narasimhan, G., Smid, M.: Geometric Spanner Networks. Cambridge University Press, New York (2007)

[18] Peleg, D., Schäffer, A.A.: Graph spanners. Journal of Graph Theory 13(1), 99–116 (1989)

[19] Santi, P.: Topology control in wireless ad hoc and sensor networks. ACM Computing Surveys (CSUR) 37(2), 164–194 (2005)

[20] Sharir, M., Agarwal, P.: Davenport-Schinzel Sequences and their Geometric Applications. Cambridge University Press (1995)

[21] Steele, J.M.: Probability Theory and Combinatorial Optimization. CBMS-NSF Regional Conference Series in Applied Mathematics, vol. 69. SIAM (1997)

Recognizing Shrinkable Complexes Is NP-Complete*

Dominique Attali[1], Olivier Devillers[2], Marc Glisse[2], and Sylvain Lazard[2]

[1] Gipsa-lab, France
http://www.gipsa-lab.grenoble-inp.fr/~dominique.attali/
[2] INRIA, France
http://www.inria.fr/sophia/members/Olivier.Devillers,
http://geometrica.saclay.inria.fr/team/Marc.Glisse/,
http://www.loria.fr/~lazard/

Abstract. We say that a simplicial complex is *shrinkable* if there exists a sequence of admissible edge contractions that reduces the complex to a single vertex. We prove that it is NP-complete to decide whether a (three-dimensional) simplicial complex is shrinkable. Along the way, we describe examples of contractible complexes which are not shrinkable.

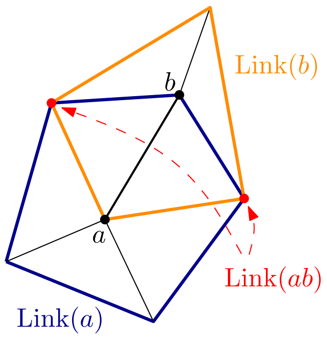
1 Introduction

Edge contraction is a useful operation for simplifying simplicial complexes. An edge contraction consists in merging two vertices, the result being a simplicial complex with one vertex less. By repeatedly applying edge contractions, one can thus reduce the size of a complex and significantly accelerate many computations. For instance, edge contractions are used in computer graphics to decimate triangulated surfaces for fast rendering [14, 16]. For such an application, it may be unimportant to modify topological details and ultimately reduce a surface to a single point since this corresponds to what the observer is expected to see if he is sufficiently far away from the scene [21]. However, for other applications, it may be desirable that every edge contraction preserves the topology. This is particularly true in the field of machine learning when simplicial complexes are used to approximate shapes that live in high-dimensional spaces [1, 6, 8, 10]. Such shapes cannot be visualized easily and their comprehension relies on our ability to extract reliable topological information from their approximating complexes [7, 11, 20].

In this paper, we are interested in edge contractions that preserve the topology, actually the homotopy type, of simplicial complexes. It is known that contracting edges that satisfy the so-called *link condition* preserves the homotopy type of simplicial complexes [13] and, moreover, for triangulated surfaces and

* This work has been supported by ANR project TopData ANR-13-BS01-0008 and by the Advanced Grant of the European Research Council GUDHI (Geometric Understanding in Higher Dimensions).

piecewise-linear manifolds, the link condition *characterizes* the edges whose contraction produces a complex that is homeomorphic to the original one (a constraint that is stronger than preserving the homotopy type) [12, 19]. An edge ab satisfies the *link condition* if the link of ab is equal to the intersection of the links of a and b, where the link of a face f is a simplicial complex defined as follows (see figure): consider the smallest simplicial complex that contains all the faces containing f, i.e. the *star* of f; the link of f is the set of faces disjoint from f in that simplicial complex [12].[1]



We only consider contractions of edges that satisfy the link condition, which implies that the homotopy type is preserved. We refer to such edge contractions as *admissible*; an admissible edge contraction is also called a *shrink* and the corresponding edge is said to be *shrinkable*. After some sequence of shrinks, the resulting complex (possibly a point) does not admit any more shrinkable edges and the complex is called (shrink) *irreducible*.

We are interested in long sequences of shrinks because they produce irreducible complexes of small size and it is natural to ask, in particular, whether a simplicial complex can be reduced to a point using admissible edge contractions. If this is the case, the simplicial complex is called *shrinkable*.

Barnette and Edelson [3] proved that a topological disk is always shrinkable (by *any* sequence of shrinks). They use this property to prove that a compact 2-manifold (orientable or not) of fixed genus admits finitely many triangulations that are (shrink) irreducible [3, 4]. For instance, the number of irreducible triangulations of the torus is 21 [17] and it is at most 396 784 for the double torus [22]. We address in this paper the problem of recognizing whether an arbitrary simplicial complex is shrinkable.

Tancer [23] recently addressed a similar problem where he considered *admissible simplex collapses* instead of admissible edge contractions. An admissible simplex collapse (called elementary collapse in [23]) is the operation of removing a simplex and one of its faces if this face belongs to no other simplex.[2] Such collapses preserve the homotopy type. Similarly to edge contractions, collapses are often used to simplify simplicial complexes, and a simplicial complex is said *collapsible* if it can be reduced to a single vertex by a sequence of admissible collapses. Tancer proved that it is NP-complete to decide whether a given (two-dimensional) simplicial complex is collapsible [23]. The proof is by reduction

[1] In other words, in an *abstract* simplicial complex, the link of σ is the set of faces λ disjoint from σ such that $\sigma \cup \lambda$ is a face of the complex.

[2] Strictly speaking, Tancer calls several of our admissible simplex collapses an elementary collapse. His elementary collapse is the removal of a nonempty non-maximal face σ and the removal of all the faces containing σ if σ is contained in a unique maximal face of the simplicial complex, where maximality is considered for the inclusion in an abstract simplicial complex [23].

from 3-SAT and gadgets are obtained by altering Bing's house [5], a space that is contractible but whose triangulations are not collapsible.

Both questions of collapsibility and shrinkability are related to the question of contractibility: given a simplicial complex, is it contractible? This question is known to be undecidable for simplicial complexes of dimension 5. A proof given in Tancer's paper [23, Appendix] relies on a result of Novikov [24, page 169], which says that there is no algorithm to decide whether a given 5-dimensional triangulated manifold is the 5-sphere. We thus cannot expect shrinks and collapses, even combined, to detect all contractible complexes, but they still provide useful heuristics towards this goal (e.g. [2]) and can even be sufficient in specific situations [13]. Actually, it is always possible to reduce a contractible simplicial complex to a point if we allow another homotopy preserving operation: the anti-collapse (the reverse operation of collapse) [9] but, of course, undecidability of contractibility implies that the length of the sequence is not bounded.

Contributions. A shrinkable simplicial complex is clearly contractible and the converse is not true because of the above undecidability result. We first present a simple shrink-irreducible contractible simplicial complex with 7 vertices. This simple complex is interesting in its own right and it inspired the proof of our main result, which is that it is NP-complete to decide whether a given (three-dimensional) simplicial complex is shrinkable. Our proof uses a reduction from 3-SAT similarly as in Tancer's NP-completeness proof of collapsibility [23] but, noticeably, our gadgets are much smaller than those used for collapsibility.

Our NP-completeness result on shrinkability together with Tancer's analog on collapsibility naturally raises the question of whether it is also NP-complete to decide if a given simplicial complex can be reduced to a single vertex by a sequence *combining* admissible edge contractions and admissible simplex collapses. In this direction, we present a contractible simplicial complex with 12 vertices that is irreducible for both shrinks and collapses.

2 Preliminaries

In this paper, simplicial complexes are *abstract* and their elements are (abstract) simplices, that is, finite non-empty collections of vertices. We can associate to every abstract simplicial complex a *geometric realization* that maps every abstract simplex to a geometric simplex of the same dimension. The union of the geometric simplices forms the *underlying space* of the complex.

As mentioned in the introduction, given a simplicial complex, we are interested in operations that preserve the homotopy type of the underlying space. One of these operations is the shrink, which is the contraction of an admissible edge, also called shrinkable edge. Below, we give a useful characterization of shrinkable edges in terms of blockers. Let $\mathcal{K}$ be a simplicial complex and recall that a *face* of a simplex is a non-empty subset of the simplex. The face is *proper* if it is distinct from the simplex.

Definition 1. *A* blocker *of $\mathcal{K}$ is a simplex that does not belong to $\mathcal{K}$ but whose proper faces all belong to $\mathcal{K}$.*

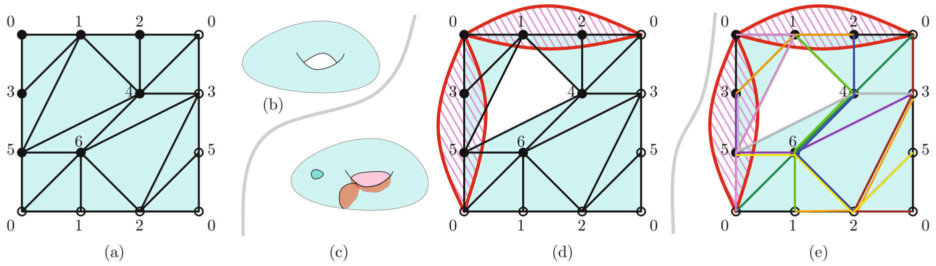


Fig. 1. (a) triangulation of the torus with 7 vertices, (d) a contractible non-shrinkable simplicial complex and (b,c) an embedding of their underlying spaces in $\mathbb{R}^3$. (e) highlights 8 blockers (015, 023, 123, 146, 246, 256, 345, 256) that suffice to cover all edges.

A blocker is also sometimes called a *missing face* [18], a *minimal non-face* [13], or a *simplicial hole* [15].

Lemma 1 ([13]). *An edge ab of $\mathcal{K}$ is shrinkable if and only if ab is not contained in any blocker of $\mathcal{K}$.*

Note that one of the direction is straightforward: if σ is a blocker containing ab, then $\sigma \setminus \{a,b\} \in \text{Link}\,(a) \cap \text{Link}\,(b)$ but $\sigma \setminus \{a,b\} \notin \text{Link}\,(ab)$.

As we contract shrinkable edges, blockers may appear or disappear and therefore edges may become non-shrinkable or shrinkable. For instance, consider the simplicial complex $\mathcal{L} = \{a, b, c, d, ab, bc, cd, da\}$ whose edges form a 4-circuit and the cone $\mathcal{K}$ on $\mathcal{L}$ with apex w, that is, the set of simplices of the form $\{w\} \cup \sigma$ where $\sigma \in \mathcal{L}$. The complex $\mathcal{K}$ does not contain any blocker and therefore all edges are shrinkable. Note however that the contraction of edge ab creates a blocker which disappears as we contract wa. Hence, as we simplify the complex, an edge that used to be shrinkable (or not) may change its status several times later on during the course of the simplification. Interestingly, the only blockers we need to consider in the paper are triangles.

3 A Simple Non-shrinkable Contractible Simplicial Complex

To construct a contractible simplicial complex that is shrink-irreducible, we start with the triangulation of the torus with 7 vertices described in Fig. 1-(a,b) (Császár polyhedron). Notice that the vertices and edges of this triangulation form a complete graph. Thus, every triple of vertices forms a cycle in this graph, which may or may not bound a face.

We now modify the complex as follows. The idea is to add two triangles so that every (arbitrary) cycle on the modified torus is contractible and to remove a triangle so as to open the cavity; see Fig. 1-(c). Namely, we add triangles 012 and 035 and remove triangle 145; see Fig. 1-(d). The resulting complex is contractible because it is collapsible; indeed all edges and vertices inside the “square” and on the boundary of the (expanding) hole can be collapsed until the hole fills the entire square, then it only remains triangles 012 and 035, which can also be trivially collapsed into a single vertex.

To see that the resulting complex is shrink irreducible, note that every edge is incident to at most 3 triangles; indeed, every edge is incident to 2 triangles in the initial triangulation of the torus, and we only added two triangles, which do not share edges. On the other hand, every edge belongs to exactly 5 cycles of length 3 since the graph is complete on 7 vertices. Hence, every edge belongs to at least 2 blockers, which implies that no edge is shrinkable, by Lemma 1.

4 NP-completeness of Shrinkability

Theorem 1. *Given an abstract simplicial complex of dimension 3 whose underlying space is contractible, it is NP-complete to decide whether the complex can be reduced to a point by a sequence of admissible edge contractions.*

The proof is given in this section by reduction from 3-SAT. We show that any Boolean formula in 3-conjunctive normal form (3CNF) can be transformed, in polynomial time, to a contractible 3-dimensional simplicial complex, such that a satisfying assignment exists if and only if the complex is shrinkable.

4.1 Gadgets Design

In the following, the gadgets are defined as *abstract* simplicial complexes but, for clarity, we describe geometric realizations of these gadgets in $\mathbb{R}^3$. Then the gadgets are assembled by identifying one triangle of one gadget with a triangle of another; this operation preserves the blockers and thus the unshrinkability of edges. A shrinkable edge remains shrinkable if it does not belong to the identified triangles or if it was shrinkable in both gadgets.

Forward Gadget

Properties. The forward gadget has a special triangle with edges A, B, C such that A is the only shrinkable edge of the gadget and once A is contracted (thus identifying B and C) there is a sequence of shrinks that reduces the gadget to a single point.

Usage. By gluing the triangle ABC to a triangle of another construction, we enforce that A is contracted before B and C, thus preventing some sequences of shrinks.

Realization. Refer to Fig. 2. Start with four points a, b, x and y in convex position in $\mathbb{R}^3$ and consider the tetrahedron $abxy$. Split this tetrahedron in four by adding a point o in its interior. The result is a simplicial complex with 5 vertices, 10 edges, 10 triangles and 4 tetrahedra. We then remove the 4 tetrahedra,

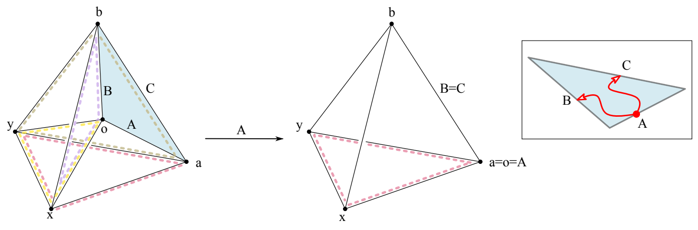


Fig. 2. Left: The forward gadget with triangle ABC in blue. Its 1-skeleton is the complete graph with vertices o, a, b, x and y and its (dotted) blockers axy, oxy, oxb, ayb are the triangles that have been collapsed. Middle: Contracting edge A produces a complex with a unique blocker, axy. Right: Schematic representation of the gadget.

by applying four triangle collapses. The first three collapses dig a gallery starting at triangle axy by successively removing the pair of simplices $(axy, axyo)$, $(oxy, oxyb)$, $(oxb, oxba)$. The fourth collapse removes the pair $(ayb, oayb)$. The obtained simplicial complex has 5 vertices, 10 edges, 6 triangles: 2 triangles of the initial tetrahedron (axb and xyb) and 4 triangles incident to o (oab, oax, oay and oyb). Notice that as we collapse these pairs of simplices (σ, Σ), the triangle σ becomes a blocker. Thus, the resulting simplicial complex has a unique blocker-free edge $A = oa$. Let $B = ob$ and $C = ab$. If A is contracted, the resulting complex contains the triangles axb, xyb, oyb, thus any of the edges incident to b can be shrunk, which reduces the complex to a triangle, which is shrinkable.

Freezer Gadget

Properties. The freezer gadget has a special triangle with edges A, B, C such that A and B are the only shrinkable edges of the gadget, and once A or B is contracted (identifying the other with C), there is a sequence of shrinks that reduce the gadget to a single point.

Usage. By gluing the triangle ABC to a triangle of another construction, we enforce that C is non-shrinkable (or frozen) until either A or B is contracted; such a contraction identifies C with the uncontracted remaining edge (B or A).

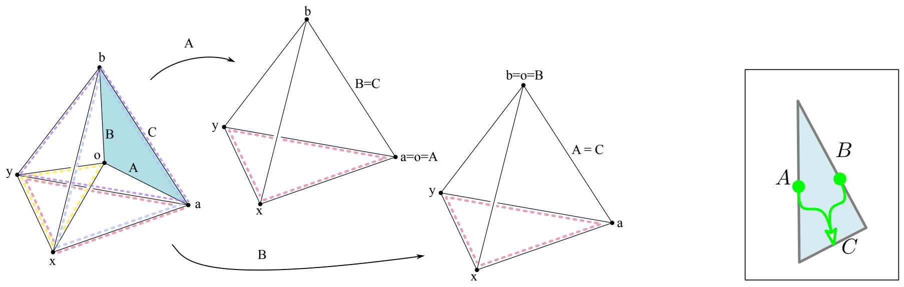


Fig. 3. The freezer gadget. Left: realization. Middle: contraction of edge A or B. Right: schematic representation.

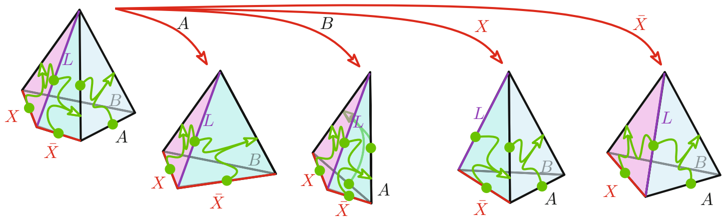


Fig. 4. The variable gadget: realization (left) and various edge contractions

Realization. Refer to Fig. 3. We start with the same construction as for the forward gadget except that instead of collapsing the pair $(oxb, oxba)$, we collapse the pair $(xab, oxab)$. The list of blockers thus created is axy, oxy, xab, ayb, and the resulting complex contains only 1 triangle of the initial tetrahedron (xyb) and 5 triangles incident to o (oab, oax, oay, oyb and oxb). The result is a simplicial complex with exactly two blocker-free edges, A and B. Similarly as for the forward gadget, once edge A or B is contracted, the resulting complex is shrinkable.

Variable Gadget

Properties. The variable gadget associated to a variable x has three special edges: $X, \bar{X}$ and L (lock). At the beginning X and $\bar{X}$ are shrinkable edges. When X or $\bar{X}$ has been contracted, the other one is not shrinkable before L and there is a sequence of shrinks that reduces the gadget to a single point.

Usage. Given a truth assignment, true (resp. false), for variable x, the edge X (resp. $\bar{X}$) of the associated gadget is contracted before the other edge $\bar{X}$ (resp. X). Gluing the lock edge to some key edges (see the clause gadgets), we ensure that once an assignment is chosen for the variable, the other edge, $\bar{X}$ (resp. X), cannot be contracted unless all the keys needed to open the lock have been released (i.e. all the blockers passing through L have been removed).

Realization. Refer to Figure 4. We consider the four triangles of a squared-base pyramid. From a vertex of the base, X and $\bar{X}$ are the incident edges on the base and L is the third incident edge on the pyramid. We glue three freezer gadgets onto three triangles incident to the apex, as shown in Figure 4, to ensure that the 3 edges that are incident to the apex and distinct from L are contracted after L, and that the edges on the base remain shrinkable. Contracting any edge on the base transforms the base into a blocker and L remains the only shrinkable edge, ensuring that L will be shrunk before one of X or $\bar{X}$.

Two-Clause Gadget

Properties. The two-clause gadget has three special edges: two literals V and W and a key K. We require that the key is not contracted before one of the

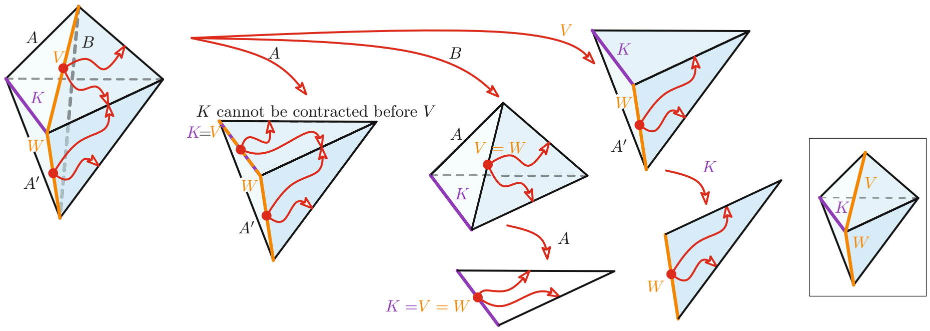


Fig. 5. The two-clause gadget. Left: realization. Middle: various edge contractions. Right: schematic representation.

two literals. Namely, at the beginning V and W are shrinkable edges and K is not shrinkable. K cannot be contracted before one of V or W and there are sequences of shrinks that contract any non-empty subset of $\{V, W\}$ before K.

Usage. Gluing the key edge to a lock edge of a variable gadget ensures that the lock will not be contracted before the key has been released (i.e. K has become shrinkable).

Realization. Consider a horizontal triangle and a vertical edge B that pierces it. Each of the triangle edges together with the piercing edge define a tetrahedron, and we consider the simplicial complex defined by these three tetrahedra; see Fig. 5. The initial triangle we considered is not part of this complex and is thus a blocker. We place K on the blocker and take for V and W the edges incident to an endpoint of K not in the blocker. Finally, we glue a forward gadget to the face incident to V but not to K and another one for W, symmetrically.

Let A (resp. A') be the third edge of the triangle defined by edges V (resp. W) and K, and recall that B is the central edge. The only edges that are initially shrinkable are A, A', B, V, and W. Contracting A identifies V and K, ensuring that K will not be contracted before V. Contracting A' is similar to contracting A (exchanging V and W). Contracting B identifies V and W, and yields a configuration where $A{=}\ A'$ and $V = W$ are the only shrinkable edges; then contracting A identifies V, W, and K ensuring that K will not be contracted before V nor W. Thus, K cannot be contracted (strictly) before one of V or W. Finally, we can contract V then K, which yields a forward gadget whose only contractible edge is W. Hence possible ordering to shrink V, W, and K are VWK, WVK, VKW, or WKV.

Three-Clause Gadget

Properties. The three-clause gadget has four special edges: three literals U, V, and W and a key K. We enforce that the key is not contracted before one of the three literals. Namely, at the beginning U, V, and W are shrinkable and K is not. K cannot be contracted before one of U, V, or W and there is a sequence of shrinks that contracts any non-empty subset of $\{U, V, W\}$ before K.

Realization. Refer to Fig. 6. The realization is done by simple association of two two-clause gadgets, gluing the key of one clause on one literal of the other, as described in Fig. 6-left. We furthermore add the two triangles defined by KV and KW (note that the triangle KU already belongs to the gadget). These two extra triangles will be needed when gluing gadgets together. By construction, our two glued two-clause gadgets satisfies the properties we require for the three-clause gadgets. Adding the two triangles KV and KW does not invalidate these properties. Indeed, let A be the third edge of triangle KV; the addition of A has created a blocker (in red in Fig. 6-right). Thus A cannot be contracted and it does not block the contraction of U, V, W, or K. Once V or K is contracted, A is identified with K or V and this extra triangle disappears. Thus, the gadget keeps its properties with these two additional triangles.

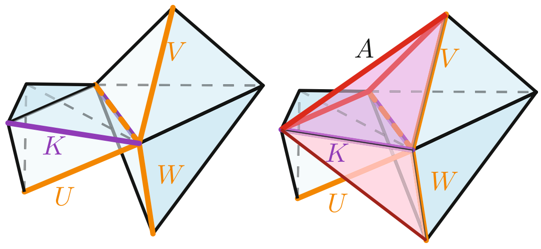


Fig. 6. Left: two glued two-clause gadgets. Right: The three-clause gadget.

4.2 Wrap up

3-SAT and Shrinkability. Given a 3CNF Boolean formula, we build a three-clause gadget per clause and a variable gadget per variable. The literal edge of each clause gadget is glued to the relevant edge of the variable gadget, that is, a literal x (resp. $\neg x$) is glued to the edge X (resp. $\bar{X}$) of the variable gadget associated to x. The lock edge of each variable gadget is glued to the key edge of each clause it appears in. We assume that the obtained complex is connected, otherwise the 3-SAT problem can be decomposed into independent subproblems, which can be solved separately.

Notice that a pair of edges key/literal forms a triangle in the three-clause gadget and that the pair of edges lock/X (or lock/$\bar{X}$) also forms a triangle in the variable gadget. Thus, the third edges of these triangles are also glued. Actually, the effect of this construction is that the edges K and L of all gadgets are identified and become a single edge in the final complex. By construction, the complex is contractible since each gadget is contractible and we are gluing them by triangles that all have a common edge, K.

Our construction is 3 dimensional, thus it can be embedded in $\mathbb{R}^7$ using general position for the vertices.

From a Truth Assignment to a Sequence of Shrinks. For every variable, if it is assigned true (resp. false), edge X (resp. $\bar{X}$) is contracted in the associated

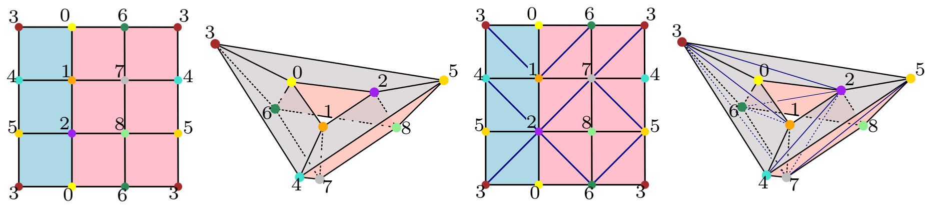


Fig. 7. Triangulation of a torus with 9 vertices. From left to right: the torus represented as a square with opposite edges identified and its embedding in $\mathbb{R}^3$ as a polyhedron with 9 trapezoidal faces; a non-shrinkable triangulation; and its embedding.

gadget. These edges are identified to literal edges of the clause gadgets, so their contractions make edge K shrinkable from the point of view of all clause gadgets and K can thus be contracted. All edges corresponding to the other values of the variable gadgets become shrinkable and the complex can be contracted to a point.

From a Sequence of Shrinks to a Truth Assignment. For every variable gadget, if edge X (resp. $\bar{X}$) is contracted before $\bar{X}$ (resp. X), we assign true (resp. false) to the variable associated to the gadget. All clauses are satisfied by this assignment since K cannot be contracted before all clause gadgets have one of their literal edge contracted.

5 A Non-shrinkable Bing's House

In this section, we construct a contractible simplicial complex which is irreducible, both for shrinks and for collapses.[3] The idea is to triangulate carefully Bing's house, in such a way that no edge is shrinkable. Bing's house has two rooms, one above the other. The only access to the upper room is through an underground tunnel that passes through the lower room and the only access to the lower room is through a chimney that passes through the upper room; see Fig. 9-middle.

To triangulate the lower room (and the tunnel), we start with a triangulation of the torus with 9 vertices presented in Fig. 7. We now proceed to two successive alterations of the complex; see Fig. 8. First, we create a room inside the torus, by adding the two (pink hashed) triangles: 036 and 236 and removing triangle 013; the two added triangles delimit the room inside the torus and the removed triangle provides access to the room from outside. We then build a tunnel through the middle of the room by removing two triangles: 023 and 026 and by adding the (blue hashed) triangle 012.

To see that the resulting complex is shrink-irreducible, notice that the triangulation of the torus is shrink-irreducible to start with. During the modification,

[3] You can actually build your own 3D model, see Appendix of hal.inria.fr/hal-01015747.

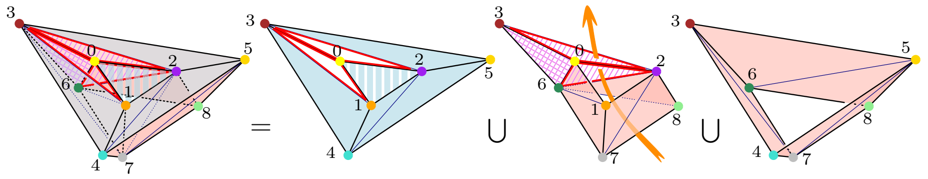


Fig. 8. Triangulation of the lower room and underground tunnel using 18 triangles: 5 (blue) triangles are coplanar and form the roof, 7 triangles (5 pink and 2 hashed) bound the tunnel under the roof and 6 (pink) triangles lie on the outer walls of the room. The arrow indicates a passage through the tunnel from the underground entrance 678 to the roof exit 023. The red loops indicate the 3 triangles removed from the torus.

the only way an edge may become non-shrinkable is if there are more triangles incident to that edge that are added than the ones that are removed. The only edges that fulfill that condition are 36 and 12 and one can check that they are still covered by blockers at the end: 361 and 123 respectively. Similarly, one can check that the room has only three collapsible edges, namely 02, 03 and 13, all lying on the roof. Indeed, no edges are collapsible in the initial triangulation of the torus and an edge is collapsible in the final complex if and only if the number of added triangles incident to that edge is one less than the number of removed triangles. The only edges with this property are 02, 03 and 13.

To finish our construction of Bing's house, we consider a copy of the lower room, which we place above the original one; see Fig. 9. Renaming vertices x by x' in the copy, this boils down to the following identifications: vertices 0 with 0',

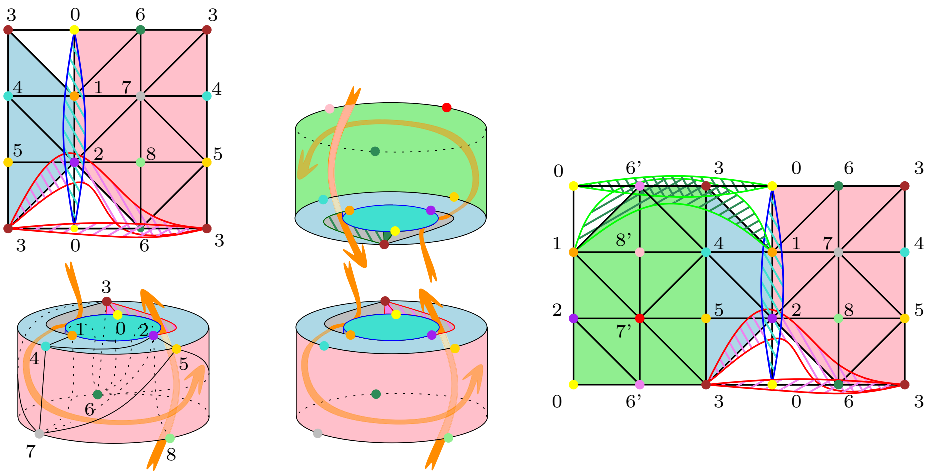


Fig. 9. Building the Bing's house. Left: triangulation of the lower room and schematic representation. Middle: the two rooms one above the other with the four arrows representing the way through the underground tunnel to the upper room and through the chimney to the lower room. Right. Triangulation of Bing's house.

1 with 2', 2 with 1', 3 with 3', 4 with 5' and 5 with 4'. The result is a simplicial complex with 12 vertices which is still shrink-irreducible but in which no edge is collapsible anymore; see Fig. 9.

Acknowledgements. This work was initiated during the 12th *INRIA–McGill–Victoria Workshop on Computational Geometry* at the Bellairs Research Institute. The authors wish to thank all the participants for creating a pleasant and stimulating atmosphere. The authors thank Uli Wagner for fruitful discussions and interesting references.

References

1. Attali, D., Lieutier, A., Salinas, D.: Vietoris-Rips complexes also provide topologically correct reconstructions of sampled shapes. Computational Geometry: Theory and Applications 46, 448–465 (2012), doi:10.1016/j.comgeo.2012.02.009
2. Attali, D., Lieutier, A., Salinas, D.: Collapsing Rips complexes. In: EuroCG 2013 (2013), `http://www.ibr.cs.tu-bs.de/alg/eurocg13/booklet_eurocg13.pdf`
3. Barnette, D.W., Edelson, A.: All orientable 2-manifolds have finitely many minimal triangulations. Israel Journal of Mathematics 62(1), 90–98 (1988), doi:10.1007/BF02767355
4. Barnette, D.W., Edelson, A.: All 2-manifolds have finitely many minimal triangulations. Israel Journal of Mathematics 67(1), 123–128 (1989), doi:10.1007/BF02764905
5. Bing, R.H.: Some Aspects of the Topology of 3-Manifolds Related to the Poincaré Conjecture. Lectures on Modern Mathematics, vol. II. Wiley, New York (1964)
6. Carlsson, G., de Silva, V.: Topological approximation by small simplicial complexes. Technical report, Mischaikow., Wanner, T. (2003), `http://math.stanford.edu/research/comptop/preprints/delaunay.pdf`
7. Carlsson, G., Ishkhanov, T., De Silva, V., Zomorodian, A.: On the local behavior of spaces of natural images. Int. J. of Computer Vision 76(1), 1–12 (2008), doi:10.1007/s11263-007-0056-x
8. Chazal, F., Cohen-Steiner, D., Lieutier, A.: A sampling theory for compact sets in Euclidean space. Discrete & Computational Geometry 41(3), 461–479 (2009), doi:10.1007/s00454-009-9144-8
9. Cohen, M.M.: A course in simple-homotopy theory. Graduate Texts in Mathematics, vol. 10. Springer (1973), doi:10.1007/978-1-4684-9372-6
10. De Silva, V.: A weak characterisation of the Delaunay triangulation. Geometriae Dedicata 135(1), 39–64 (2008), doi:10.1007/s10711-008-9261-1
11. De Silva, V., Carlsson, G.: Topological estimation using witness complexes. In: Proceedings of the First Eurographics conference on Point-Based Graphics, pp. 157–166. Eurographics Association (2004), doi:10.2312/SPBG/SPBG04/157-166
12. Dey, T.K., Edelsbrunner, H., Guha, S., Nekhayev, D.V.: Topology preserving edge contraction. Publ. Inst. Math (Beograd) (N.S.) 66, 23–45 (1999), `http://www.cs.duke.edu/ edels/Papers/1999-J-03-TopologyPreservingContraction.pdf`
13. Ehrenborg, R., Hetyei, G.: The topology of the independence complex. European Journal of Combinatorics 27(6), 906–923 (2006), doi:10.1016/j.ejc.2005.04.010

14. Garland, M., Heckbert, P.S.: Surface simplification using quadric error metrics. In: SIGGRAPH 1997 Proc., pp. 209–216 (1997), doi:10.1145/258734.258849
15. Goaoc, X.: Transversal Helly numbers, pinning theorems and projections of simplicial complexes, Habilitation thesis, Université Nancy (January 2011), http://tel.archives-ouvertes.fr/tel-00650204
16. Hoppe, H.: Progressive meshes. In: SIGGRAPH 1996 Proc., pp. 99–108 (1996), doi:10.1145/237170.237216
17. Lawrencenko, S.: Irreducible triangulations of the torus. Journal of Mathematical Sciences 51, 2537–2543 (1990), http://www.lawrencenko.ru/files/itott_en.pdf
18. Melikhov, S.: Combinatorics of embeddings. Research Report arXiv (2011), http://arxiv.org/abs/1103.5457
19. Nevo, E.: Higher minors and Van Kampen's obstruction. Mathematica Scandinavica 101(2), 161–176 (2006), http://ojs.statsbiblioteket.dk/index.php/math/article/view/15037
20. Niyogi, P., Smale, S., Weinberger, S.: Finding the Homology of Submanifolds with High Confidence from Random Samples. Discrete & Computational Geometry 39(1-3), 419–441 (2008), doi:10.1007/s00454-008-9053-2
21. Popović, J., Hoppe, H.: Progressive simplicial complexes. In: SIGGRAPH 1997 Proc., pp. 217–224 (1997), doi:10.1145/258734.258852
22. Sulanke, T.: Irreducible triangulations of low genus surfaces. Research Report arXiv (2006), http://arxiv.org/abs/math/0606690
23. Tancer, M.: Recognition of collapsible complexes is NP-complete. Research Report arXiv (2012), http://arxiv.org/abs/1211.6254
24. Volodin, I.A., Kuznetsov, V.E., Fomenko, A.T.: The problem of discriminating algorithmically the standard three-dimensional sphere. Russian Mathematical Surveys 29(5), 71 (1974), doi:10.1070/RM1974v029n05ABEH001296

Improved Approximation Algorithms for Box Contact Representations*

Michael A. Bekos[1], Thomas C. van Dijk[2], Martin Fink[2,5], Philipp Kindermann[2], Stephen Kobourov[3], Sergey Pupyrev[3,4], Joachim Spoerhase[2], and Alexander Wolff[2,**]

[1] Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, Germany
[2] Lehrstuhl für Informatik I, Universität Würzburg, Germany
[3] Department of Computer Science, University of Arizona, USA
[4] Institute of Mathematics and Computer Science, Ural Federal University, Russia
[5] Department of Computer Sicence, University of California, Santa Barbara, USA

Abstract. We study the following geometric representation problem: Given a graph whose vertices correspond to axis-aligned rectangles with fixed dimensions, arrange the rectangles without overlaps in the plane such that two rectangles touch if the graph contains an edge between them. This problem is called CONTACT REPRESENTATION OF WORD NETWORKS (CROWN) since it formalizes the geometric problem behind drawing word clouds in which semantically related words are close to each other. CROWN is known to be NP-hard, and there are approximation algorithms for certain graph classes for the optimization version, MAX-CROWN, in which realizing each desired adjacency yields a certain profit.

We show that the problem is APX-complete on bipartite graphs of bounded maximum degree. We present the first $O(1)$-approximation algorithm for the general case, when the input is a complete weighted graph, and for the bipartite case. Since the subgraph of realized adjacencies is necessarily planar, we consider several planar graph classes (stars, trees, outerplanar, and planar graphs), improving upon the known results. For some graph classes, we also describe improvements in the unweighted case, where each adjacency yields the same profit.

1 Introduction

In the last few years, word clouds have become a standard tool for abstracting, visualizing, and comparing text documents. For example, word clouds were used in 2008 to contrast the speeches of the U.S. presidential candidates. More recently, the German media used them to visualize the newly signed coalition agreement and to compare it to a similar agreement from 2009. A word cloud of a given document consists of the most important (or most frequent) words in that document. Each word is printed in a given

* The work of M. A. Bekos is implemented within the framework of the Action "Supporting Postdoctoral Researchers" of the Operational Program "Education and Lifelong Learning" (Action's Beneficiary: General Secretariat for Research and Technology), and is co-financed by the European Social Fund (ESF) and the Greek State. Ph. Kindermann and A. Wolff acknowledge support by the ESF EuroGIGA project GraDR. S. Kobourov and S. Pupyrev are supported by NSF grants CCF-1115971 and DEB 1053573.

font and scaled by a factor roughly proportional to its importance (the same is done with the names of towns and cities on geographic maps, for example). The printed words are arranged without overlap and tightly packed into some shape (usually a rectangle). Tag clouds look similar; they consist of keyword metadata (tags) that have been attributed to resources in some collection such as web pages or photos.

Wordle [23] is a popular tool for drawing word or tag clouds. The Wordle website allows users to upload a list of words and, for each word, its relative importance. The user can further select font, color scheme, and decide whether all words must be placed horizontally or whether words can also be placed vertically. The tool then computes a placement of the words, each scaled according to its importance, such that no two words overlap. Generally, the drawings are very compact and aesthetically appealing.

In the automated analysis of text one is usually not just interested in the most important words and their frequencies, but also in the connections between these words. For example, if a pair of words often appears together in a sentence, then this is often seen as evidence that this pair of words is linked semantically [17]. In this case, it makes sense to place the two words close to each other in the word cloud that visualizes the given text. This is captured by an input graph $G = (V, E)$ of desired contacts. We are also given, for each vertex $v \in V$, the dimensions (but not the position) of a *box* B_v, that is, an axis-aligned rectangle. We denote the height and width of B_v by $h(B_v)$ and $w(B_v)$, respectively, or, more briefly, by $h(v)$ and $w(v)$. For each edge $e = (u, v)$ of G, we are given a positive number $p(e) = p(u, v)$, that corresponds to the *profit* of e. For ease of notation, we set $p(u, v) = 0$ for any non-edge $(u, v) \in V^2 \setminus E$ of G.

Given a box B and a point q in the plane, let $B(q)$ be a placement of B with lower left corner q. A *representation* of G is a map $\lambda : V \to \mathbb{R}^2$ such that for any two vertices $u \neq v$, it holds that $B_u(\lambda(u))$ and $B_v(\lambda(v))$ are interior-disjoint. Boxes may *touch*, that is, their boundaries may intersect. If the intersection is non-degenerate, that is, a line segment of positive length, we say that the boxes are *in contact*. We say that a representation λ *realizes* an edge (u, v) of G if boxes $B_u(\lambda(u))$ and $B_v(\lambda(v))$ are in contact.

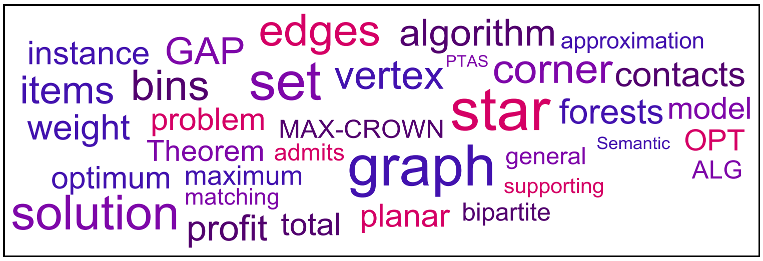


Fig. 1. Semantics-preserving word cloud for the 35 most "important" words in this paper. Following the text processing pipeline of Barth et al. [2], these are the words ranked highest by LexRank [11], after removal of stop words such as "the". The edge profits are proportional to the relative frequency with which the words occur in the same sentences. The layout algorithm of Barth et al. [2] first extracts a heavy star forest from the weighted input graph as in Theorem 6 and then applies a force-directed post-processing.

This yields the problem *Contact Representation of Word Networks* (CROWN): Given an edge-weighted graph G whose vertices correspond to boxes, find a representation of G with the vertex boxes such that every edge of G is realized. In this paper, we study the optimization version of CROWN, MAX-CROWN, where the aim is to maximize the total profit (that is, the sum of the weights) of the realized edges. We also consider the unweighted version of the problem, where all desired contacts yield a profit of 1.

Previous Work. Barth et al. [1] introduced MAX-CROWN and showed that the problem is strongly NP-hard even for trees and weakly NP-hard even for stars. They presented an exact algorithm for cycles and approximation algorithms for stars, trees, planar graphs, and graphs of constant maximum degree; see the first column of Table 1. Some of their solutions use an approximation algorithm with ratio $\alpha = e/(e-1) \approx 1.58$ [13] for the GENERALIZED ASSIGNMENT PROBLEM (GAP): Given a set of bins with capacity constraints and a set of items that possibly have different sizes and values for each bin, pack a maximum-valued subset of items into the bins. The problem is APX-hard [6].

MAX-CROWN is related to finding *rectangle representations* of graphs, where vertices are represented by axis-aligned rectangles with non-intersecting interiors and edges correspond to rectangles with a common boundary of non-zero length. Every graph that can be represented this way is planar and every triangle in such a graph is a facial triangle. These two conditions are also sufficient to guarantee a rectangle representation [5]. Rectangle representations play an important role in VLSI layout, cartography, and architecture (floor planning). In a recent survey, Felsner [12] reviews many rectangulation variants. Several interesting problems arise when the rectangles in the representation are restricted. Eppstein et al. [10] consider rectangle representations which can realize any given area-requirement on the rectangles, so-called *area-preserving rectangular cartograms*, which were introduced by Raisz [22] already in the 1930s. Unlike cartograms, in our setting there is no inherent geography, and hence, words can be positioned anywhere. Moreover, each word has fixed dimensions enforced by its importance in the input text, rather than just fixed area. Nöllenburg et al. [20] recently considered a variant where the edge weights prescribe the length of the desired contacts.

Finally, the problem of computing semantics-aware word clouds is related to classic graph layout problems, where the goal is to draw graphs so that vertex labels are readable and Euclidean distances between pairs of vertices are proportional to the underlying graph distance between them. Typically, however, vertices are treated as points and label overlap removal is a post-processing step [9,15]. Most tag cloud and word cloud tools such as Wordle [23] do not show the semantic relationships between words, but force-directed graph layout heuristics are sometimes used to add such functionality [2,8,21,24].

Our Contribution. Known results and our contributions to MAX-CROWN are shown in Table 1. Note that the results of Barth et al. [1] in column 1 are simply based on existing decompositions of the respective graph classes into star forests or cycles.

Our results rely on a variety of algorithmic tools. First, we devise sophisticated decompositions of the input graphs into heterogeneous classes of subgraphs, which also requires a more general combination method than that of Barth et al. Second, we use randomization to obtain a simple constant-factor approximation for general

Table 1. Previously known and new results for the unweighted and weighted versions of MAX-CROWN (for $\alpha \approx 1.58$ and any $\varepsilon > 0$)

	Weighted			Unweighted	
Graph class	Ratio [1]	Ratio [new]	Ref.	Ratio	Ref.
cycle, path	1				
star	α	$1+\varepsilon$	Thm. 1		
tree	2α	$2+\varepsilon$	Thm. 1	2	Thm. 7
	NP-hard				
max-degree Δ	$\lfloor(\Delta+1)/2\rfloor$				
planar max-deg. Δ				$1+\varepsilon$	Thm. 8
outerplanar		$3+\varepsilon$	Thm. 3		
planar	5α	$5+\varepsilon$	Thm. 1		
bipartite		$16\alpha/3$ (≈ 8.4)	Thm. 4		
		APX-complete	Thm. 2		
general		$32\alpha/3$ (≈ 16.9; rand.)	Thm. 5	$5+16\alpha/3$	Thm. 9
		$40\alpha/3$ (≈ 21.1; det.)	Thm. 6		

weighted graphs. Previously, such a result was not even known for unweighted bipartite graphs. Third, to obtain an improved algorithm for the unweighted case, we prove a lower bound on the size of a matching in a planar graph of high average degree. Fourth, we use a planar separator result of Frederickson [14] to obtain a polynomial-time approximation scheme (PTAS) for degree-bounded planar graphs.

We start our paper with basic results on simple graph classes and prove that MAX-CROWN is APX-complete on bipartite graphs of maximum degree 9 (Section 2). Then, we tackle weighted graphs (Section 3). We obtain improved results for several unweighted graph classes (Section 4). Finally, we list some open problems (Section 5).

Model. As in most work on rectangle contact representations, we do not count *point contacts*, that is, we consider two boxes in contact only if their intersection is a line segment of positive length. Hence, the contact graph of the boxes is planar. Our algorithms can easily be modified to guarantee $O(1)$-approximations also in the model that allows and rewards point contacts [3]. We allow words only to be placed horizontally.

Runtimes. Most of our algorithms involve approximating a number of GAP instances as a subroutine, using either the PTAS [4] if the number of bins is constant or the approximation algorithm of Fleischer et al. [13] for general instances. Because of this, the runtime of our algorithms consists mostly of approximating GAP instances. Both algorithms to approximate GAP instances solve linear programs, so we refrain from explicitly stating the runtime of these algorithms.

For practical purposes, one can use a purely combinatorial approach for approximating GAP [7], which utilizes an algorithm for the KNAPSACK problem as a subroutine. The algorithm translates into a 3-approximation for GAP running in $O(NM)$ time (or a $(2+\varepsilon)$-approximation running in $O(MN\log 1/\varepsilon + M/\varepsilon^4)$ time), where N is the number of items and M is the number of bins. In our setting, the simple 3-approximation implies a randomized 32-approximation (or a deterministic 40-approximation) algorithm with running time $O(|V|^2)$ for MAX-CROWN on general weighted graphs.

2 Some Basic Results

We first present two technical lemmas that will help us prove our main results on weighted and unweighted MAX-CROWN. The second lemma immediately improves the results of Barth et al. [1] for stars, trees, and planar graphs. Finally, we prove APX-completeness of MAX-CROWN on bipartite graphs of bounded maximum degree.

2.1 A Combination Lemma

Several of our algorithms cover the input graph with subgraphs that belong to graph classes for which the MAX-CROWN problem is known to admit good approximations. The following lemma allows us to combine the solutions for the subgraphs. We say that a graph $G = (V,E)$ is *covered* by graphs $G_1 = (V,E_1),\dots,G_k = (V,E_k)$ if $E = E_1 \cup \dots \cup E_k$.

Lemma 1. *Let graph $G = (V,E)$ be covered by graphs $G_1,G_2,\dots,G_k$. If, for $i = 1,2,\dots,k$, weighted* MAX-CROWN *on graph G_i admits an α_i-approximation, then weighted* MAX-CROWN *on G admits a $\left(\sum_{i=1}^{k}\alpha_i\right)$-approximation.*

Proof. Our algorithm works as follows. For $i = 1,\dots,k$, we apply the α_i-approximation algorithm to G_i and report the result with the largest profit as the result for G. We show that this algorithm has the claimed performance guarantee. For the graphs $G,G_1,\dots,G_k$, let $\mathrm{OPT},\mathrm{OPT}_1,\dots,\mathrm{OPT}_k$ be the optimum profits and let $\mathrm{ALG},\mathrm{ALG}_1,\dots,\mathrm{ALG}_k$ be the profits of the approximate solutions. By definition, $\mathrm{ALG}_i \geq \mathrm{OPT}_i/\alpha_i$ for $i = 1,\dots,k$. Moreover, $\mathrm{OPT} \leq \sum_{i=1}^{k}\mathrm{OPT}_i$ because the edges of G are covered by the edges of $G_1,\dots,G_k$. Assume, w.l.o.g., that $\mathrm{OPT}_1/\alpha_1 = \max_i(\mathrm{OPT}_i/\alpha_i)$. Then

$$\mathrm{ALG} = \mathrm{ALG}_1 \geq \frac{\mathrm{OPT}_1}{\alpha_1} \geq \frac{\sum_{i=1}^{k}\mathrm{OPT}_i}{\sum_{i=1}^{k}\alpha_i} \geq \frac{\mathrm{OPT}}{\sum_{i=1}^{k}\alpha_i}. \qquad \square$$

2.2 Improvement on Existing Approximation Algorithms

Lemma 2 ([4]). *For any $\varepsilon > 0$, there is a $(1+\varepsilon)$-approximation algorithm for* GAP *with a constant number of bins. The algorithm takes $n^{O(1/\varepsilon)}$ time.* □

Using Lemmas 1 and 2, we improve the approximation algorithms of Barth et al. [1].

Theorem 1. *Weighted* MAX-CROWN *admits a $(1+\varepsilon)$-approximation algorithm on stars, a $(2+\varepsilon)$-approximation algorithm on trees, and a $(5+\varepsilon)$-approximation algorithm on planar graphs.*

Proof. By Lemma 1, the claim for stars implies the other two claims since a tree can be covered by two star forests and a planar graph can be covered by five star forests in polynomial time [16]. We now show that we can use Lemma 2 to get a PTAS for stars. Here, we give the PTAS for the model with point contacts; in the full version [3], we show how to handle the model without point contacts.

Let u be the center vertex of the star. We create eight bins: four *corner bins* u_1^c, u_2^c, u_3^c, and u_4^c modeling adjacencies on the four corners of the box u, two *horizontal bins* u_1^h and u_2^h modeling adjacencies on the top and bottom side of u, and two *vertical bins* u_1^v and u_2^v modeling adjacencies on the left and right side of u. The capacity of the corner bins is 1, the capacity of the horizontal bins is the width $w(u)$ of u, and the capacity of the vertical bins is the height $h(u)$ of u. Next, we introduce an item $i(v)$ for any leaf vertex v of the star. The size of $i(v)$ is 1 in any corner bin, $w(v)$ in any horizontal bin, and $h(v)$ in any vertical bin. The profit of $i(v)$ in any bin is the profit $p(u,v)$ of the edge (u,v).

Note that any feasible solution to the MAX-CROWN instance can be normalized so that any box that touches a corner of u has a point contact with u. Hence, the above is an approximation-preserving reduction from weighted MAX-CROWN on stars (with point contacts) to GAP. By Lemma 2, we obtain a PTAS. □

2.3 APX-Completeness

The proof for the following theorem is given in the full version [3].

Theorem 2. *Weighted* MAX-CROWN *is APX-complete even if the input graph is bipartite of maximum degree 9, each edge has profit 1, 2 or 3, and each vertex corresponds to a square of one out of three different sizes.*

3 The Weighted Case

In this section, we provide new approximation algorithms for more involved classes of (weighted) graphs than in the previous section. Recall that $\alpha = e/(e-1) \approx 1.58$. First, we give a $(3+\varepsilon)$-approximation for outerplanar graphs. Then, we present a $16\alpha/3$-approximation for bipartite graphs. For general graphs, we provide a simple randomized $32\alpha/3$-approximation and a deterministic $40\alpha/3$-approximation.

Theorem 3. *Weighted* MAX-CROWN *on outerplanar graphs admits a* $(3+\varepsilon)$*-approximation.*

Proof. It is known that the star arboricity of an outerplanar graph is 3, that is, it can be partitioned into at most three star forests [16]. Here we give a simple algorithm for finding such a partitioning.

Any outerplanar graph has degeneracy at most 2, that is, it has a vertex of degree at most 2. We prove that any outerplanar graph G can be partitioned into three star forests such that every vertex of G is the center of only one star. Clearly, it is sufficient to prove the claim for maximal outerplanar graphs in which all vertices have degree at least 2. We use induction on the number of vertices of G. The base of the induction corresponds to a 3-cycle for which the claim clearly holds. For the induction step, let v be a degree-2 vertex of G and let (v,u) and (v,w) be its incident edges. The graph $G-v$ is maximal outerplanar and thus, by induction hypothesis, it can be partitioned into star forests F_1, F_2, and F_3 such that u is the center of a star in F_1 and w is the center of a star in F_2. Now we can cover G with three star forests: we add (v,u) to F_1, we add (v,w) to F_2, and we create a new star centered at v in F_3.

Applying Lemma 1 and Theorem 1 to the star forests completes the proof. □

Theorem 4. *Weighted* MAX-CROWN *on bipartite graphs admits a* $16\alpha/3$-*approximation.*

Proof. Let $G=(V,E)$ be a bipartite input graph with $V=V_1 \dot\cup V_2$ and $E \subseteq V_1 \times V_2$. Using G, we build an instance of GAP as follows. For each vertex $u \in V_1$, we create eight bins $u_1^c, u_2^c, u_3^c, u_4^c, u_1^h, u_2^h, u_1^v, u_2^v$ and set the capacities exactly as we did for the star center in Theorem 1. Next, we add an item $i(v)$ for every vertex $v \in V_2$. The size of $i(v)$ is, again, 1 in any corner bin, $w(v)$ in any horizontal bin, and $h(v)$ in any vertical bin. For $u \in V_1$, the profit of $i(v)$ is $p(u,v)$ in any bin of u.

It is easy to see that solutions to the GAP instance are equivalent to word cloud solutions (with point contacts) in which the realized edges correspond to a forest of stars with all star centers being vertices of V_1. Hence, we can find an approximate solution of profit $\mathrm{ALG}_1' \geq \mathrm{OPT}_1'/\alpha$ where OPT_1' is the profit of an optimum solution (with point contacts) consisting of a star forest with centers in V_1.

We now show how to get a solution without point contacts. If the three bins on the top side of a vertex u (two corner bins and one horizontal bin) are not completely full, we can slightly move the boxes in the corners so that point contacts are avoided. Otherwise, we remove the lightest item from one of these bins. We treat the three bottommost bins analogously. Note that in both cases we only remove an item if all three bins are completely full. The resulting solution can be realized without point contacts. We do the same for the three left and three right bins and choose the heavier of the two solutions. It is easy to see that we lose at most $1/4$ of the profit for the star center u: Assume that the heaviest solution results from removing weight w_1 from one of the upper and weight w_2 from one of the lower bins. As we remove the lightest items only, the remaining weight from the upper and lower bins is at least $2(w_1+w_2)$. On the other hand, the weight in the two vertical at least w_1+w_2; otherwise, dropping everything from these vertical bins would be cheaper. Hence, we keep at least weight $3(w_1+w_2)$.

If we do so for all star centers, we get a solution with profit $\mathrm{ALG}_1 \geq 3/4 \cdot \mathrm{ALG}_1' \geq 3\,\mathrm{OPT}_1'/(4\alpha) \geq 3\,\mathrm{OPT}_1/(4\alpha)$ where OPT_1 is the profit of an optimum solution (without point contacts) consisting of a star forest with centers in V_1.

Similarly, we can find a solution of profit $\mathrm{ALG}_2 \geq 3\,\mathrm{OPT}_2/(4\alpha)$ with star centers in V_2, where OPT_2 is the maximum profit that a star forest with centers in V_2 can realize. Among the two solutions, we pick the one with larger profit $\mathrm{ALG} = \max\{\mathrm{ALG}_1, \mathrm{ALG}_2\}$.

Let $G^\star = (V, E^\star)$ be the contact graph realized by a fixed optimum solution, and let $\mathrm{OPT} = p(E^\star)$ be its total profit. We now show that $\mathrm{ALG} \geq 3\,\mathrm{OPT}/(16\alpha)$. As $G^\star$ is a planar bipartite graph, $|E^\star| \leq 2n-4$. Hence, we can decompose $E^\star$ into two forests H_1 and H_2 using a result of Nash-Williams [18]. We can further decompose H_1 into two star forests S_1 and S_1' in such a way that the star centers of S_1 are in V_1 and the star centers of S_1' are in V_2. Similarly, we decompose H_2 into a forest S_2 of stars with centers in V_1

and a forest S_2' of stars with centers in V_2. As we decomposed the optimum solution into four star forests, one of them—say S_1—has profit $p(S_1) \geq \mathrm{OPT}/4$. On the other hand, $\mathrm{OPT}_1 \geq p(S_1)$. Summing up, we get

$$\mathrm{ALG} \geq \mathrm{ALG}_1 \geq 3\,\mathrm{OPT}_1/(4\alpha) \geq 3p(S_1)/(4\alpha) \geq 3\,\mathrm{OPT}/(16\alpha). \qquad \square$$

Theorem 5. *Weighted* MAX-CROWN *on general graphs admits a randomized* $32\alpha/3$*-approximation.*

Proof. Let $G = (V,E)$ be the input graph and let OPT be the weight of a fixed optimum solution. Our algorithm works as follows. We first randomly partition the set of vertices into V_1 and $V_2 = V \setminus V_1$, that is, the probability that a vertex v is included in V_1 is $1/2$. Now we consider the bipartite graph $G' = (V_1 \dot\cup V_2, E')$ with $E' = \{(v_1,v_2) \in E \mid v_1 \in V_1 \text{ and } v_2 \in V_2\}$ that is induced by V_1 and V_2. By applying Theorem 4 on G', we can find a feasible solution for G with weight $\mathrm{ALG} \geq 3\,\mathrm{OPT}'/(16\alpha)$, where OPT' is the weight of an optimum solution for G'.

Any edge of the optimum solution is contained in G' with probability $1/2$. Let $\overline{\mathrm{OPT}}$ be the total weight of the edges of the optimum solution that are present in G'. Then, $E[\overline{\mathrm{OPT}}] = \mathrm{OPT}/2$. So, $E[\mathrm{ALG}] \geq 3E[\mathrm{OPT}']/(16\alpha) \geq 3E[\overline{\mathrm{OPT}}]/(16\alpha) = 3\,\mathrm{OPT}/(32\alpha)$. $\square$

Theorem 6. *Weighted* MAX-CROWN *on general graphs admits a* $40\alpha/3$*-approximation.*

Proof. Let $G = (V,E)$ be the input graph. As in the proof of Theorem 4, our algorithm constructs an instance of GAP based on G. The difference is that, *for every vertex* $v \in V$, we create *both eight bins and an item* $i(v)$. Capacities and sizes remain as before. The profit of placing item $i(v)$ in a bin of vertex u, with $u \neq v$, is $p(u,v)$.

Let OPT be the value of an optimum solution of MAX-CROWN in G, and let $\mathrm{OPT}_{\mathrm{GAP}}$ be the value of an optimum solution for the constructed instance of GAP. Since any optimum solution of MAX-CROWN, being a planar graph, can be decomposed into five star forests [16], there exists a star forest carrying at least $\mathrm{OPT}/5$ of the total profit. Such a star forest corresponds to a solution of GAP for the constructed instance; therefore, $\mathrm{OPT}_{\mathrm{GAP}} \geq \mathrm{OPT}/5$. Now we compute an α-approximation for the GAP instance, which results in a solution of total profit $\mathrm{ALG}_{\mathrm{GAP}} \geq \mathrm{OPT}_{\mathrm{GAP}}/\alpha \geq \mathrm{OPT}/(5\alpha)$. Next, we show how our solution induces a feasible solution of MAX-CROWN where every vertex $v \in V$ is either a bin or an item.

Consider the directed graph $G_{\mathrm{GAP}} = (V, E_{\mathrm{GAP}})$ with $(u,v) \in E_{\mathrm{GAP}}$ if and only if the item corresponding to $u \in V$ is placed into a bin corresponding to $v \in V$. A connected component in G_{GAP} with n' vertices has at most n' edges since every item can be placed into at most one bin. If $n' = 2$, we arbitrarily make one of the vertices a bin and the other an item. If $n' > 2$, the connected component is a 1-tree, that is, a tree and an edge. We partition the edges into two subgraphs: a star forest and the disjoint union of a star forest and a cycle. Note that both subgraphs can be represented by touching boxes if we allow point contacts because the stars correspond to a GAP solution. Hence, choosing a subgraph with larger weight and post-processing the solution as in the proof of Theorem 4 results in a feasible solution of MAX-CROWN with no point contacts. Initially, we discarded at most half of the weight and the post-processing keeps at least $3/4$ of the weight, so $\mathrm{ALG} \geq 3\,\mathrm{ALG}_{\mathrm{GAP}}/8$. Therefore, $\mathrm{ALG} \geq 3\,\mathrm{OPT}/(40\alpha)$. □

4 The Unweighted Case

In this section, we consider the unweighted MAX-CROWN problem, that is, all desired contacts have profit 1. Thus, we want to maximize the number of edges of the input graph realized by the contact representation. We present approximation algorithms for different graph classes. First, we give a 2-approximation for trees. Then, we present a PTAS for planar graphs of bounded degree. Finally, we provide a $(5+16\alpha/3)$-approximation for general graphs.

Theorem 7. *Unweighted* MAX-CROWN *on trees admits a 2-approximation.*

Proof. Let T be the input tree. We first decompose T into edge-disjoint stars as follows. If T has at most two vertices, then the decomposition is straight-forward. So, we assume w.l.o.g. that T has at least three vertices and is rooted at a non-leaf vertex. Let u be a vertex of T such that all its children, say $v_1,\dots,v_k$, are leaf vertices. If u is the root of T, then the decomposition contains only one star centered at u. Otherwise, denote by π the parent of u in T, create a star S_u centered at u with edges $(u,\pi),(u,v_1),\dots,(u,v_k)$ and call the edge (u,π) of S_u the *anchor edge* of S_u. The removal of $u,v_1,\dots,v_k$ from T results in a new tree. Therefore, we can recursively apply the same procedure. The result is a decomposition of T into edge-disjoint stars covering all edges of T.

We next remove, for each star, its anchor edge from T. We apply the PTAS of Theorem 1 to the resulting star forest and claim that the result is a 2-approximation for T. To prove the claim, consider a star S'_u of the new star forest, centered at u with edges $(u,v_1),\dots,(u,v_k)$ and let ALG be the total number of contacts realized by the $(1+\varepsilon)$-approximation algorithm on S'_u. We consider the following two cases.

(a) $1 \leq k \leq 4$: Since it is always possible to realize four contacts of a star, $\mathrm{ALG} \geq k$. Note that an optimal solution may realize at most $k+1$ contacts (due to the absence of the anchor edge from S'_u). Hence, our algorithm has approximation ratio $(k+1)/k \leq 2$.

(b) $k \geq 5$: Since it is always possible to realize four contacts of a star, we have $\mathrm{ALG} \geq 4$. On the other hand, an optimal solution realizes at most $(1+\varepsilon)\,\mathrm{ALG}+1$ contacts. Thus, the approximation ratio is $((1+\varepsilon)\,\mathrm{ALG}+1)/\,\mathrm{ALG} \leq (1+\varepsilon)+1/4 < 2$.

The theorem follows from the fact that all edges of T are incident to the star centers. □

Next, we develop a PTAS for bounded-degree planar graphs. Our construction needs two lemmas, the first of which was shown by Barth et al. [1].

Lemma 3 ([1]). *If the input graph $G = (V,E)$ has maximum degree Δ then* $\mathrm{OPT} \geq 2|E|/(\Delta+1)$.

The second lemma provides an exponential-time exact algorithm for MAX-CROWN. The proof is given in the full version [3].

Lemma 4. *There is an exact algorithm for unweighted* MAX-CROWN *with running time* $2^{O(n \log n)}$.

Theorem 8. *Unweighted* MAX-CROWN *on planar graphs with maximum degree* Δ *admits a PTAS. More specifically, for any* $\varepsilon > 0$ *there is an* $(1+\varepsilon)$*-approximation algorithm with linear running time* $n2^{(\Delta/\varepsilon)^{O(1)}}$.

Proof. Let r be a parameter to be determined later. Frederickson [14] showed that we can find a vertex set $X \subseteq V$ (called *r-division*) of size $O(n/\sqrt{r})$ such that the following holds. The vertex set $V \setminus X$ can be partitioned into n/r vertex sets $V_1, \ldots, V_{n/r}$ such that (i) $|V_i| \leq r$ for $i = 1, \ldots, n/r$ and (ii) there is no edge running between any two distinct vertex sets V_i and V_j. In what follows, we assume w.l.o.g. that G is connected, as we can apply the PTAS to every connected component separately.

We apply the result of Frederickson to the input graph and compute an r-division X. By removing the vertex set X from the graph, we remove $O(n\Delta/\sqrt{r})$ edges from G. Now, we apply the exact algorithm of Lemma 4 to each of the induced subgraphs $G[V_i]$ separately. The solution is the union of the optimum solutions to $G[V_i]$.

Since no edge runs between the distinct sets V_i and V_j, the subgraphs $G[V_i]$ cover $G - X$. Let $E^\star$ be the set of edges realized by an optimum solution to G, let $\mathrm{OPT} = |E^\star|$, and let $\mathrm{OPT}' = |E^\star \cap E(G-X)|$. By Lemma 3, we have that $\mathrm{OPT} \geq 2(n-1)/(\Delta+1) = \Omega(n/\Delta)$. When we removed X from G, we removed $O(n\Delta/\sqrt{r})$ edges. Hence, $\mathrm{OPT} = \mathrm{OPT}' + O(n\Delta/\sqrt{r})$ and $\mathrm{OPT}' = \Omega(n(1/\Delta - \Delta/\sqrt{r}))$.

Since we solved each sub-instance $G[V_i]$ optimally and since these sub-instances cover $G - X$, the solution created by our algorithm realizes at least OPT' many edges. Using this fact and the above bounds on OPT and OPT′, the total performance of our algorithm can be bounded by

$$\frac{\mathrm{OPT}}{\mathrm{OPT}'} = \frac{\mathrm{OPT}' + O(n\Delta/\sqrt{r})}{\mathrm{OPT}'} = 1 + O\left(\frac{n\Delta/\sqrt{r}}{n(1/\Delta - \Delta/\sqrt{r})}\right) = 1 + O\left(\frac{\Delta^2}{\sqrt{r} - \Delta^2}\right).$$

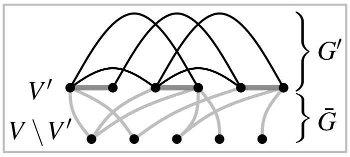
We want this last term to be smaller than $1+\varepsilon$ for some prescribed error parameter $0 < \varepsilon \leq 1$. It is not hard to verify that this can be achieved by letting $r = \Theta(\Delta^4/\varepsilon^2)$. Since each of the subgraphs $G[V_i]$ has at most r vertices, the total running time for determining the solution is $n2^{(\Delta/\varepsilon)^{O(1)}}$. □

Before tackling the case of general graphs, we need a lower bound on the size of maximum matchings in planar graphs in terms of the numbers of vertices and edges.

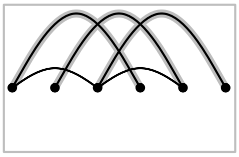
Lemma 5. *Any planar graph with* n *vertices and* m *edges contains a matching of size at least* $(m-2n)/3$.

Proof. Let G be a planar graph. Our proof is by induction on n. The claim holds for $n = 1$.

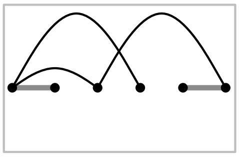
For the inductive step assume that $n > 1$. If G is not connected, the claim follows by applying the inductive hypothesis to every connected component. Now assume that G has a vertex u of degree less than 3. Consider the graph $G' = G - u$ with $n' = n - 1$ vertices and $m' \geq m - 2$ edges. By the induction hypothesis, G' (and hence, G, too) has a matching of size at least $(m' - 2n')/3 \geq ((m-2) - 2(n-1))/3 = (m-2n)/3$.



(a) G is covered by $\bar{G}$ (bipartite, gray) and G'. The graph G' is induced by the matching M (gray, bold).



(b) maximum matching M'' (gray/black) in $G'' = G' - M$



(c) optimum solution to G': graph G^* (black) and part of M (gray)

Fig. 2. Partitioning the input graph and the optimum solution in the proof of Theorem 9

It remains to tackle the case where G is connected and has minimum degree 3. Nishizeki and Baybars [19] showed that any connected planar graph with at least $n \geq 10$ vertices and minimum degree 3 has a matching of size at least $\lceil (n+2)/3 \rceil \geq n/3$. This shows the claim for $n \geq 10$ since $m \leq 3n - 6$.

In the remaining cases, G has $n \leq 9$ vertices. Due to planarity, we have $(m-2n)/3 \leq (n-6)/3 \leq 1$. Hence, any nonempty matching is large enough. □

Theorem 9. *Unweighted* MAX-CROWN *on general graphs admits a* $(5 + 16\alpha/3)$-*approximation.*

Proof. The algorithm first computes a maximal matching M in G. Let V' be the set of vertices matched by M, let G' be the subgraph induced by V', and let E' be the edge set of G'. Note that $\bar{G} = G - E'$ is a bipartite graph with partition $(V', V \setminus V')$. This is because the matching M is maximal, which implies that every edge in $E \setminus E'$ is incident to a vertex in V' and to a vertex not in V'; see Fig. 2a. Hence, we can compute a $16\alpha/3$-approximation to $\bar{G}$ using the algorithm presented in Theorem 4.

Consider the graph $G'' = (V', E' \setminus M)$ and compute a maximum matching M'' in G''; see Fig. 2b. The edge set $M \cup M''$ is a set of vertex-disjoint paths and cycles and can therefore be completely realized [1]. The algorithm realizes this set. Below, we argue that this realization is in fact a 5-approximation for G', which completes the proof (due to Lemma 1 and since G is covered by G' and $\bar{G}$).

Let $n' = |V'|$ be the number of vertices of G'. Let E^* be the set of edges realized by an optimum solution to G', and let $\text{OPT} = |E^*|$. Consider the subgraph $G^* = (V', E^* \setminus M)$ of G''; see Fig. 2c. Note that G^* is planar and contains at least $\text{OPT} - n'/2$ many edges. Applying Lemma 5 to G^*, we conclude that the maximum matching M'' of G'' has size at least $(\text{OPT} - 5n'/2)/3$. Hence, by splitting OPT appropriately, we obtain

$$\text{OPT} = (\text{OPT} - 5n'/2) + 5n'/2 \leq 3|M''| + 5|M| \leq 5|M'' \cup M|. \qquad \square$$

5 Conclusions and Open Problems

We presented approximation algorithms for the MAX-CROWN problem, which can be used for constructing semantics-preserving word clouds. Apart from improving approximation factors for various graph classes, many open problems remain. Most of our algorithms are based on covering the input graph by subgraphs and packing solutions for the individual subgraphs. Both subproblems—covering graphs with special types of subgraphs and packing individual solutions together—are interesting problems in their own right. Practical variants of the problem are also of interest, for example, restricting the heights of the boxes to predefined values (determined by font sizes), or defining more than immediate neighbors to be in contact, thus considering non-planar "contact" graphs. Another interesting variant is when the bounding box of the representation has a certain fixed size or aspect ratio.

Acknowledgement. We thank an anonymous reviewer for pointing out a simpler analysis for the last case in the proof of Lemma 5.

References

1. Barth, L., Fabrikant, S.I., Kobourov, S.G., Lubiw, A., Nöllenburg, M., Okamoto, Y., Pupyrev, S., Squarcella, C., Ueckerdt, T., Wolff, A.: Semantic word cloud representations: Hardness and approximation algorithms. In: Pardo, A., Viola, A. (eds.) LATIN 2014. LNCS, vol. 8392, pp. 514–525. Springer, Heidelberg (2014)
2. Barth, L., Kobourov, S.G., Pupyrev, S.: Experimental comparison of semantic word clouds. In: Gudmundsson, J., Katajainen, J. (eds.) SEA 2014. LNCS, vol. 8504, pp. 247–258. Springer, Heidelberg (2014)
3. Bekos, M., van Dijk, T., Fink, M., Kindermann, P., Kobourov, S.G., Pupyrev, S., Spoerhase, J., Wolff, A.: Improved approximation algorithms for box contact representations. Arxiv report (2014) arxiv.org/abs/1403.4861
4. Briest, P., Krysta, P., Vöcking, B.: Approximation techniques for utilitarian mechanism design. SIAM J. Comput. 40(6), 1587–1622 (2011)
5. Buchsbaum, A.L., Gansner, E.R., Procopiuc, C.M., Venkatasubramanian, S.: Rectangular layouts and contact graphs. ACM Trans. Algorithms 4(1) (2008)
6. Chekuri, C., Khanna, S.: A PTAS for the multiple knapsack problem. In: 11th ACM-SIAM Symp. Discrete Algorithms (SODA), pp. 213–222. SIAM (2000)
7. Cohen, R., Katzir, L., Raz, D.: An efficient approximation for the generalized assignment problem. Inf. Process. Lett. 100(4), 162–166 (2006)
8. Cui, W., Wu, Y., Liu, S., Wei, F., Zhou, M., Qu, H.: Context-preserving dynamic word cloud visualization. IEEE Comput. Graph. Appl. 30(6), 42–53 (2010)
9. Dwyer, T., Marriott, K., Stuckey, P.J.: Fast node overlap removal. In: Healy, P., Nikolov, N.S. (eds.) GD 2005. LNCS, vol. 3843, pp. 153–164. Springer, Heidelberg (2006)
10. Eppstein, D., Mumford, E., Speckmann, B., Verbeek, K.: Area-universal and constrained rectangular layouts. SIAM J. Comput. 41(3), 537–564 (2012)
11. Erkan, G., Radev, D.R.: Lexrank: graph-based lexical centrality as salience in text summarization. J. Artif. Int. Res. 22(1), 457–479 (2004)
12. Felsner, S.: Rectangle and square representations of planar graphs. In: Pach, J. (ed.) Thirty Essays on Geometric Graph Theory, pp. 213–248. Springer, Heidelberg (2013)
13. Fleischer, L., Goemans, M.X., Mirrokni, V., Sviridenko, M.: Tight approximation algorithms for maximum separable assignment problems. Math. Oper. Res. 36(3), 416–431 (2011)
14. Frederickson, G.N.: Fast algorithms for shortest paths in planar graphs, with applications. SIAM J. Comput. 16(6), 1004–1022 (1987)
15. Gansner, E.R., Hu, Y.: Efficient, proximity-preserving node overlap removal. J. Graph Algortihms Appl. 14(1), 53–74 (2010)

16. Hakimi, S.L., Mitchem, J., Schmeichel, E.F.: Star arboricity of graphs. Discrete Math. 149(1-3), 93–98 (1996)
17. Li, H.: Word clustering and disambiguation based on co-occurrence data. J. Nat. Lang. Eng. 8(1), 25–42 (2002)
18. Nash-Williams, C.: Decomposition of finite graphs into forests. J. L. Math. Soc. 39, 12 (1964)
19. Nishizeki, T., Baybars, I.: Lower bounds on the cardinality of the maximum matchings of planar graphs. Discrete Math. 28(3), 255–267 (1979)
20. Nöllenburg, M., Prutkin, R., Rutter, I.: Edge-weighted contact representations of planar graphs. J. Graph Algorithms Appl. 17(4), 441–473 (2013)
21. Paulovich, F.V., Toledo, F.M.B., Telles, G.P., Minghim, R., Nonato, L.G.: Semantic wordification of document collections. Comput. Graph. Forum 31(3), 1145–1153 (2012)
22. Raisz, E.: The rectangular statistical cartogram. Geogr. Review 24(3), 292–296 (1934)
23. Viégas, F.B., Wattenberg, M., Feinberg, J.: Participatory visualization with Wordle. IEEE Trans. Vis. Comput. Graph. 15(6), 1137–1144 (2009)
24. Wu, Y., Provan, T., Wei, F., Liu, S., Ma, K.L.: Semantic-preserving word clouds by seam carving. Comput. Graph. Forum 30(3), 741–750 (2011)

Minimum Partial-Matching and Hausdorff RMS-Distance under Translation: Combinatorics and Algorithms

Rinat Ben-Avraham[1,⋆], Matthias Henze[2,⋆⋆], Rafel Jaume[2,⋆⋆⋆], Balázs Keszegh[3,†], Orit E. Raz[1,‡], Micha Sharir[1,§], and Igor Tubis[1]

[1] Blavatnik School of Computer Science, Tel Aviv University, Tel Aviv, Israel
rinatba@gmail.com, {oritraz,michas}@post.tau.ac.il, mrtubis@gmail.com

[2] Institut für Informatik, Freie Universität Berlin, Berlin, Germany
matthias.henze@fu-berlin.de, jaume@mi.fu-berlin.de

[3] Alfréd Rényi Institute of Mathematics, Hungarian Academy of Sciences, Budapest, Hungary
keszegh@renyi.hu

Abstract. We consider the RMS-distance (sum of squared distances between pairs of points) under translation between two point sets in the plane. In the Hausdorff setup, each point is paired to its nearest neighbor in the other set. We develop algorithms for finding a local minimum in near-linear time on the line, and in nearly quadratic time in the plane. These improve substantially the worst-case behavior of the popular ICP heuristics for solving this problem. In the partial-matching setup, each point in the smaller set is matched to a distinct point in the bigger set. Although the problem is not known to be polynomial, we establish several structural properties of the underlying subdivision of the plane and derive improved bounds on its complexity. In addition, we show how to compute a local minimum of the partial-matching RMS-distance under translation, in polynomial time.

Keywords: partial matching, Hausdorff RMS-distance, polyhedral subdivision, local minimum.

⋆ Supported by Grant 2012/229 from the U.S.-Israel Binational Science Foundation.

⋆⋆ Supported by ESF EUROCORES programme EuroGIGA-VORONOI, (DFG): RO 2338/5-1.

⋆⋆⋆ Supported by La-Caixa and the DAAD.

† Supported by Hungarian National Science Fund (OTKA), under grant PD 108406, NN 102029 (EUROGIGA project GraDR 10-EuroGIGA-OP-003), NK 78439, by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences and by the DAAD.

‡ Supported by Grant 892/13 from the Israel Science Foundation.

§ Supported by Grant 2012/229 from the U.S.-Israel Binational Science Foundation, by Grant 892/13 from the Israel Science Foundation, by the Israeli Centers for Research Excellence (I-CORE) program (center no. 4/11), and by the Hermann Minkowski–MINERVA Center for Geometry at Tel Aviv University.

1 Introduction

Let A and B be two finite sets of points in the plane, of respective cardinalities n and m. We are interested in measuring the similarity between A and B, under a suitable proximity measure. We consider two such measures where the proximity is the sum of the squared distances between pairs of points. In the first, we assume that $n > m$ and we want to match all the points of B (a specific pattern that we want to identify), in a one-to-one manner, to a subset of A (a larger picture that "hides" the pattern) of size $|B|$. This is motivated by situations where we want a one-to-one matching between A and B [9,15,16]. In the second, each point is assigned to its nearest neighbor in the other set. See [1] for a similar generalization of the Hausdorff distance.

We refer to the measured distance between the sets, in both versions, as the *RMS distance*. In the former setup the measure is called the *partial-matching RMS-distance*, and in the latter we call it the *Hausdorff RMS-distance*. In both variants the sets A and B are in general not aligned, so we seek a translation of one of them that will minimize the appropriate RMS-distance, partial matching or Hausdorff.

The Partial-Matching RMS-Distance Problem. Let $A = \{a_1, \ldots, a_n\}$ and $B = \{b_1, \ldots, b_m\}$ be two sets of points in the plane. Here we assume that $m < n$, and we seek a minimum-weight *maximum-cardinality matching* of B into A. This is a subset M of edges of the complete bipartite graph with edge set $B \times A$, so that each $b \in B$ appears in exactly one edge of M, and each $a \in A$ appears in at most one edge. The weight of an edge (b, a) is $\|b - a\|^2$, and the weight of a matching is the sum of the weights of its edges.

A maximum-cardinality matching can be identified as an injective assignment π of B into A. With a slight abuse of notation, we denote by $a_{\pi(i)}$ the point a_j that π assigns to b_i. In this notation, the minimum RMS partial-matching problem (for fixed locations of the sets) is to compute

$$M(B, A) = \min_{\pi : B \to A \text{ injective}} \sum_{i=1}^{m} \left\| b_i - a_{\pi(i)} \right\|^2 .$$

Allowing the pattern B to be translated, we obtain the problem of the minimum partial-matching RMS-distance under translation, defined as

$$M_T(B, A) = \min_{t \in \mathbb{R}^2} M(B + t, A) = \min_{\substack{t \in \mathbb{R}^2, \pi : B \to A, \\ \pi \text{ injective}}} \sum_{i=1}^{m} \left\| b_i + t - a_{\pi(i)} \right\|^2 .$$

The function $F(t) := M(B + t, A)$ induces a subdivision of $\mathbb{R}^2$, where two points $t_1, t_2 \in \mathbb{R}^2$ are in the same region if the minimum of F at t_1 and at t_2 is attained by the same assignment $\pi : B \to A$. We refer to this subdivision, following Rote [12], as the *partial-matching subdivision* and denote it by $\mathcal{D}_{B,A}$. We say that a matching is *optimal* if it attains $F(t)$ for some $t \in \mathbb{R}^2$.

The Hausdorff RMS Distance Problem. Let $N_A(x)$ (resp., $N_B(x)$) denote the nearest neighbor in A (resp., in B) of a point $x \in \mathbb{R}^2$. The *unidirectional (Hausdorff) RMS distance* between B and A is defined as

$$RMS(B, A) = \sum_{b \in B} \|b - N_A(b)\|^2.$$

We also consider *bidirectional* RMS distances, in which we also measure distances from the points of A to their nearest neighbors in B. We consider two variants of this notion. The first variant is the *L_1-bidirectional RMS distance* between A and B, which is defined as

$$RMS_1(B, A) = RMS(A, B) + RMS(B, A).$$

The second variant is the *L_∞-bidirectional RMS distance* between A and B, and is defined as

$$RMS_\infty(B, A) = \max\{RMS(A, B), RMS(B, A)\}.$$

Allowing one of the sets (say, B) to be translated, we define the *minimum unidirectional RMS distance under translation* to be

$$RMS_T(B, A) = \min_{t \in \mathbb{R}^2} RMS(B + t, A) = \min_{t \in \mathbb{R}^2} \sum_{b \in B} \|b + t - N_A(b + t))\|^2,$$

where $B + t = \{b_1 + t, \ldots, b_m + t\}$. Similarly, we define the *minimum L_1- and L_∞-bidirectional RMS distances under translation* to be

$$RMS_{T,1}(B, A) = \min_{t \in \mathbb{R}^2} RMS_1(B + t, A) \qquad \text{and}$$
$$RMS_{T,\infty}(B, A) = \min_{t \in \mathbb{R}^2} RMS_\infty(B + t, A).$$

Background. A thorough initial study of the minimum RMS partial-matching distance under translations is given by Rote [12]; see also [5,13] for two follow-up studies, another study in [11], and an abstract of an earlier version of parts of this paper [8]. The resulting subdivision $\mathcal{D}_{B,A}$, as defined above, is shown in [12] to be a convex subdivision. Rote's main contribution for the analysis of the complexity of $\mathcal{D}_{B,A}$ was to show that a line crosses only $O(mn)$ regions of the subdivision (see Theorem 1 below). However, obtaining sharp bounds for the complexity of $\mathcal{D}_{B,A}$ is still an open issue, where the best known upper bounds are exponential.

The problem of Hausdorff RMS minimization under translation has been considered in the literature (see, e.g., [1] and references therein), although only scarcely so. If A and B are sets of points on the line, the complexity of the Hausdorff RMS function, as a function of t, is $O(mn)$ (and this bound is tight in the worst case). Moreover, the function can have many local minima (up to $\Theta(mn)$ in the worst case). Hence, finding a translation that minimizes the Hausdorff RMS distance can be done in brute force, in $O(mn \log(mn))$ time, but a worst-case near linear algorithm is not known. In practice, though, there exists a popular heuristic technique, called the ICP (Iterated Closest Pairs) algorithm, proposed by Besl and McKay [3] and analyzed in Ezra et al. [7]. Although the algorithm is reported to be efficient in practice, it might perform $\Theta(mn)$ iterations in the worst case. Moreover, each iteration takes close to linear time (to find the nearest neighbors in the present location).

The situation is worse in the plane, where the complexity of the RMS function is $O(m^2n^2)$, a bound which is worst-case tight, and the bounds for the performance of the ICP algorithm, are similarly worse. Similar degradation shows up in higher dimensions too; see, e.g., [7].

Our Results. In this paper we study these two fairly different variants of the problem of minimizing the RMS distance under translation, and improve the state of the art in both of them.

In the partial-matching variant, we first analyze the complexity of $\mathcal{D}_{B,A}$. We significantly improve the bound from the naive $O(n^m)$ to $O(n^2m^{3.5}(e\ln m+e)^m)$. A preliminary informal exposition of this analysis by a subset of the authors is given in the (non-archival) note [8]. This paper expands the previous note, derives additional interesting structural properties of the subdivision, and significantly improves the complexity bound. The arguments that establish the bound can be generalized to bound the number of regions of the analogous subdivision in $\mathbb{R}^d$ by $O\left((n^2m)^d(e\ln m+e)^m)/\sqrt{m}\right)$. The derivation of the upper bound proceeds by a reduction that connects partial matchings to a combinatorial question based on a game theoretical problem, which we believe to be of independent interest.

Next we present a polynomial-time algorithm for finding a local minimum of the partial-matching RMS-distance. This is significant, given that we do not have a polynomial bound on the size of the subdivision. We also fill in the details of explicitly computing the intersections of a line with $\mathcal{D}_{B,A}$. Although Rote hinted at such an algorithm in [12], by exploiting some new properties of $\mathcal{D}_{B,A}$ derived here, we manage to compute the intersections in a simple, more efficient manner.

We also note that by combining the combinatorial bound for the complexity of $\mathcal{D}_{B,A}$, along with the procedures in the algorithm for finding a local minimum of the partial-matching RMS-distance, it is possible to traverse all of $\mathcal{D}_{B,A}$, and compute a *global* minimum of the partial-matching RMS-distance in time $O(n^3m^{7.5}(e\ln m+e)^m)$. This is the best known bound for this problem.

For the Hausdorff variant, we provide improved algorithms for computing a local minimum of the RMS function, in one and two dimensions. Assuming $|A| = |B| = n$, in the one-dimensional case the algorithms run in time $O(n\log^2 n)$, and in the two-dimensional case they run in time $O(n^2\log n)$. Our approach thus beats the worst-case running time of the ICP algorithm (used for about two decades to solve this problem). The approach is an efficient search through the (large number of) critical values of the RMS function. The techniques are reasonably standard, although their assembly is somewhat involved.

2 Properties of $\mathcal{D}_{B,A}$

We begin by reconstructing several basic properties of $\mathcal{D}_{B,A}$ that have been noted in [12]. First, if we fix the translation $t \in \mathbb{R}^2$ and the assignment π, the cost of the matching, denoted by $f(\pi, t)$, is

$$f(\pi, t) = \sum_{i=1}^{m} \left\| b_i + t - a_{\pi(i)} \right\|^2 = c_\pi + \langle t, d_\pi \rangle + m \left\| t \right\|^2, \tag{1}$$

where $c_\pi = \sum_{i=1}^{m} \left\| b_i - a_{\pi(i)} \right\|^2$ and $d_\pi = 2\sum_{i=1}^{m}(b_i - a_{\pi(i)})$. For t fixed, the assignment π that minimizes $f(\pi, t)$ is the same assignment that minimizes $g(\pi, t) := c_\pi + \langle t, d_\pi \rangle$. It follows that $\mathcal{D}_{B,A}$ is the minimization diagram (the xy-projection) of the graph of the function

$$\mathcal{E}_{B,A}(t) = \min_{\pi: B \to A \text{ injective}} \left(c_\pi + \langle t, d_\pi \rangle \right), \quad t \in \mathbb{R}^2.$$

This is a lower envelope of a finite number of planes, so its graph is a convex polyhedron, and its projection $\mathcal{D}_{B,A}$ is a convex subdivision of the plane, whose faces are convex polygons. The great open question regarding minimum partial-matching RMS-distance under translation, is whether the number of regions of $\mathcal{D}_{B,A}$ is polynomial in m and n. A significant, albeit small step towards settling this question is the following result of Rote [12].

Theorem 1 (Rote [12]). *A line intersects the interior of at most $m(n-m)+1$ different regions of the partial-matching subdivision $\mathcal{D}_{B,A}$.*

The following property, observed by Rote [12], seems to be well known [16].

Lemma 1. *For any $A' \subset A$, with $|A'| = m$, the optimal assignment that realizes the minimum $M(B + t, A')$ is independent of the translation $t \in \mathbb{R}^2$.*

Next, we derive several additional properties of $\mathcal{D}_{B,A}$ which show that the diagram has, locally, low-order polynomial complexity.

Lemma 2. *Every edge of $\mathcal{D}_{B,A}$ has a normal vector of the form $a_j - a_i$ for suitable $i, j \in \{1, \ldots, n\}$.*

Proof. Let E be an edge of $\mathcal{D}_{B,A}$ common to the regions associated with the injections $\pi, \sigma : B \to A$. By definition, $g(\pi, t) = g(\sigma, t) \le g(\delta, t)$ for every injection $\delta : B \to A$ and for any $t \in E$. By Equation (1), E is contained in the line $\ell(\pi, \sigma) = \{t \in \mathbb{R}^2 : \langle t, d_\pi - d_\sigma \rangle = c_\sigma - c_\pi\}$. Let $H = (\pi \setminus \sigma) \cup (\sigma \setminus \pi)$. It is easy to see that H consists of a vertex-disjoint union of cycles and alternating paths. Let $\gamma_1, \ldots, \gamma_p$ be these cycles and paths. It is not hard to see that every cycle and every path can be "flipped" independently while preserving the validity of the matching; that is, we can choose, within any of the γ_j's, either all the edges corresponding to π or all the ones corresponding to σ, without interfering with other cycles or paths, so that the resulting collection of edges still represents an injection from B into A. Observe now that $\ell(\pi,\sigma) = \{t \in \mathbb{R}^2 : \left\langle t, \sum_{j=1}^{p} d_{\gamma_j} \right\rangle = -\sum_{j=1}^{p} c_{\gamma_j}\}$, where d_{γ_j} is the sum of the terms in $d_\pi - d_\sigma$ that involve only the $a_i \in A$ contained in γ_j and c_{γ_j} is analogously defined for $c_\pi - c_\sigma$. Note that d_{γ_j} is 0 for every cycle γ_j and, therefore, at least one of the γ_j's is a path. Then, we must have $\left\langle t, d_{\gamma_j} \right\rangle = -c_{\gamma_j}$ for all $j = 1, \ldots, p$ and every $t \in \ell(\pi,\sigma)$. Otherwise, a flip in a path or cycle violating the equation would contradict the optimality of π or of σ along $\ell(\pi,\sigma)$. Therefore, all the vectors d_{γ_j} must be orthogonal to $\ell(\pi,\sigma)$. Hence, the direction of $d_\pi - d_\sigma$ is the same as the one of d_{γ_j} for every *path* γ_j. If a path, say γ_1, starts at some a_j and ends at some a_i, then $d_{\gamma_1} = a_j - a_i$, which concludes the proof. □

Remark. It follows that if A is in general position then H has exactly one alternating path, and the pair a_i, a_j is unique.

Lemma 3. *i) $\mathcal{D}_{B,A}$ has at most $4m(n-m)$ unbounded regions.*
ii) Every region in $\mathcal{D}_{B,A}$ has at most $m(n-m)$ edges.
iii) Every vertex in $\mathcal{D}_{B,A}$ has degree at most $2m(n-m)$.
iv) Any convex path can intersect at most $m(n-m)+n(n-1)$ regions of $\mathcal{D}_{B,A}$, i.e., while translating B along any convex path, the optimal partial matching can change at most $m(n-m)+n(n-1)$ times.

Proof. i) Take a bounding box that encloses all the vertices of the diagram. By Theorem 1, every edge of the bounding box crosses at most $m(n-m)+1$ regions of $\mathcal{D}_{B,A}$. The edges of the box traverse only unbounded regions, and cross every unbounded region exactly once, except for the coincidences of the last region traversed by an edge and the first region traversed by the next edge.

ii) By Lemma 2, the normal vector of every edge of a region corresponding to the injection π is a multiple of $a_j - a_i$ for some $a_i \in \pi(B)$ and $a_j \notin \pi(B)$. There are exactly $m(n-m)$ such possibilities.

iii) Let v be a vertex of $\mathcal{D}_{B,A}$. Draw two generic parallel lines close enough to each other to enclose v and no other vertex. Each edge adjacent to v is crossed by one of the lines, and by Theorem 1 each of these lines crosses at most $m(n-m)$ edges.

iv) We use the following property that was observed in Rote's proof of Theorem 1. Suppose that we translate B along a line in some direction v. Rank the points of A by their order in the v-direction, i.e., $a < a'$ means that $\langle a, v\rangle < \langle a', v\rangle$ (for simplicity, assume that v is generic so there are no ties). Let Φ denote the sum of the ranks of the m points of A that participate in the optimal partial match. As Rote has shown, whenever the optimal assignment changes, Φ must increase. Now follow our convex path γ, which, without loss of generality, can be assumed to be polygonal. As we traverse an edge of γ, Φ obeys the above property, increasing every time we cross into a new region of $\mathcal{D}_{B,A}$. When we turn (counterclockwise) at a vertex of γ, the ranking of A may change, but each such change consists of a sequence of swaps of consecutive elements in the present ranking. At each such swap, Φ can decrease by at most 1. Since γ is convex, each pair of points of A can be swapped at most twice, so the total decrease in Φ is at most $2\binom{n}{2} = n(n-1)$. Hence, the accumulated increase in Φ, and thus also the total number of regions of $\mathcal{D}_{B,A}$ crossed by γ, is at most $\Big(n + (n-1) + \ldots + (n-m+1)\Big) - \Big(1 + 2 + \ldots + m\Big) + n(n-1)$. □

In the remainder of this section, we focus on establishing a global bound on the complexity of the diagram $\mathcal{D}_{B,A}$. We begin by deriving the following technical auxiliary results.

Lemma 4. *Let π be an optimal assignment for a fixed translation $t \in \mathbb{R}^2$.*

i) There is no cyclic sequence $(i_1, i_2, \ldots, i_k, i_1)$ satisfying $\|b_{i_j} + t - a_{\pi(i_j)}\| < \|b_{i_j} + t - a_{\pi(i_{j+1})}\|$ for all $j \in \{1, \ldots, k\}$ (modulo k).
ii) Each point of $B + t$ is matched to one of its m nearest neighbors in A.
iii) At least one point in $B + t$ is matched to its nearest neighbor in A.
iv) There exists an ordering $\langle b_1, \ldots, b_m\rangle$ of the elements of B, such that each b_k is assigned by π to one of its k nearest neighbors in A, for $k = 1, \ldots, m$.

Proof. i) For the sake of contradiction, we assume that there exists a cyclic sequence that satisfies all the prescribed inequalities. Consider the assignment σ defined by $\sigma(i_j) = \pi(i_{(j-1) \bmod k})$ for all $j \in \{1, \ldots, k\}$ and $\sigma(\ell) = \pi(\ell)$ for all other indices ℓ. Since π is a one-to-one matching, we have that $\pi(i_j) \neq \pi(i_{j'})$ for all different $j, j' \in \{1, \ldots, k\}$ and, consequently, σ is one-to-one as well. It is easily checked that $f(\sigma, t) < f(\pi, t)$, contradicting the optimality of π.

ii) For contradiction, assume that for some point $b \in B$, $b + t$ is not matched by π to one of its m nearest neighbors in A. Then, at least one of these neighbors, say a, cannot be matched (because these m points can be claimed only by the remaining $m-1$ points of $B+t$). Thus, we can reduce the cost of π by matching $b + t$ to a, a contradiction that establishes the claim.

iii) Again we assume for contradiction that π does not match any of the points of $B + t$ to its nearest neighbor in A. We construct the following cyclic sequence in the matching π. We start at some arbitrary point $b_1 \in B$, and denote by a_1 its nearest neighbor in A (to simplify the presentation, we do not explicitly mention the translation t in what follows). By assumption, b_1 is not matched to a_1. If a_1 is also not claimed in π by any of the points of B, then b_1 could have claimed it, thereby reducing the cost of π, which is impossible. Let then b_2 denote the point that claims a_1 in π. Again, by assumption, a_1 is not the nearest neighbor a_2 of b_2, and the preceding argument then implies that a_2 must be claimed by some other point b_3 of B. We continue this process, and obtain an alternating path $(b_1, a_1, b_2, a_2, b_3, \ldots)$ such that the edges (b_i, a_i) are not in π, and the edges (b_{i+1}, a_i) belong to π, for $i = 1, 2, \ldots$. The process must terminate when we reach a point b_k that either coincides with b_1, or is such that its nearest neighbor is among the already encountered points a_i, $i < k$. We thus obtain a cyclic sequence as in part i), reaching a contradiction.

iv) Start with some point $b_1 \in B$ such that $b_1 + t$ goes to its nearest neighbor a_1 in A in the optimal partial-matching π; such a point exists by part iii). Delete b_1 from B, and a_1 from A. The optimal matching of $B \setminus \{b_1\}$ into $A \setminus \{a_1\}$ (relative to t) is equal to the restriction of π to the points in $B \setminus \{b_1\}$, because otherwise we could have improved π itself. We apply part iii) to the reduced sets, and obtain a second point $b_2 \in B \setminus \{b_1\}$ whose translation b_2+t is matched to its nearest neighbor a_2 in $A \setminus \{a_1\}$, which is either its first or second nearest neighbor in the original set A. We keep iterating this process until the entire set B is exhausted. At the k-th step we obtain a point $b_k \in B \setminus \{b_1, \dots, b_{k-1}\}$, such that the nearest neighbor a_k in $A \setminus \{a_1, \dots, a_{k-1}\}$ is matched to b_k by π, so a_k is among the k nearest neighbors in A of $b_k + t$. □

Observe, that the geometric properties in Lemma 4 can be interpreted in purely combinatorial terms. Indeed, for t fixed, associate with each $b_i \in B$ an ordered list $L_t(b_i)$, called its *preference list*, which consists of the points of A sorted by their distances from $b_i + t$. In general, given m such ordered lists on n elements, an injective assignment from $\{1, \dots, m\}$ to $\{1, \dots, n\}$ such that there is no cycle as in part i) is called *stable* or *Pareto efficient.* The problem of finding a stable matching was studied, for the case $m = n$, in the game theory literature under the name of the *House Allocation Problem* [14]. Note also that the proofs of parts ii)–iv) can be carried out in this abstract setting, and hold for any stable matching. Note that part iv) immediately yields an upper bound of $m!$ on the number of stable matchings and, in addition, implies that only the first m elements of each $L_t(b_i)$ are relevant. This bound is tight for the combinatorial problem, since if the ordered lists all coincide there are $m!$ different stable matchings. A recent article, motivated by the extended abstract [8] prior to this work, studied this combinatorial problem and derived the following.

Lemma 5 (Asinowski et al. [2]). *The number of elements that belong to some stable matching on m ordered preference lists is at most $m(\ln m + 1)$.*

The properties derived so far imply the following significantly improved upper bound on the complexity of $\mathcal{D}_{B,A}$.

Theorem 2. *The combinatorial complexity of $\mathcal{D}_{B,A}$ is $O(n^2 m^{3.5}(e \ln m + e)^m)$.*

Proof. The proof has two parts. First, we identify a convex subdivision K such that in each of its regions the first m elements of the ordered preference lists $L_t(b)$ of neighbors of each $b+t$, according to their distance from $b+t$, are fixed for all $b \in B$. We show that the complexity of K is only polynomial; specifically, it is $O(n^2m^4)$. Second, we give an upper bound on how many regions of $\mathcal{D}_{B,A}$ can intersect a given region of K, using Lemma 5. Together, these imply an upper bound on the complexity of $\mathcal{D}_{B,A}$. The proof of the first part, which is based on a somewhat non-standard application of the Clarkson-Shor technique, is omitted in this version. We now consider all possible translations t in the interior of some fixed region τ of K and their corresponding optimal matchings. Lemma 4(i) ensures that all of them must be stable with respect to the fixed preference lists $L_t(b)$, for $b \in B$, over $t \in \tau$. In addition, Lemma 1 ensures that we only need to bound the number of different image sets of such stable matchings. Using the bound in Lemma 5, we can derive that the number of optimal matchings for translations in τ is then $O\left(\binom{m(\ln m+1)}{m}\right) = O\left(\frac{m^m(\ln m+1)^m}{m!}\right) = O\left(\frac{(e\ln m+e)^m}{\sqrt{m}}\right)$, where in the second step we used Stirling's approximation. Hence, by multiplying this bound by the number of regions in K, we conclude that the number of assignments corresponding to optimal matchings, and thus also the complexity of $\mathcal{D}_{B,A}$, is at most $O(n^2 m^{3.5}(e \ln m + e)^m)$. $\square$

The following proposition (proof omitted in this version) sets an obstruction for the combinatorial approach alone to yield a polynomial bound for $\mathcal{D}_{B,A}$.

Proposition 1. *For every $n \geq \lfloor \frac{m}{2} \rfloor + m$, there exists m preference lists of $\{1, \ldots, n\}$ with $\Omega\left(\frac{2^m}{\sqrt{m}}\right)$ different images of stable matchings.*

3 Finding a Local Minimum of the Partial-Matching RMS-Distance under Translation

The High-Level Algorithm. We now concentrate on the algorithmic problem of computing, in polynomial time, a local minimum of the partial-matching RMS-distance under translation.

We "home in" on a local minimum of $F(t)$ by maintaining a vertical slab I in the plane that is known to contain such a local minimum in its interior, and by repeatedly shrinking it until we obtain a slab I^* that does not contain any vertex of $\mathcal{D}_{B,A}$. That is, any (vertical) line contained in I^* intersects the same sequence of regions, and, by Theorem 1, the number of these regions is $O(mn)$. We compute these regions, find the optimal partial matching assignment in each region, and the corresponding explicit (quadratic) expression of $F(t)$, and search for a local minimum within each region.

A major component of the algorithm is a procedure, that we call $\Pi_1(\ell)$, which, for a given input line ℓ, constructs the intersection of $\mathcal{D}_{B,A}$ with ℓ, computes the global minimum t^* of F on ℓ, and determines a side of ℓ, in which F attains strictly smaller values than $F(t^*)$. If no such decrease is found in the neighborhood of t^* then it is a local minimum of F, and we stop. Using Lemma 2 and the Hungarian algorithm [6,10], $\Pi_1(\ell)$ runs in $O(m^5 n^2)$ time.

We use this "decision procedure" as follows. Suppose we have a current vertical slab I, bounded on the left by a line ℓ^- and on the right by a line ℓ^+. We assume that Π_1 has been executed on ℓ^- and on ℓ^+, and that we have determined that F assumes smaller values than its global minimum on ℓ^- to the right of ℓ^-, and that it assumes smaller values than its global minimum on ℓ^+ to the left of ℓ^+. This is easily seen to imply that F must contain a local minimum in the interior of I. (We note that just finding a *local* minimum of F along ℓ^+ or ℓ^- is not sufficient; see the full version for details.) Let ℓ be some vertical line passing through I. We run Π_1 on ℓ. If it determines that F attains smaller values to its left (resp., to its right), we shrink I to the slab bounded by ℓ^- and ℓ (resp., the slab bounded by ℓ and ℓ^+). By what has just been argued, this ensures that the new slab also contains a local minimum of F in its interior.

To initialize the slab I, we choose an arbitrary *horizontal* line λ, and run Π_1 on λ, to find the sequence S of its intersection points with the edges of $\mathcal{D}_{B,A}$. We run a binary search through S, where at each step we execute Π_1 on the vertical line through the current point. When the search terminates, we have a vertical slab I_0 whose intersection with λ is contained in a single region σ_0 of $\mathcal{D}_{B,A}$.

After this initialization, we find the region σ_1 that lies directly above σ_0 and that the final slab I^* should cross. In general, there are possibly many such regions, but fortunately, by Lemma 3(ii), their number is only at most $m(n-m)$.

To find σ_1, we compute the boundary of σ_0; this is done similarly to the execution of Π_1; see the full version for details. Once we have explored the boundary of σ_0, we take the sequence of all vertices of σ_0, and run a Π_1-guided binary search on the vertical lines passing through them, exactly as we did with the vertices of S, to shrink I_0 into a slab I_1, so that σ_0 intersects I_1 in a trapezoid (or a triangle), with a single (portion of an) edge at the top and a single edge at the bottom. This allows us to determine σ_1, which is the region lying on the other (higher) side of the top edge, in $O(m^5n^2\log(mn))$ time. A symmetric variant of this procedure will find the region lying directly below σ_0 in the final slab.

We repeat the previous step to find the entire stack of $O(nm)$ regions that I^* crosses, where each step shrinks the current slab and then crosses to the next region in the stack. Once this is completed, we find a local minimum within I^* as explained above. Again, details are omitted in this version.

In summary, we have the following main result of this section.

Theorem 3. *Given two finite point sets A, B in $\mathbb{R}^2$, with $n = |A| > |B| = m$, and such that for every two pairs $(a_1, a_2), (a_3, a_4) \in A \times A$ the vectors $a_1 - a_2$ and $a_3 - a_4$ are non-parallel, a local minimum of the partial-matching RMS-distance under translation can be computed in $O(m^6n^3\log n)$ time.*

4 Finding a Local Minimum of the Hausdorff RMS-Distance under Translation

In this section, we turn to the simpler problem involving the Hausdorff RMS-distance, and present efficient algorithms for computing a local minimum of the RMS function in one and two dimensions. Due to lack of space, most of the material in this section is omitted, and we only provide here a high-level review of our algorithms.

The One-Dimensional Unidirectional Case. Let $N_A(b+t)$ be the nearest neighbor in A of $b+t$, for $b \in B$, and $t \in \mathbb{R}$. The function $r(t) := RMS(B+t, A) = \sum_{b\in B}(b+t-N_A(b+t))^2$ is continuous and piecewise parabolic, with $O(mn)$ non-smooth breakpoints, which are the breakpoints of the step functions $N_A(b+t)$. For any given t_0, it is easy to compute, in $O(m\log n)$ time, the derivative $r'(t_0)$, or its left and right one-sided versions $r'(t_0)^-$, $r'(t_0)^+$ (when t_0 is a breakpoint). A simple observation is that if $I = [t_1, t_2]$ is an interval satisfying $r'(t_1)^+ < 0$ and $r'(t_2)^- > 0$ then I contains a local minimum of r. We thus start with a large interval I that contains all breakpoints of r, and keep shrinking it, halving the number of breakpoints in I in each step, until it contains only linearly many breakpoints, in which case r can be constructed explicitly over I, and searched for a local minimum, in near-linear time. Specifically, we obtain:

Theorem 4. *Given two finite point sets A, B on the real line, with $|A| = n$ and $|B| = m$, a local minimum of the unidirectional RMS distance under translation from B to A can be obtained in time $O(m \log^2 n + n \log n)$.*

The one-dimensional bidirectional case. Simple extensions of the procedure given above apply to the two variants of the minimum bidirectional Hausdorff RMS-distance, as defined in the introduction. Omitting the fairly routine details of these extensions, we obtain:

Theorem 5. *Given two finite point sets A, B on the real line, with $|A| = n$ and $|B| = m$, a local minimum under translation of the L_1-bidirectional or L_∞-bidirectional RMS distance between A and B, can be computed in time $O((n \log m + m \log n) \log \min\{m, n\})$.*

Minimum Hausdorff RMS-distance under translation in two dimensions. Here the function $r(t) := RMS(B+t, A) = \sum_{b \in B} \|b+t-N_A(b+t)\|^2$ induces a convex subdivision of the plane, where in each of its regions σ, all the m values $N_A(b+t)$, for $b \in B$, are fixed for $t \in \sigma$. This subdivision is simply the overlay M of the m shifted copies $\mathcal{V}(A-b)$, for $b \in B$, of the Voronoi diagram of A. These copies have a total of $O(mn)$ edges, and their overlay has thus complexity $O(m^2n^2)$ (which is tight in the worst case). Over each region of M, $r(t)$ is a quadratic function (a paraboloid), and the explicit expression for $r(t)$ can be updated in $O(1)$ time as we cross from one region to an adjacent one.

The goal is to search for a local minimum of r without explicitly constructing these many features of M. Similarly to the one-dimensional case, we maintain a vertical slab I, known to contain a local minimum, and keep shrinking it until it contains no vertices of M. In this case it overlaps only $O(mn)$ regions of M, vertically stacked above one another, and it is straightforward to enumerate all of them, get the explicit expressions of r over each of them, and search for a local minimum in each part, in a total of $O(mn)$ time.

The shrinking of I is performed in two phases. We first enumerate all $O(mn)$ Voronoi vertices of the original diagrams, and run a binary search through them, as above. The resulting intermediate slab contains no original vertices, so the edges that cross it behave like lines. They might still intersect at $O(m^2n^2)$ points within I, but we can run a binary search through them efficiently, using the (dual version of the) *slope selection* algorithm of [4], so that each step takes only $O(mn \log mn)$ time.

Concretely, we obtain:

Theorem 6. *Given two finite point sets A, B in $\mathbb{R}^2$, with $|A| = n$ and $|B| = m$, a local minimum of the unidirectional Hausdorff RMS-distance from B to A under translation can be computed in time $O(mn \log^2 mn)$.*

The bidirectional variants can be handled in much the same way, and, omitting the details, we get:

Theorem 7. *Given two finite point sets A, B in $\mathbb{R}^2$, with $|A| = n$ and $|B| = m$, a local minimum of the L_1-bidirectional or the L_∞-bidirectional Hausdorff RMS-distance between A and B under translation can be computed in $O(mn \log^2 mn)$ time.*

References

1. Agarwal, P.K., Har-Peled, S., Sharir, M., Wang, Y.: Hausdorff distance under translation for points, disks, and balls. ACM Trans. on Algorithms 6, 1–26 (2010)
2. Asinowski, A., Keszegh, B., Miltzow, T.: Counting houses of Pareto optimal matchings in the House Allocation Problem, arXiv:1401.5354v2
3. Besl, P.J., McKay, N.D.: A method for registration of 3-d shapes. IEEE Trans. Pattern Anal. Mach. Intell. 14, 239–256 (1992)
4. Cole, R., Salowe, J., Steiger, W., Szemerédi, E.: An optimal-time algorithm for slope selection. SIAM J. Comput. 18, 792–810 (1989)
5. Dumitrescu, A., Rote, G., Tóth, C.D.: Monotone paths in planar convex subdivisions and polytopes. In: Bezdek, K., Deza, A., Ye, Y. (eds.) Discrete Geometry and Optimization. Fields Institute Communications, vol. 69, pp. 79–104. Springer (2013)
6. Edmonds, J., Karp, R.M.: Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. J. ACM 19(2), 248–264 (1972)
7. Ezra, E., Sharir, M., Efrat, A.: On the ICP Algorithm. Comput. Geom. Theory Appl. 41, 77–93 (2008)
8. Henze, M., Jaume, R., Keszegh, B.: On the complexity of the partial least-squares matching Voronoi diagram, in. In: Proc. 29th European Workshop Comput. Geom (EuroCG 2013), pp. 193–196 (2013)
9. Jung, I., Lacroix, S.: A robust interest points matching algorithm. In: Proc. ICCV 2001, vol. 2, pp. 538–543 (2001)
10. Kuhn, H.W.: The Hungarian method for the assignment problem. Naval Research Logistics Quarterly 2(1-2), 83–97 (1955)
11. Phillips, J.M., Agarwal, P.K.: On bipartite matching under the RMS distance. In: Proc. 18th Canadian Conf. Comput. Geom (CCCG 2006), pp. 143–146 (2006)
12. Rote, G.: Partial least-squares point matching under translations. In: Proc. 26th European Workshop Comput. Geom (EuroCG 2010), pp. 249–251 (2010)
13. Rote, G.: Long monotone paths in convex subdivisions. In: Proc. 27th European Workshop Comput. Geom. (EuroCG 2011), pp. 183–184 (2011)
14. Shapley, L.S., Scarf, H.: On cores and indivisibility. J. Math. Economics 1, 23–37 (1974)
15. Umeyama, S.: Least-squares estimation of transformation parameters between two point patterns. IEEE Trans. Pattern Anal. Mach. Intell. 13(4), 376–380 (1991)
16. Zikan, K., Silberberg, T.M.: The Frobenius metric in image registration. In: Shapiro, L., Rosenfeld, A. (eds.) Computer Vision and Image Processing, pp. 385–420. Elsevier (1992)

The Batched Predecessor Problem in External Memory*

Michael A. Bender[1,2], Martín Farach-Colton[2,3], Mayank Goswami[4], Dzejla Medjedovic[5], Pablo Montes[1], and Meng-Tsung Tsai[3]

[1] Stony Brook University, Stony Brook NY 11794, USA
{bender,pmontes}@cs.stonybrook.edu
[2] Tokutek, Inc.
[3] Rutgers University, Piscataway NJ 08854, USA
{farach,mtsung.tsai}@cs.rutgers.edu
[4] Max-Planck Institute for Informatics, Saarbrücken 66123, Germany
gmayank@mpi-inf.mpg.de
[5] Sarajevo School of Science and Technology, Sarajevo 71000, Bosnia-Herzegovina
dzejla.medjedovic@ssst.edu.ba

Abstract. We give lower and upper bounds for the batched predecessor problem in external memory. We study tradeoffs between the I/O budget to preprocess a dictionary S versus the I/O requirement to find the predecessor in S of each element in a query set Q. For Q polynomially smaller than S, we give lower bounds in three external-memory models: the I/O comparison model, the I/O pointer-machine model, and the indexability model.

In the comparison I/O model, we show that the batched predecessor problem needs $\Omega(\log_B n)$ I/Os per query element ($n = |S|$) when the preprocessing is bounded by a polynomial. With exponential preprocessing, the problem can be solved faster, in $\Theta((\log_2 n)/B)$ per element. We give the tradeoff that quantifies the minimum preprocessing required for a given searching cost.

In the pointer-machine model, we show that with $\mathcal{O}(n^{4/3-\varepsilon})$ preprocessing for any constant $\varepsilon > 0$, the optimal algorithm cannot perform asymptotically faster than a B-tree. In the indexability model, we exhibit the tradeoff between the redundancy r and access overhead α of the optimal indexing scheme, showing that to report all query answers in $\alpha(x/B)$ I/Os, $\log r = \Omega((B/\alpha^2)\log(n/B))$.

Our lower bounds have matching or nearly matching upper bounds.

1 Introduction

A ***static dictionary*** is a data structure that represents a set $S = \{s_1, s_2, \ldots, s_n\}$ subject to the following operations:

PREPROCESS(S):	Prepare a data structure to answer queries.
SEARCH(q, S):	Return TRUE if $q \in S$ and FALSE otherwise.
PREDECESSOR(q, S):	Return $\max_{s_i \in S}\{s_i < q\}$.

* This research was supported in part by NSF grants CCF 1114809, CCF 1114930, CCF 1217708, IIS 1247726, IIS 1247750, and IIS 1251137.

The traditional static dictionary can be extended to support batched operations. Let $Q = \{q_1, \ldots, q_x\}$. Then, the ***batched predecessor*** problem can be defined as follows:

$$\textsc{BatchedPred}(Q, S): \quad \text{Return } A = \{a_1, \ldots, a_x\}, \text{ where } a_i = \textsc{Predecessor}(q_i, S).$$

In this paper we prove lower bounds on the batched predecessor problem in ***external memory*** [3], that is, when the dictionary is too large to fit into main memory. We study tradeoffs between the searching cost and the cost to preprocess the underlying set S. We present our results in three models: the comparison-based I/O model [3], the pointer-machine I/O model [18], and the indexability model [10, 11].

We focus on query size $x \leq n^c$, for constant $c < 1$. Thus, the query Q can be large, but is still much smaller than the underlying set S. This query size is interesting because, although there is abundant parallelism in the batched query, common approaches such as linear merges and buffering [4, 6, 7] are suboptimal.

Our results show that the batched predecessor problem in external memory cannot be solved asymptotically faster than $\Omega(\log_B n)$ I/Os per query element if the preprocessing is bounded by a polynomial; on the other hand, the problem *can* be solved asymptotically faster, in $\Theta((\log_2 n)/B)$ I/Os, if we impose no constraints on preprocessing. These bounds stand in marked contrast to single-predecessor queries, where one search costs $\Omega(\log_B n)$ even if preprocessing is unlimited.

We assume that S and Q are sorted. Without loss of generality, Q is sorted because Q's sort time is subsumed by the query time. Without loss of generality, S is sorted, as long as the preprocessing time is slightly superlinear. We consider sorted S throughout the paper. For notational convenience, we let $s_1 < s_2 < \cdots < s_n$ and $q_1 < q_2 < \cdots < q_x$, and therefore $a_1 \leq a_2 \leq \cdots \leq a_x$.

Given that S and Q are sorted, an alternative interpretation of this paper is as follows: *how can we optimally merge two sorted lists in external memory?* Specifically, what is the optimal algorithm for merging two sorted lists in external memory when one list is some polynomial factor smaller than the other?

Observe that the naïve linear-scan merging is suboptimal because it takes $\Theta(n/B)$ I/Os, which is greater than the $\mathcal{O}(n^c \log_B n)$ I/Os of a B-tree-based solution. Buffer trees [4, 6, 7] also take $\Theta(n/B)$ I/Os during a terminal flush phase. This paper shows that with polynomial preprocessing, performing independent searches for each element in Q is optimal, but it is possible to do better for higher preprocessing.

Single and Batched Predecessor Problems in RAM. In the comparison model, a single predecessor can be found in $\Theta(\log n)$ time using binary search. The batched predecessor problem is solved in $\Theta(x \log(n/x) + x)$ by combining merging and binary search [13, 14]. The bounds for both problems remain tight for any preprocessing budget.

Pătraşcu and Thorup [15] give tight lower bounds for single predecessor queries in the cell-probe model. We are unaware of prior lower bounds for the batched predecessor problem in the pointer-machine and cell-probe models.

Although batching does not help algorithms that rely on comparisons, Karpinski and Nekrich [12] give an upper bound for this problem in the word-RAM model (bit operations are allowed), which achieves $\mathcal{O}(x)$ for all batches of size $x = \mathcal{O}(\sqrt{\log n})$ ($\mathcal{O}(1)$ per element amortized) with superpolynomial preprocessing.

Batched Predecessor Problem in External Memory. Dittrich et al. [8] consider multisearch problems where queries are simultaneously processed and satisfied by navigating through large data structures on parallel computers. They give a lower bound of $\Omega(x \log_B(n/x) + x/B)$ under stronger assumptions: no duplicates of nodes are allowed, the ith query has to finish before the $(i + 1)$st query starts, and $x < n^{1/(2+\varepsilon)}$, for a constant $\varepsilon > 0$.

Buffering is a standard technique for improving the performance of external-memory algorithms [4, 6, 7]. By buffering, partial work on a set of operations can share an I/O, thus reducing the per-operation I/O cost. Queries can similarly be buffered. In this paper, the number of queries, x, is much smaller than the size, n, of the data structure being queried. As a result, as the partial work on the queries progresses, the query paths can diverge within the larger search structure, eliminating the benefit of buffering.

Goodrich et al. [9] present a general method for performing x simultaneous external memory searches in $\mathcal{O}((n/B + x/B) \log_{M/B}(n/B))$ I/Os when x is large. When x is small, this technique achieves $\mathcal{O}(x \log_B(n/B))$ I/Os with a modified version of the parallel fractional cascading technique of Tamassia and Vitter [19].

Results

We first consider the **comparison-based I/O model** [3]. In this model, the problem cannot be solved faster than $\Omega(\log_B n)$ I/Os per element if preprocessing is polynomial. That is, batching queries is not faster than processing them one by one. With exponential preprocessing, the problem can be solved faster, in $\Theta((\log_2 n)/B)$ I/Os per element. We generalize to show a query-preprocessing tradeoff.

Next we study the **pointer-machine I/O model** [18], which is less restrictive than the comparison I/O model in main memory, but more restrictive in external memory.[1] We show that with preprocessing at most $\mathcal{O}(n^{4/3-\varepsilon})$ for constant $\varepsilon > 0$, the cost per element is again $\Omega(\log_B n)$.

Finally, we turn to the more general **indexability model** [10, 11]. This model is frequently used to describe reporting problems, and it focuses on bounding the number of disk blocks that contain the answers to the query subject to the space limit of the data structure; the searching cost is ignored. Here, the ***redundancy parameter r*** measures the number of times an element is stored in the data structure, and the ***access overhead parameter α*** captures how far the reporting cost is from the optimal.

We show that to report all query answers in $\alpha(x/B)$ I/Os, $r = (n/B)^{\Omega(B/\alpha^2)}$. The lower bounds in this model also hold in the previous two models. This result shows that it is impossible to obtain $\mathcal{O}(1/B)$ per element unless the space used by the data structure is exponential, which corresponds to the situation in RAM, where exponential preprocessing is required to achieve $\mathcal{O}(1)$ amortized time per query element [12].

The rest of this section formally outlines our results.

[1] An algorithm can perform arbitrary computations in RAM, but a disk block can be accessed only via a pointer that has been seen at some point in past.

Theorem 1 (Lower and Upper Bound, Unrestricted Preprocessing, I/O Comparison Model). *Let S be a set of size n and Q a set of size $x \le n^c$, $0 \le c < 1$. In the I/O comparison model, computing* $\textsc{BatchedPred}(Q, S)$ *requires*

$$\Omega\left(\frac{x}{B}\log\frac{n}{xB} + \frac{x}{B}\right)$$

I/Os in the worst-case, no matter the preprocessing. There exists a comparison-based algorithm matching this bound.

Traditional information-theoretic techniques give tight sorting-like lower bounds for this problem in the RAM model. In external memory, the analogous approach yields a lower bound of $\Omega\left(\frac{x}{B}\log_{M/B}\frac{n}{x} + \frac{x}{B}\right)$. On the other hand, repeated finger searching in a B-tree yields an upper bound of $\mathcal{O}(x \log_B n)$. Theorem 1 shows that both bounds are weak, and that in external memory this problem has a complexity that is between sorting and searching.

We can interpret results in the comparison model through the amount of information that can be learned from each I/O. For searching, a block input reduces the choices for the target position of the element by a factor of B, thus learning $\log B$ *bits of information*. For sorting, a block input learns up to $\log\binom{M}{B} = \Theta(B\log(M/B))$ bits (obtained by counting the ways that an incoming block can intersperse with elements resident in main memory). Theorem 1 demonstrates that in the batched predecessor problem, the optimal, unbounded-preprocessing algorithm learns B bits per I/O, more than for searching but less than for sorting.

The following theorem captures the tradeoff between the searching and preprocessing: at one end of the spectrum lies a B-tree ($j = 1$) with linear construction time and $\log_B n$ searching cost per element, and on the other end is the parallel binary search ($j = B$) with exponential preprocessing cost and $(\log_2 n)/B$ searching cost. This tradeoff shows that even to obtain a performance that is only twice as fast as that of a B-tree, quadratic preprocessing is necessary. To learn up to $j\log(B/j+1)$ bits per I/O, the algorithm needs to spend $n^{\Omega(j)}$ in preprocessing.

Theorem 2 (Search-Preprocessing Tradeoff, I/O Comparison Model). *Let S be a set of size n and Q a set of size $x \le n^c$, $0 \le c < 1$. In the I/O comparison model, computing* $\textsc{BatchedPred}(Q, S)$ *in $\mathcal{O}((x\log_{B/j+1} n)/j)$ I/Os requires that* $\textsc{Preprocessing}(S)$ *use $n^{\Omega(j)}$ blocks of space and I/Os.*

In order to show results in the I/O pointer-machine model, we define a graph whose nodes are the blocks on disk of the data structure and whose edges are the pointers between blocks. Since a block has size B, it can contain at most B pointers, and thus the graph is fairly sparse. We show that any such sparse graph has a large set of nodes that are far apart. If the algorithm must visit those well-separated nodes, then it must perform many I/Os. The crux of the proof is that, as the preprocessing increases, the redundancy of the data structure increases, thus making it hard to pin down specific locations of the data structure that must be visited. We show that if the data structure is reasonable in size—in our case $\mathcal{O}(n^{4/3-\varepsilon})$—then we can still find a large, well dispersed set of nodes that must be visited, thus establishing the following lower bound:

Theorem 3 (Lower Bound, I/O Pointer-Machine Model). *Let S be a set of size n. In the I/O pointer-machine model, if* $\text{PREPROCESSING}(S)$ *uses* $\mathcal{O}(n^{4/3-\varepsilon})$ *blocks of space and I/Os, for any constant* $\varepsilon > 0$*, then there exists a constant c and a set Q of size n^c such that computing* $\text{BATCHEDPRED}(Q, S)$ *requires* $\Omega(x \log_B(n/x) + x/B)$ *I/Os.*

We note that in this theorem, c is a function of ε in that, the smaller the preprocessing, the larger the set for which the lower bound can be established.

Finally, we consider the indexability model [10, 11], where we show:

Theorem 4 ($r - \alpha$ Tradeoff, Indexability Model). *In the indexability model, any indexing scheme for the batched predecessor problem with access overhead $\alpha \leq \sqrt{B}/4$ has redundancy r satisfying* $\log r = \Omega\left(B \log(n/B)/\alpha^2\right)$.

A crucial ingredient in our proof is a well-known result from extremal set theory due to Rödl [16]. Partly due to the techniques we use and partly due to the generality of this model, we do not get lower bounds for query time exceeding $Q/\sqrt{B}$, which was possible in the previous two models.

2 Batched Predecessor in the I/O Comparison Model

In this section we give the lower bound for when preprocessing is unrestricted. Then we study the tradeoff between preprocessing and the optimal number of I/Os.

2.1 Lower Bounds for Unrestricted Space/Preprocessing

We begin with the definition of a search interval.

Definition 5 (*Search interval*). *At step t of an execution, the search interval $S_i^t = [\ell_i^t, r_i^t]$ for an element q_i comprises those elements in S that are still potential values for a_i, given the information that the algorithm has learned so far. When there is no ambiguity, the superscript t is omitted.*

Proof of Theorem 1 (Lower Bound). Consider the following problem instance:

1. For all q_i, $|S_i| = n/x$. That is, all elements in Q have been given the first $\log x$ bits of information about where they belong in S.
2. For all i and j ($1 \leq i \neq j \leq x$), $S_i \cap S_j = \emptyset$. That is, search intervals are disjoint.

We do not charge the algorithm for transferring elements of Q between main memory and disk. This accounting scheme is equivalent to allowing all elements of Q to reside in main memory at all times while still having the entire memory free for other manipulations. Storing Q in main memory does not provide the algorithm with any additional information, since the sorted order of Q is already known.

Now we only consider I/Os of elements in S. Denote a block being input as $b = (b_1, \ldots, b_B)$. Observe that every b_i ($1 \leq i \leq B$) belongs to at most one S_j. The element b_i acts as a **pivot** and helps q_j learn at most one bit of information—by shrinking S_j to its left or its right half.

Since a single pivot gives at most one bit of information, the entire block b can supply at most B bits, during an entire execution of $\textsc{BatchedPred}(Q, S)$.

We require the algorithm to identify the final block in S where each q_i belongs. Thus, the total number of bits that the algorithm needs to learn to solve the problem is $\Omega(x \log(n/xB))$. Along with the scan bound to output the answer, the minimum number of block transfers required to solve the problem is $\Omega\left(\frac{x}{B} \log \frac{n}{xB} + \frac{x}{B}\right)$. □

We devise a matching algorithm (assuming $B \log n < M$), which has $\mathcal{O}(n^B)$ preprocessing cost. This algorithm has huge preprocessing costs but establishes that the lower bound from Theorem 1 is tight.

Proof of Theorem 1 (Upper Bound). The algorithm processes Q in batches of size B, one batch at a time. A single batch is processed by simultaneously performing binary search on all elements of the batch until they find their rank within S.

In the preprocessing phase, the algorithm produces all $\binom{n}{B}$ possible blocks. The algorithm also constructs a perfectly balanced binary search tree T on S. The former takes at most $B\binom{n}{B}$ I/Os, which is $\mathcal{O}(n^B)$, while the latter has a linear cost. The $\binom{n}{B}$ blocks are laid out in a lexicographical order in external memory, and it takes $B \log n$ bits to address the location of any block. □

2.2 Preprocessing-Searching Tradeoffs

We give a lower bound on the space required by the batched predecessor problem when the budget for searching is limited. We prove Theorem 2 by proving Theorem 7.

Definition 6. *An I/O containing elements of S is a **j-parallelization I/O** if j distinct elements of Q acquire bits of information during this I/O.*

Theorem 7. *For $x \leq n^{1-\varepsilon}$ ($0 < \varepsilon \leq 1$) and a constant $\gamma > 0$, any algorithm that solves* $\textsc{BatchedPred}(Q, S)$ *in at most $(\gamma x \log n)/(j \log(B/j + 1)) + x/B$ I/Os requires at least $\left(\varepsilon j n^{\varepsilon/2}/2e\gamma B\right)^{\varepsilon j/2\gamma}$ I/Os for preprocessing in the worst case.*

Proof. The proof is by a deterministic adversary argument. In the beginning, the adversary partitions S into x equal-sized chunks $C_1, \ldots, C_x$, and places each query element into a separate chunk (i.e., $S_i = C_i$). Now each element knows $\log x \leq (1 - \varepsilon) \log n$ bits of information. Each element is additionally given half of the number of bits that remain to be learned. This leaves another $T \geq (\varepsilon x \log n)/2$ total bits yet to be discovered. As in the proof of Theorem 1, we do not charge for the inputs of elements in Q, thereby stipulating that all remaining bits to be learned are through the inputs of elements of S.

Lemma 8. *To learn T bits in at most $(\gamma x \log n)/(j \log(B/j + 1))$ I/Os, there must be at least one I/O in which the algorithm learns at least $(j \log(B/j + 1))/a$ bits, where $a = 2\gamma/\varepsilon$.*

If multiple I/Os learn at least $(j \log(B/j + 1))/a$ bits, consider the last such I/O during the algorithm execution. Denote the contents of the I/O as $b_i = (p_1, \ldots, p_B)$.

Lemma 9. *The maximum number of bits an I/O can learn while parallelizing d elements is $d \log(B/d + 1)$.*

Lemma 10. *The I/O b_i parallelizes at least j/a elements.*

Proof. Given that the most bits an I/O can learn while parallelizing $j/a - 1$ elements is $(j/a-1)\log(B/(j/a-1)+1)$ bits. For all $a \geq 1$ and $j \geq 2$, $\frac{j}{a}\log\left(\frac{B}{j}+1\right) > \left(\frac{j}{a}-1\right)\log\left(\frac{B}{j/a-1}+1\right)$. Thus, we can conclude that with the block transfer of b_i, the algorithm must have parallelized strictly more than $j/a - 1$ distinct elements. □

We focus our attention on an arbitrarily chosen group of j/a elements parallelized during the transfer of $b_i = \{p_1, \ldots, p_B\}$, which we call $q_1, \ldots, q_{j/a}$.

Lemma 11. *For every q_u parallelized during the transfer of b_i there is at least one pivot p_v, $1 \leq v \leq B$, such that $p_v \in S_u$.*

Consider the vector $V = (S_1, S_2, \ldots, S_{j/a})$ where S_u denotes the search interval of q_u right before the input of b_i.

Each element of Q has acquired at least $(1-\varepsilon/2)\log n$ bits, $(\varepsilon \log n)/2$ of which were given for free after the initial $(1-\varepsilon)\log n$. For any i, the total number of distinct choices for S_i in the vector V is at least $n^{\varepsilon/2}$, because the element could have been sent to any of these $n^{\varepsilon/2}$-sized ranges in the initial n^{ε} range. We obtain the following:

Lemma 12. *The number of distinct choices for V at the time of parallelization is at least $n^{j\varepsilon/2a}$.*

Lemma 13. *For each of the $n^{j\varepsilon/2a}$ choices of $V = (S_1, \ldots, S_{j/a})$ (arising from the $n^{\varepsilon/2}$ choices for each S_i), there must exist a block with pivots $p_1, p_2, \ldots, p_{j/a}$, such that $p_k \in S_k$.*

If the algorithm did not preprocess a block for each vector choice, the adversary could scan all blocks, find a vector for which no block exists, and assign those search intervals to $q_1, \ldots, q_{j/a}$, thus avoiding parallelization.

The same block can serve multiple vector choices, because the block has B elements and we are parallelizing only j/a elements. The next lemma quantifies the maximum number of vectors covered by one block.

Lemma 14. *A block can cover at most $\binom{B}{j/a}$ distinct vector choices.*

As a consequence, the minimum number of blocks the algorithm needs to preprocess is at least $n^{j\varepsilon/2a}/\binom{B}{j/a} \geq \left(n^{\varepsilon/2}/(eaB/j)\right)^{j/a}$. Substituting for the value of a, we get that the minimum preprocessing is at least $\left(\varepsilon j n^{\varepsilon/2}/2e\gamma B\right)^{\varepsilon j/2\gamma}$. □

Algorithms. An algorithm that runs in $\mathcal{O}((x \log n)/j \log(B/j+1) + x/B)$ I/Os follows an idea similar to the optimal algorithm for unrestricted preprocessing. The difference is that we preprocess $\binom{n}{j}$ blocks, where each block correspond to a distinct combination of some j elements. The block will contain B/j evenly spaced pivots for each element. The searching algorithm uses batches of size j.

3 Batched Predecessor in the I/O Pointer-Machine Model

Here we analyze the batched predecessor problem in the I/O pointer-machine model. We show that if the preprocessing time is $\mathcal{O}(n^{4/3-\varepsilon})$ for any constant $\varepsilon > 0$, then there exists a query set Q of size x such that reporting $\text{BATCHEDPRED}(Q, S)$ requires $\Omega(x/B + x \log_B n/x)$ I/Os. Before proving our theorem, we briefly describe the model.

I/O Pointer Machine Model. The I/O pointer machine model [18] is a generalization of the pointer machine model introduced by Tarjan [21]. Many results in range reporting have been obtained in this model [1, 2].

To answer $\text{BATCHEDPRED}(Q, S)$, an algorithm preprocesses S and builds a data structure comprised of n^k blocks, where k is a constant to be determined later. We use a directed graph $\mathcal{G} = (V, E)$ to represent the n^k blocks and their associated directed pointers. Every algorithm that answers $\text{BATCHEDPRED}(Q, S)$ begins at the start node v_0 in V and at each step picks a directed edge to follow from those seen so far. Thus, the nodes in a computation are all reachable from v_0. Furthermore, each fetched node contains elements from S, and the computation cannot terminate until the visited set of elements is a superset of the answer set A. A node in V contains at most B elements from S and at most B pointers to other nodes.

Let $\mathcal{L}(W)$ be the union of the elements contained in a node set W, and let $\mathcal{N}(a)$ be the set of nodes containing element a. We say that a node set W ***covers*** a set of elements A if $A \subseteq \mathcal{L}(W)$. An algorithm for computing A can be modeled as the union of a set of paths from v_0 to each node in a node set W that covers A.

To prove a lower bound on $\text{BATCHEDPRED}(Q, S)$, we show that there is a query set Q whose answer set A requires many I/Os. In other words, for every node set W that covers A, a connected subgraph spanning W contains many nodes. We achieve this result by showing that there is a set A such that, for every pair of nodes $a_1, a_2 \in A$, the distance between $\mathcal{N}(a_1)$ and $\mathcal{N}(a_2)$ is large, that is, all the nodes in $\mathcal{N}(a_1)$ are far from all the nodes in $\mathcal{N}(a_2)$. Since the elements of A can appear in more than one node, we need to guarantee that the node set V of $\mathcal{G}$ is not too large; otherwise the distance between $\mathcal{N}(a_1)$ and $\mathcal{N}(a_2)$ can be very small. For example, if $|V| \geq \binom{n}{2}$, every pair of elements can share a node, and a data structure exists whose minimum pairwise distance between any $\mathcal{N}(a_1)$ and $\mathcal{N}(a_2)$ is 0.

First, we introduce two measures of distance between nodes in any (undirected or directed) graph $G = (V, E)$. Let $d_G(u, v)$ be the length of the shortest (di-)path from node u to node v in G. Furthermore, let $\Lambda_G(u, v) = \min_{w \in V} (d_G(w, u) + d_G(w, v))$. Thus, $\Lambda_G(u, v) = d_G(u, v)$ for undirected graphs, but not necessarily for directed graphs.

For each $W \subseteq V$, define $f_G(W)$ to be the minimum number of nodes in any connected subgraph H such that (1) the node set of H contains $W \cup \{v_0\}$ and (2) H contains a path from v_0 to each $v \in W$. Observe that $f_G(\{u, v\}) \geq \Lambda_G(u, v)$. The following lemma gives a more general lower bound for $f_G(W)$. In other words, the size of the graph containing nodes of W is linear in the minimum pairwise distance within W.

Lemma 15. *For any directed graph $G = (V, E)$ and any $W \subseteq V$ of size $|W| \geq 2$, $f_G(W) \geq r_W |W|/2$, where $r_W = \min_{u,v \in W, u \neq v} \Lambda_G(u, v)$.*

Proof Sketch. Consider the undirected version of G, and consider a TSP of the nodes in W. It must have length $r_W|W|$. Any tree that spans W must therefore have size at least $r_W|W|/2$. Finally, $f_G(W)$ contains a tree that spans W. □

Our next goal is to find a query set Q such that every node set W that covers the corresponded answer set A has a large r_W. The answer set A will be an independent set of a certain kind, that we define next. For a directed graph $G = (V, E)$ and an integer $r > 0$, we say that a set of nodes $I \subseteq V$ is ***r-independent*** if $\Lambda_G(u, v) > r$ for all $u, v \in I$ where $u \neq v$. The next lemma guarantees a substantial r-independent set.

Lemma 16. *Given a directed graph $G = (V, E)$, where each node has out-degree at most $B \geq 2$, there exists an r-independent set I of size at least $\frac{|V|^2}{|V|+4r|V|B^r}$.*

Proof. Construct an undirected graph $H = (U, F)$ such that $U = V$ and $(u, v) \in F$ iff $\Lambda_G(u, v) \in [1, r]$. Then, H has at most $2r|V|B^r$ edges. By Turán's Theorem [20], there exists an independent set of the desired size in H, which corresponds to an r-independent set in G, completing the proof. □

In addition to r-independence, we want the elements in A to occur in few blocks, in order to control the possible choices of the node set W that covers A. We define the ***redundancy*** of an element a to be $|\mathcal{N}(a)|$. Because there are n^k blocks and each block has at most B elements, the average redundancy is $\mathcal{O}(n^{k-1}B)$. We say that an element has ***low redundancy*** if its redundancy is at most twice the average. We show that there exists an r-independent set I of size n^{ε} (here ε depends on r) such that no two blocks share the same low-redundancy element. We will then construct our query set Q using this set of low-redundancy elements in this r-independent set.[2]

Finally, we add enough edges to place all occurrences of every low-redundancy element within $\rho < r/2$ of all other occurrences of that element. We show that we can do this by adding few edges to each node, therefore maintaining the sparsity of G. Since this augmented graph also contains a large r-independent set, all the nodes of this set cannot share any low-redundancy elements.

The following lemma shows that nodes sharing low-redundancy elements can be connected with low diameter and small degrees.

Lemma 17. *For any $k > 0$ and $m > k$ there exists an undirected k-regular graph H of order m having diameter $\log_{k-1} m + o(\log_{k-1} m)$.*

Proof. In [5], Bollobás shows that a random k-regular graph has the desired diameter with probability close to 1. Thus there exists some graph satisfying the constraints. □

Consider two blocks B_1 and B_2 in the r-independent set I above, and let a and b be two low-redundancy elements such that $a \in B_1, b \notin B_1$ and $a \notin B_2, b \in B_2$. Any other pair of blocks $B_1^{'}$ and $B_2^{'}$ that contain a and b respectively must be at least $(r - 2\rho)$ apart, since $B_i^{'}$ is at most ρ apart from B_i. By this argument, every node set W that covers A has $r_W \geq (r - 2\rho)$. Now, by Lemma 15, we get a lower bound of $\Omega((r - 2\rho)|W|)$ on the query complexity of Q. We choose $r = c_1 \log_B(n/x)$ and get

[2] Our construction does not work if the query set contains high redundancy elements, because high redundancy elements might be placed in every block.

$\rho = c_2 \log_B(n/x)$ for appropriate constants $c_1 > 2c_2$. This is the part where we require the assumption that $k < 4/3$ as shown in Theorem 3, where n^k was the size of the entire data structure. We then apply Lemma 16 to obtain that $|W| = \Omega(x)$.

Proof of Theorem 3. We partition S into S_ℓ and S_h by the redundancy of elements in these n^k blocks and claim that there exists $A \subseteq S_\ell$ such that query time for the corresponded Q matches the lower bound.

Let S_ℓ be the set of elements of redundancy no more than $2Bn^k/n$ (i.e., twice of the average redundancy). The rest of elements belong to S_h. By the Markov inequality, we have $|S_\ell| = \Theta(n)$ and $|S_h| \leq n/2$. Let $\mathcal{G} = (V, E)$ represent the connections between the n^k blocks as the above stated. We partition V into V_1 and V_2 such that V_1 is the set of blocks containing some elements in S_ℓ and $V_2 = V \setminus V_1$. Since each block can at most contain B elements in S_ℓ, $|V_1| = \Omega(n/B)$.

Then, we add some additional pointers to $\mathcal{G}$ and obtain a new graph $\mathcal{G}'$ such that, for each $e \in S_\ell$, every pair $u, v \in \mathcal{N}(e)$ has small $\Lambda_{\mathcal{G}'}(u, v)$. We achieve this by, for each $e \in S_\ell$, introducing graph H_e to connect all the n^k blocks containing element e such that the diameter in H_e is small and the degree for each node in H_e is $\mathcal{O}(B^\delta)$ for some constant δ. By Lemma 17, the diameter of H_e can be as small as

$$\rho \leq \frac{1}{\delta} \log_B |H_e| + o(\log_B |H_e|) \leq \frac{k-1}{\delta} \log_B n + o(\log_B n).$$

We claim that the graph $\mathcal{G}'$ has a $(2\rho + \varepsilon)$-independent set of size n^c, for some constants $\varepsilon, c > 0$. For the purpose, we construct an undirected graph $H(V_1, F)$ such that $(u, v) \in F$ iff $\Lambda_{\mathcal{G}'}(u, v) \leq r$. Since the degree of each node in $\mathcal{G}'$ is bounded by $\mathcal{O}(B^{\delta+1})$, by Lemma 16, there exists an r-independent set I of size

$$|I| \geq \frac{|V_1|^2}{|V_1| + 4r|V|\mathcal{O}(B^{r(\delta+1)})} \geq \frac{n^{2-k}}{4r\mathcal{O}(B^{r(\delta+1)+2})} = n^c.$$

Then, $r = ((2 - k - c)\log_B n)/(\delta + 1) + o(\log_B n)$. To satisfy the condition made in the claim, let $r > 2\rho$. Hence, $(2 - k - c)/(\delta + 1) > 2(k - 1)/\delta$. Then, $k \to 4/3$ for sufficiently large δ. Observe that, for each $e \in S_\ell$, e is contained in at most one node in I; in addition, for every pair $e_1, e_2 \in S_\ell$ where e_1, e_2 are contained in separated nodes in I, then $\Lambda_{\mathcal{G}'}(u, v) \geq \varepsilon$ for any $u \ni e_1, v \ni e_2$. By Lemma 15, we are done. □

4 Batched Predecessor in the Indexability Model

This section analyzes the batched predecessor problem in the indexability model [10, 11]. This model is used to analyze reporting problems by focusing on the number of blocks that an algorithm must access to report all the query results. Lower bounds on queries are obtained solely based on how many blocks were preprocessed. The search cost is ignored—the blocks containing the answers are given to the algorithm for free.

A ***workload*** is given by a pair $\mathcal{W} = (S, \mathcal{A})$, where S is the set of n input objects, and $\mathcal{A}$ is a set of subsets of S—the output to the queries. An ***indexing scheme*** $\mathcal{I}$ for a given workload $\mathcal{W}$ is given by a collection $\mathcal{B}$ of B-sized subsets of S such that $S = \cup\mathcal{B}$; each $b \in \mathcal{B}$ is called a block.

An indexing scheme has two parameters associated with it. The first parameter, called the ***redundancy***, represents the average number of times an element is replicated (i.e., an indexing scheme with redundancy r uses $r\lceil n/B\rceil$ blocks). The second parameter is called the ***access overhead***. Given a query with answer A, the query time is $\min\{|\mathcal{B}^{'}| : \mathcal{B}^{'} \subseteq \mathcal{B}, A \subseteq \cup\mathcal{B}^{'}\}$, because this is the minimum number of blocks that contain all the answers to the query. If the size of A is x, then the best indexing scheme would require a query time of $\lceil x/B\rceil$. The access overhead of an indexing scheme is the factor by which it is suboptimal. An indexing scheme with access overhead α uses $\alpha\lceil x/B\rceil$ I/Os to answer a query of size x in the worst case.

Every lower bound in this model applies to our previous two models as well. To show the tradeoff between α and r, we use the Redundancy Theorem from [11, 17]:

Theorem 18 (Redundancy Theorem [11, 17]). *For a workload $\mathcal{W} = (S, \mathcal{A})$ where $\mathcal{A} = \{A_1, \cdots, A_m\}$, let $\mathcal{I}$ be an indexing scheme with access overhead $\alpha \le \sqrt{B}/4$ such that for any $1 \le i, j \le m$, $i \ne j$, $|A_i| \ge B/2$ and $|A_i \cap A_j| \le B/(16\alpha^2)$. Then the redundancy of $\mathcal{I}$ is bounded by $r \ge \frac{1}{12n}\sum_{i=1}^{m}|A_i|$.*

Proof of Theorem 4. For the sake of the lower bound, we restrict to queries where all the reported predecessors reported are distinct. To use the redundancy theorem, we want to create as many queries as possible.

Call a family of k-element subsets of S β-sparse if any two members of the family intersect in less than β elements. The size $C(n, k, \beta)$ of a maximal β-sparse family is crucial to our analysis. For a fixed k and β this was conjectured to be asymptotically equal to $\binom{n}{\beta}/\binom{k}{\beta}$ by Erdös and Hanani and later proven by Rödl in [16]. Thus, for large enough n, $C(n, k, \beta) = \Omega(\binom{n}{\beta}/\binom{k}{\beta})$.

We now pick a $(B/2)$-element, $B/(16\alpha^2)$-sparse family of S, where α is the access overhead of $\mathcal{I}$. The result in [16] gives us that

$$C\left(n, \frac{B}{2}, \frac{B}{16\alpha^2}\right) = \Omega\left(\binom{n}{B/(16\alpha^2)}\Big/\binom{B/2}{B/(16\alpha^2)}\right).$$

Thus, there are at least $(2n/eB)^{B/(16\alpha^2)}$ subsets of size $B/2$ such that any pair intersects in at most $B/(16\alpha^2)$ elements. The Redundancy Theorem then implies that the redundancy r is greater than or equal to $(n/B)^{\Omega(B/\alpha^2)}$, completing the proof. □

We describe an indexing scheme that is off from the lower bound by a factor α.

Theorem 19 (Indexing Scheme for the Batched Predecessor Problem). *Given any $\alpha \le \sqrt{B}$, there exists an indexing scheme $\mathcal{I}_\alpha$ for the batched predecessor problem with access overhead α^2 and redundancy $r = \mathcal{O}((n/B)^{B/\alpha^2})$*

Proof. Call a family of k-element subsets of S β-dense if any subset of S of size β is contained in at least one member from this family. Let $c(n, k, \beta)$ denote the minimum number of elements of such a β-dense family. Rödl [16] proves that for a fixed k and β,

$$\lim_{n\to\infty} c(n, k, \beta)\binom{k}{\beta}\binom{n}{\beta}^{-1} = 1,$$

and thus, for large enough n, $c(n, k, \beta) = \mathcal{O}(\binom{n}{\beta}/\binom{k}{\beta})$.

The indexing scheme $\mathcal{I}_\alpha$ consists of all sets in a B-element, (B/α^2)-dense family. By the above, the size of $\mathcal{I}_\alpha$ is $\mathcal{O}((n/B)^{B/\alpha^2})$.

Given a query answer $A = \{a_1, \cdots, a_x\}$ of size x, fix $1 \le i < \lceil x/B \rceil$ and consider the B-element sets $C_i = \{a_{(i-1)B}, \cdots, a_{iB}\}$ ($C_{\lceil x/B \rceil}$ may have less than B elements). Since $\mathcal{I}_\alpha$ is an indexing scheme, we are told all the blocks in $\mathcal{I}_\alpha$ that contain the a_is. By construction, there exists a block in $\mathcal{I}_\alpha$ that contains a $1/\alpha^2$ fraction of C_i. In at most α^2 I/Os we can output C_i, by reporting B/α^2 elements in every I/O. The number of I/Os needed to answer the entire answer A is thus $\alpha^2 \lceil x/B \rceil$, which proves the theorem. □

References

1. Afshani, P., Arge, L., Larsen, K.D.: Orthogonal range reporting: Query lower bounds, optimal structures in 3-d, and higher-dimensional improvements. In: 26th Annual Symposium on Computational Geometry (SoCG), pp. 240–246 (2010)
2. Afshani, P., Arge, L., Larsen, K.G.: Higher-dimensional orthogonal range reporting and rectangle stabbing in the pointer machine model. In: 28th Annual Symposium on Computational Geometry (SoCG), pp. 323–332 (2012)
3. Aggarwal, A., Vitter, J.S.: The input/output complexity of sorting and related problems. Commun. ACM 31, 1116–1127 (1988)
4. Arge, L.: The buffer tree: A technique for designing batched external data structures. Algorithmica 37(1), 1–24 (2003)
5. Bollobás, B., Fernandez de la Vega, W.: The diameter of random regular graphs. Combinatorica 2(2), 125–134 (1982)
6. Brodal, G.S., Fagerberg, R.: Lower bounds for external memory dictionaries. In: 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 546–554 (2003)
7. Buchsbaum, A.L., Goldwasser, M., Venkatasubramanian, S., Westbrook, J.R.: On external memory graph traversal. In: 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 859–860 (2000)
8. Dittrich, W., Hutchinson, D., Maheshwari, A.: Blocking in parallel multisearch problems (extended abstract). In: 10th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), pp. 98–107 (1998)
9. Goodrich, M.T., Tsay, J.J., Cheng, N.C., Vitter, J., Vengroff, D.E., Vitter, J.S.: External-memory computational geometry. In: 1993 IEEE 34th Annual Foundations of Computer Science (FOCS), pp. 714–723 (1993)
10. Hellerstein, J.M., Koutsoupias, E., Papadimitriou, C.H.: On the analysis of indexing schemes. In: 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), pp. 249–256 (1997)
11. Hellerstein, J.M., Koutsoupias, E., Miranker, D.P., Papadimitriou, C.H., Samoladas, V.: On a model of indexability and its bounds for range queries. J. ACM 49, 35–55 (2002)
12. Karpinski, M., Nekrich, Y.: Predecessor queries in constant time? In: Brodal, G.S., Leonardi, S. (eds.) ESA 2005. LNCS, vol. 3669, pp. 238–248. Springer, Heidelberg (2005)
13. Knudsen, M., Larsen, K.: I/O-complexity of comparison and permutation problems. Master's thesis, DAIMI (November 1992)
14. Knuth, D.E.: The Art of Computer Programming: Sorting and Searching, vol. 3. Addison-Wesley (1973)
15. Pătraşcu, M., Thorup, M.: Time-space trade-offs for predecessor search. In: 38th Annual ACM Symposium on Theory of Computing (STOC), pp. 232–240 (2006)
16. Rödl, V.: On a packing and covering problem. European Journal of Combinatorics 6(1), 69–78 (1985)

17. Samoladas, V., Miranker, D.P.: A lower bound theorem for indexing schemes and its application to multidimensional range queries. In: 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), pp. 44–51 (1998)
18. Subramanian, S., Ramaswamy, S.: The p-range tree: A new data structure for range searching in secondary memory. In: Sixth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 378–387 (1995)
19. Tamassia, R., Vitter, J.S.: Optimal cooperative search in fractional cascaded data structures. In: Algorithmica, pp. 307–316 (1990)
20. Tao, T., Vu, V.H.: Additive Combinatorics. Cambridge University Press (2009)
21. Tarjan, R.E.: A class of algorithms which require nonlinear time to maintain disjoint sets. Journal of Computer and System Sciences 18(2), 110–127 (1979)

Polynomial Decompositions in Polynomial Time

Arnab Bhattacharyya

Indian Institute of Science, Bangalore, India
arnabb@csa.iisc.ernet.in

Abstract. Fix a prime p. Given a positive integer k, a vector of positive integers $\boldsymbol{\Delta} = (\Delta_1, \Delta_2, \ldots, \Delta_k)$ and a function $\Gamma : \mathbb{F}_p^k \to \mathbb{F}_p$, we say that a function $P : \mathbb{F}_p^n \to \mathbb{F}_p$ is $(k, \boldsymbol{\Delta}, \Gamma)$-*structured* if there exist polynomials $P_1, P_2, \ldots, P_k : \mathbb{F}_p^n \to \mathbb{F}_p$ with each $\deg(P_i) \leqslant \Delta_i$ such that for all $x \in \mathbb{F}_p^n$,

$$P(x) = \Gamma(P_1(x), P_2(x), \ldots, P_k(x)).$$

For instance, an n-variate polynomial over the field $\mathbb{F}_p$ of total degree d factors nontrivially exactly when it is $(2, (d-1, d-1), \mathsf{prod})$-structured where $\mathsf{prod}(a, b) = a \cdot b$.

We show that if $p > d$, then for **any** fixed $k, \boldsymbol{\Delta}, \Gamma$, we can decide whether a given polynomial $P(x_1, x_2, \ldots, x_n)$ of degree d is $(k, \boldsymbol{\Delta}, \Gamma)$-structured and if so, find a witnessing decomposition. The algorithm takes poly(n) time. Our approach is based on higher-order Fourier analysis.

1 Introduction

(Linear) Fourier analysis over a finite field $\mathbb{F}_p$ studies the structure of exponentials of linear functions, i.e. functions of the form $\omega^{\ell(x)}$ where $\ell : \mathbb{F}_p^n \to \mathbb{F}_p$ is a linear function and $\omega = e^{2\pi i/p}$ is the p'th root of unity. Fourier analysis over finite fields has, by now, a rich history of widespread success in theoretical computer science. Here is a sample of applications: coding theory, computational learning theory, influence of variables in boolean functions, probabilistically checkable proofs, cryptography, communication complexity, and quantum computing. For more, consult the lovely survey of de Wolf [dW08].

Higher-order Fourier analysis is a novel generalization of Fourier analysis. In higher-order Fourier analysis over finite fields, we study the structure of exponentials of low-degree polynomials, i.e. functions of the form $\omega^{Q(x)}$ where $Q : \mathbb{F}_p^n \to \mathbb{F}_p$ is a polynomial[1] of bounded degree. The theory (although conceptually originating with the classical equidistribution results of Weyl) really got its start from the spectacular proof by Gowers of Szemerédi's theorem [Gow98,Gow01], where the Gowers norm was introduced. Another significant influence was the work of Host and Kra [HK05] in ergodic theory. Subsequently, Green, Tao and Ziegler through several works [GT08,GT10,GTZ11,GTZ,TZ10,TZ12] largely completed

[1] Throughout, our functions are of n variables over $\mathbb{F}_p$, where n is growing but p is fixed.

the research program of understanding the relationships between different aspects of the theory. The book [Tao12] by Tao on the subject surveys the current state of knowledge.

Green, Tao and Ziegler applied higher-order Fourier analysis to find asymptotics for various linear patterns in the prime numbers. In theoretical computer science, low-degree polynomials over finite fields has long been under consideration due to the use of arithmetization. Specifically, there is a long history of testing whether a function is correlated with a low-degree polynomial, and higher-order Fourier analysis can be immediately phrased in this context. In fact, it was shown in [BCSX11,BGS10,BFL13,BFH$^+$13] that higher-order Fourier analysis can be used to analyze tests not only for low-degreeness but also for any locally characterized affine-invariant property (see the cited papers for definitions). Besides property testing, the Gowers norm has also been used in computer science to show worst case to average case reductions for polynomials [KL08] and XOR lemmas for polynomials [VW08].

In this paper, we demonstrate a new algorithmic application of higher-order Fourier analysis. Consider the following family of properties of functions over a finite field $\mathbb{F}_p$ of fixed prime order p.

Definition 1. *Given a positive integer k, a vector of positive integers $\boldsymbol{\Delta} = (\Delta_1, \Delta_2, \ldots, \Delta_k)$ and a function $\Gamma : \mathbb{F}_p^k \to \mathbb{F}_p$, we say that a function $P : \mathbb{F}_p^n \to \mathbb{F}_p$ is $(k, \boldsymbol{\Delta}, \Gamma)$-*structured *if there exist polynomials $P_1, P_2, \ldots, P_k : \mathbb{F}_p^n \to \mathbb{F}_p$ with each* $\deg(P_i) \leqslant \Delta_i$ *such that for all $x \in \mathbb{F}_p^n$,*

$$P(x) = \Gamma(P_1(x), P_2(x), \ldots, P_k(x)).$$

*The polynomials $P_1, \ldots, P_k$ are said to form a $(k, \boldsymbol{\Delta}, \Gamma)$-*decomposition.

For instance, an n-variate polynomial over the field $\mathbb{F}_p$ of total degree d factors nontrivially exactly when it is $(2, (d-1, d-1), \mathsf{prod})$-structured where $\mathsf{prod}(a, b) = a \cdot b$. Informally, a *degree-structural property* refers to a property from the family of $(k, \boldsymbol{\Delta}, \Gamma)$-structured properties.

Our main result is that every degree-structural property can be decided in polynomial time:

Theorem 1. *For every positive integer k, every vector of positive integers $\boldsymbol{\Delta} = (\Delta_1, \Delta_2, \ldots, \Delta_k)$ and every function $\Gamma : \mathbb{F}_p^k \to \mathbb{F}_p$, there is a deterministic algorithm $\mathcal{A}_{k,\boldsymbol{\Delta},\Gamma}$ that takes as input a polynomial $P : \mathbb{F}_p^n \to \mathbb{F}_p$ of degree $d < p$, runs in time polynomial in n, and outputs a $(k, \boldsymbol{\Delta}, \Gamma)$-decomposition of P if one exists while otherwise returning* NO.

1.1 Discussion

The main result is surprisingly strong in that it holds for every k, $\boldsymbol{\Delta}$ and Γ. Thus, for instance, it immediately implies a (deterministic) poly(n)-time algorithm for factoring an n-variate polynomial of degree d over $\mathbb{F}_p$, as long as $p > d$ and p and d are fixed. Also, we observe (see the full version [Bha14]) that the proof of Theorem 1 implies a polynomial time algorithm for deciding whether a d-dimensional tensor over $\mathbb{F}_p$ has rank at most r, where d, p and r are constants and $d < p$.

We must remark that these results on factoring and tensor rank are not new, in the sense that there were already algorithms known for stronger versions of these two problems. Specifically, for deciding constant tensor rank, Karnin and Shpilka [KS09] showed a polynomial time algorithm for the more general problem of reconstructing multilinear $\Sigma\Pi\Sigma$ circuits with a constant number of multiplication gates. And for factoring multivariate polynomials over finite fields, it is known [vzGK85] how to factor in time $\mathrm{poly}(n, d, p)$ deterministically and in time $\mathrm{poly}(n, d, \log p)$ probabilistically.

However, Theorem 1 gives polynomial time algorithms for a whole host of problems not known to have non-trivial solutions previously, such as whether a polynomial of degree d can be expressed as $P_1 \cdot P_2 + P_3 \cdot P_4$ where each P_1, P_2, P_3, P_4 are of degree $d - 1$ or less. Thus, these problems become useful targets for reductions in future. Our main result can be described as a *black-box reconstruction algorithm* as in [KS09], in the sense that the algorithm is given blackbox query access to the polynomial and it runs in time linear in the dense representation of the polynomial (i.e., input size is measured as $\binom{n+d}{d}$). The property of having $(k, \boldsymbol{\Delta}, \Gamma)$-structure is also similar in spirit to a function having a *concise representation*, a notion introduced by Diakonikolas et al. in [DLM$^+$07]. We leave open as to whether there are formal connections here.

There are two main questions raised by Theorem 1:

1. Does Theorem 1 hold when $p \leqslant d$? The main difficulty here seems technical and stems from the fact that the proof of the Gowers inverse theorem for polynomials is currently very non-constructive [TZ12] when $p \leqslant d$, in contrast to the case of high characteristic [GT09].
2. Is there an analogous theorem when n is fixed and d and p are growing? Such questions are probably very difficult, because over $\mathbb{Z}_n$, we do not even know how to deterministically factorize the univariate polynomial $x^2 - a$, for a given $a \in \mathbb{Z}_n$. In fact, in recent work, Kopparty, Saraf and Shpilka [KSS14] have shown an equivalence between deterministic factorization of multivariate polynomials and derandomization of polynomial identity testing, a long-standing challenge. Polynomial time *randomized* algorithms exist for factorization of course but are not known to exist for arbitrary degree-structural properties over large fields. In particular, Neeraj Kayal (personal communication) asks whether it is possible in randomized polynomial time to decompose a univariate polynomial $P : \mathbb{F}_p \to \mathbb{F}_p$ of degree $n < p$ as $P = P_1 \cdot P_2 + P_3 \cdot P_4$ where P_1, P_2, P_3, P_4 are of degree $< n$. Even an average-case algorithm would be interesting, meaning P is known to be formed out of random polynomials P_1, P_2, P_3, P_4, and the task is to recover them given access to P.

1.2 Proof Overview

The proof of Theorem 1 is actually a straightforward combination of ideas from [BFH$^+$13] and [BHT13]. In [BFH$^+$13], it was shown that any degree-structural property is constant query *testable*. That is, for all $k, \boldsymbol{\Delta}$, and Γ, one can decide correctly, with probability at least 2/3, whether a given function is $(k, \boldsymbol{\Delta}, \Gamma)$-structured or whether it is 1%-far from any $(k, \boldsymbol{\Delta}, \Gamma)$-structured function, by querying the input function's value on only a constant number of points. The main contribution of [BFH$^+$13] is a reduction from the testability problem to the following combinatorial problem:

> Does there exist $s = s(k, \boldsymbol{\Delta}, \Gamma)$ such that a function is $(k, \boldsymbol{\Delta}, \Gamma)$-structured if and only if so is the restriction of the function to all affine subspaces of dimension s?

[BFH$^+$13] gave a positive answer to this problem (thus showing, by virtue of their main reduction, that degree-structure is testable). One can view their answer as a solution to the search problem of finding an s-dimensional subspace on which the function is not degree-structural. However, their proof is non-constructive, in the sense that no non-trivial algorithm is provided for finding the witnessing s-dimensional subspace.

At a high level, the reason that a violation to a degree-structural decomposition can be witnessed by a finite sized subspace is the following. Let the input polynomial be P on n variables and of degree d. Higher-order Fourier analysis gives a way to write P as:

$$P(x) = G(Q_1(x), Q_2(x), \ldots, Q_C(x))$$

where C is a constant, $Q_1, \ldots, Q_C$ polynomials of degree $\leqslant d$ and G is an arbitrary function on $\mathbb{F}^C$. Most importantly, $Q_1, \ldots, Q_C$ have a certain pseudorandomness property called *high rank*, which allows us to think of $Q_1(x), \ldots, Q_C(x)$ as C uncorrelated variables[2]. Thus, higher-order Fourier analysis *finitizes* P on n variables into a function G on only a constant number of variables. Moreover, G remains the same when P is restricted to a function on $n-1$ variables by setting one of the variables to zero. Thus, we can keep on setting variables to zero until we have only a constant number of variables remaining. That is, P is now restricted to a finite sized subspace H, with:

$$P_H(x) = G(Q_{1|H}(x), \ldots, Q_{C|H}(x))$$

where $Q_{1|H}, \ldots, Q_{C|H}$ still enjoy the pseudorandom property of high rank. Now, P_H can be decomposed by brute force, and moreover, it can be decomposed in terms of $Q_{1|H}, \ldots, Q_{C|H}$ (and perhaps other polynomials) due to their high rank. Finally, at this point, $Q_1, \ldots Q_C$ can be directly substituted instead of $Q_{1|H}, \ldots, Q_{C|H}$ into the decomposition, and so the decomposition of the original

[2] More precisely, the distribution of $(Q_1(X), \ldots, Q_C(X))$ is close to uniform for uniform $X \in F^n$.

polynomial P is recovered. The fact that the last substitution doesn't increase the degree is again due to the high rank of $Q_1, \dots, Q_C$.

In this argument, the rank of a polynomial plays a central role in the analysis, but we do not know how to compute this quantity in time polynomial in n. Hence, the argument in [BFH$^+$13] is non-constructive in this aspect. However, in [BHT13], it was noticed that when the polynomial degree is smaller than the field characteristic, instead of the rank of a polynomial, one could equally well work with the *Gowers uniformity norm* (see Section 2.1) of the polynomial, and the Gowers norm can be estimated upto constant additive error with good probability by evaluating the polynomial on a constant number of random samples. Via this approach, [BHT13] found an algorithmic *regularity lemma* for degree-d n-variate polynomials (see Section 2.2) that runs in time $O(n^d)$ when $d < p$.

In spirit, our algorithm is very similar to Kaltofen's factorization algorithm [Kal95], where the polynomial is first restricted to a random two-dimensional subspace, then factored using bivariate factorization algorithms, and then lifted back to the original space. From this perspective, we show that the "restrict-solve-lift" paradigm can be used for any degree-structural decomposition problem, not just factorization (at least when the field order is a constant prime but larger than the degree of the input polynomial). We hope that this work brings the techniques of higher-order Fourier analysis to the attention of a wider audience in computer science.

2 Technical Preliminaries

From a bird's eye viewpoint, higher-order Fourier analysis is a study of how the analytic properties of a collection of polynomial relate to the collection's algebraic/combinatorial structure. We make precise all the needed notions in the subsections below.

To start off, let us define the important notion of a polynomial factor:

Definition 2. *If $P_1, \dots, P_C : \mathbb{F}_p^n \to \mathbb{F}$ is a sequence of polynomials, then the tuple $\mathcal{B} = (P_1, \dots, P_C)$ is called a* polynomial factor*. The* complexity *of $\mathcal{B}$, denoted $|\mathcal{B}|$, is the number of defining polynomials, C. The* degree *of $\mathcal{B}$ is the maximum degree among its defining polynomials $P_1, \dots, P_C$. Also, $\|\mathcal{B}\| = p^C$ is called the* order *of $\mathcal{B}$; the number of nonempty atoms of $\mathcal{B}$ is bounded by $\|\mathcal{B}\|$. By an abuse of notation, we also use $\mathcal{B}$ to denote the map $x \mapsto (P_1(x), \dots, P_C(x))$.*

2.1 Three Notions of Polynomial Pseudorandomness

The main results of higher-order Fourier analysis revolve around three measures of pseudorandomness for polynomial factors. Each is a statistical test that is perfectly met by truly random polynomial factors, and the question is how well are they met by factors of degree d.

Bias. The first pseudorandomness measure is the familiar notion of bias, generalizing the definition of Naor and Naor [NN93] over $\mathbb{F}_2$.

Definition 3 (Unbiased). *The* bias *of a function $F : \mathbb{F}_p^n \to \mathbb{F}_p$ is:*

$$\mathsf{bias}(F) = \left| \mathop{\mathbf{E}}_{x \in \mathbb{F}_p^n} [\mathsf{e}(F(x))] \right|$$

*Given a function $\beta : \mathbb{Z}^+ \to (0,1)$ and a polynomial factor $\mathcal{B}$ defined by a sequence of polynomials $P_1, \ldots, P_C : \mathbb{F}_p^n \to \mathbb{F}$, the factor $\mathcal{B}$ is said to be β-*unbiased *if for every $(a_1, \ldots, a_C) \in \{0, \ldots, p-1\}^C \setminus \{0^C\}$,*

$$\mathsf{bias}\left(\sum_{i=1}^{C} a_i P_i\right) < \beta(C).$$

The following facts are straightforward and folklore.

Lemma 1 (Equidistribution). *Given $\beta : \mathbb{Z}^+ \to (0,1)$, let $\mathcal{B}$ be a β-unbiased polynomial factor of complexity C. For any $b \in \mathbb{F}^C$:*

$$\Pr_x[\mathcal{B}(x) = b] = \frac{1}{\|\mathcal{B}\|} \pm \beta(C).$$

Corollary 1 (Atom Dispersal). *If $\beta(k) = \frac{1}{2p^k}$ and $\mathcal{B}$ is a β-unbiased polynomial factor, then all of the $\|\mathcal{B}\|$ atoms of $\mathcal{B}$ are nonempty.*

The following theorem[3], proved in [BFH$^+$13] shows that a function of an unbiased factor of degree d has the degree which one would expect from a generic collection of polynomials of degree d.

Theorem 2 (Degree Preservation, Theorem 4.1 of [BFH$^+$13]). *For any positive integer $d < p$, there is a function $\alpha_2^d : \mathbb{Z}^+ \to (0,1)$ such that the following is true. Let $\mathcal{B}$ be any factor defined by polynomials $P_1, \ldots, P_C : \mathbb{F}_p^n \to \mathbb{F}_p$ of degree $\leqslant d$. Suppose $\mathcal{B}$ is α_2^d-unbiased. Let $\Gamma : \mathbb{F}_p^C \to \mathbb{F}_p$ be an arbitrary function. Define the polynomial $F : \mathbb{F}_p^n \to \mathbb{F}_p$ by $F(x) = \Gamma(\mathcal{B}(x))$.*

Then, for any factor $\mathcal{B}'$ defined by polynomials $Q_1, \ldots, Q_C : \mathbb{F}_p^n \to \mathbb{F}_p$ with $\deg(Q_i) \leqslant \deg(P_i)$ *for every $i \in [C]$, if $G : \mathbb{F}_p^n \to \mathbb{F}_p$ is the polynomial $G(x) = \Gamma(\mathcal{B}'(x))$, it holds that* $\deg(G) \leqslant \deg(F)$.

Uniformity. Bias is often a very weak measure of pseudorandomness: the bias of any linear function is 0, even though it is clearly not a random function. We could strengthen low bias by additionally requiring that all the Fourier coefficients be small, which would ensure that the function is not (correlated with) a linear function. Continuing down this path leads us to the notion of uniformity, which measures the correlation of a function with polynomials of bounded degree.

[3] A variant of Theorem 2 is true when $p \leqslant d$ also, as shown in [BFH$^+$13], but in that case, they require the stronger assumption of uniformity (see next section) instead of unbiasedness.

Definition 4 (Multiplicative Derivative). *Given a function $f : \mathbb{F}_p^n \to \mathbb{C}$ and an element $h \in \mathbb{F}_p^n$, the* multiplicative derivative of f in direction h *is the function $\Delta_h f : \mathbb{F}_p^n \to \mathbb{C}$ satisfying $\Delta_h f(x) = f(x+h)\overline{f(x)}$ for all $x \in \mathbb{F}_p^n$.*

Definition 5 (Uniformity). *Given a function $f : \mathbb{F}_p^n \to \mathbb{C}$ and an integer $d \geqslant 1$, the* Gowers uniformity norm of order d *for f is given by:*

$$\|f\|_{U^d} = \left| \underset{h_1,\dots,h_d \in \mathbb{F}_p^n}{\mathbf{E}} \ \underset{x \in \mathbb{F}_p^n}{\mathbf{E}} \left[(\Delta_{h_1}\Delta_{h_2}\cdots\Delta_{h_d} f)(x)\right] \right|^{1/2^d}$$

*Given a function $\gamma : \mathbb{Z}^+ \to (0,1)$ and a polynomial factor $\mathcal{B}$ defined by a sequence of polynomials $P_1, \dots, P_C : \mathbb{F}_p^n \to \mathbb{F}$, the factor $\mathcal{B}$ is said to be γ-*uniform *if for every $(a_1, \dots, a_C) \in \{0, \dots, p-1\}^C \setminus \{0^C\}$,*

$$\left\| \mathsf{e}\left(\sum_{i=1}^C a_i P_i\right) \right\|_{U^d} < \gamma(C)$$

where $d = \max_i \deg(a_i P_i)$.

Note that $\mathsf{bias}(P) = \|\mathsf{e}(P)\|_{U^1}$ for any $P : \mathbb{F}_p^n \to \mathbb{F}$. Moreover, it holds that $\|f\|_{U^d} \leqslant \|f\|_{U^{d+1}}$ for any $f : \mathbb{F}_p^n \to \mathbb{C}$ and $d \geqslant 1$ [Gow98]. So:

Lemma 2 (Uniformity Implies Unbiased). *If $\mathcal{B}$ is a polynomial factor that is γ-uniform for some function $\gamma : \mathbb{Z}^+ \to (0,1)$, then $\mathcal{B}$ is also γ-unbiased.*

Regularity. A third measure of pseudorandomness was introduced by Green and Tao [GT09] as a bridge between the algebraic structure of polynomials and the analytic notions of bias and uniformity.

Definition 6 (Regularity). *Given a function $F : \mathbb{F}_p^n \to \mathbb{F}_p$ and an integer $d > 1$, the* d-rank of F, *denoted* $\mathsf{rank}_d(F)$, *is defined to be the smallest integer r such that there exist polynomials $Q_1, \dots, Q_r : \mathbb{F}_p^n \to \mathbb{F}_p$ of degree $\leqslant d-1$ and a function $\Gamma : \mathbb{F}_p^r \to \mathbb{F}_p$ satisfying $P(x) = \Gamma(Q_1(x), \dots, Q_r(x))$. If $d = 1$, the 1-rank is defined to be ∞ if F is non-constant and 0 otherwise.*

*Given a function $R : \mathbb{Z}^+ \to \mathbb{Z}^+$ and a polynomial factor $\mathcal{B}$ defined by a sequence of polynomials $P_1, \dots, P_C : \mathbb{F}_p^n \to \mathbb{F}_p$, the factor $\mathcal{B}$ is said to be R-*regular *if for every $a_1, \dots, a_C \in \{0, 1, \dots, p-1\}^C \setminus \{0^C\}$,*

$$\mathsf{rank}_d\left(\sum_{i=1}^C a_i P_i\right) > R(C)$$

where $d = \max_i \deg(a_i P_i)$. Also, the rank of $\mathcal{B}$ *is at least $R(C)$.*

Regularity and uniformity turn out to be essentially equivalent, due to the following two remarkable theorems. The first theorem is folklore and essentially due to (linear) Fourier analysis.

Theorem 3 (Uniformity Implies Regularity). *Suppose that $p > d$ and let $R : \mathbb{Z}^+ \to \mathbb{Z}^+$ be any non-decreasing function. Then, there is a function $\gamma_3^{d,R} : \mathbb{Z}^+ \to (0,1)$ such that the following holds. Any polynomial factor of degree d that is $\gamma_3^{d,R}$-uniform is also R-regular.*

Theorem 4 (Regularity Implies Uniformity, Proposition 6.1 of [GT09]). *Suppose that $p > d$, and let $\gamma : \mathbb{Z}^+ \to (0,1)$ be any non-increasing function. Then, there is a function $R_4^{d,\gamma} : \mathbb{Z}^+ \to \mathbb{Z}^+$ such that the following holds. Any polynomial factor of degree d that is $R_4^{d,\gamma}$-regular is also γ-uniform.*

Remark 1. Importantly, when $d < p$, $\gamma_3^{d,R}$ is explicitly known, given d and R. In other words, given access to an evaluation oracle for R, $\gamma_3^{d,R}$ is polynomial-time computable. Similarly, $R_4^{d,\gamma}$ is explicitly known.

While unbiasedness and uniformity are analytic properties of a factor, regularity is an algebraic notion and is hence more amenable to algebraic operations on the function. For instance, we have:

Lemma 3 (Subspace Restriction, Lemma 2.13 of [BFH$^+$13]). *Suppose $P : \mathbb{F}_p^n \to \mathbb{F}_p$ is a polynomial of degree d and rank r, where $r > p + 1$. Let A be a hyperplane in $\mathbb{F}_p^n$, and denote by P' the restriction of P to A. Then, P' is a polynomial of degree d and rank $\geqslant r - p$, unless $d = 1$ and P is constant on A.*

2.2 Algorithmic Regularity Lemma

The celebrated Szemerédi graph regularity lemma [Sze78] permits the decomposition of an arbitrary graph into bipartite subgraphs which are regular (in the graph-theoretic sense). One can carry out an analogous type of refinement for our notions of regularity also. First, let us specify what we mean by refinements of a factor.

Definition 7 (Semantic and Syntactic Refinements). *$\mathcal{B}'$ is called a* semantic refinement *(or simply, a* refinement*) if the partition induced by $\mathcal{B}'$ is a combinatorial refinement of the partition induced by $\mathcal{B}$. In other words, if for every $x, y \in \mathbb{F}_p^n$, $\mathcal{B}'(x) = \mathcal{B}'(y)$ implies $\mathcal{B}(x) = \mathcal{B}(y)$. $\mathcal{B}'$ is called a* syntactic refinement *of $\mathcal{B}$ if the sequence of polynomials defining $\mathcal{B}'$ extends that of $\mathcal{B}$. A syntactic refinement is clearly a semantic refinement but not necessarily, vice versa.*

The algorithmic regularity lemma of [BHT13] (analogous to the algorithmic version [ADL$^+$94] of Szemerédi's regularity lemma) is as follows:

Theorem 5 (Uniform Refinement, Lemma 4.1 of [BHT13]). *Suppose $d < p$ is a positive integer, $\rho \in (0,1)$, and $\gamma : \mathbb{Z}^+ \to (0,1)$ is a non-increasing function. There is a function $C_5^{\gamma,d} : \mathbb{Z}^+ \to \mathbb{Z}^+$ and an algorithm that takes as input a factor $\mathcal{B}$ of $\mathbb{F}_p^n$ of degree d, runs in time $O(n^d)$ and with probability $1 - \rho$, outputs a γ-uniform factor $\tilde{\mathcal{B}}$ where $\tilde{\mathcal{B}}$ is a refinement of $\mathcal{B}$, is of degree d, and $|\tilde{\mathcal{B}}| \leqslant C_5^{\gamma,d}(|\mathcal{B}|)$.*

Combining with Theorem 3 immediately implies:

Corollary 2 (Regular Refinement). *Suppose $d < p$ is a positive integer, $\rho \in (0,1)$ and $R : \mathbb{Z}^+ \to \mathbb{Z}^+$ is a non-decreasing function. There is a function $C_2^{R,d} : Z^+ \to \mathbb{Z}^+$ and an algorithm that takes as input a factor $\mathcal{B}$ of $\mathbb{F}_p^n$ of degree d, runs in time $O(n^d)$ and with probability $1-\rho$, outputs a R-regular factor $\tilde{\mathcal{B}}$ where $\tilde{\mathcal{B}}$ is a refinement of $\mathcal{B}$, is of degree d, and $|\tilde{\mathcal{B}}| \leqslant C_2^{R,d}(|\mathcal{B}|)$. Additionally, if $\mathcal{B}$ is defined by polynomials $P_1, P_2, \ldots, P_m$, then we can find functions $\Gamma_1, \ldots, \Gamma_m : \mathbb{F}_p^{|\tilde{\mathcal{B}}|} \to \mathbb{F}_p$ such that $P_i(x) = \Gamma_i(\tilde{\mathcal{B}}(x))$ for every $i \in [m]$.*

Moreover, if $\mathcal{B}$ is itself a syntactic refinement of some $\mathcal{B}'$ that is of rank at least $R(|\mathcal{B}|)+1$, then $\tilde{\mathcal{B}}$ will also be a syntactic refinement of $\mathcal{B}'$.

The second-to-last sentence of Corollary 2 comes from observing that the proof of Lemma 4.1 in [BHT13] explicitly constructs the functions Γ_i. The last sentence of Corollary 2 follows from Lemma 3.17 of [BFL13].

3 The Main Proof

First, we prove Theorem 1 allowing the algorithm to be randomized.

Theorem 6. *If $p > d$, then for any fixed $k, \boldsymbol{\Delta}$ and Γ, there is a randomized algorithm which given a polynomial $P : \mathbb{F}_p^n \to \mathbb{F}_p$ of degree d runs in time $O(n^{d+1})$ and has the following behavior:*

1. *If P is $(k, \boldsymbol{\Delta}, \Gamma)$-structured, with probability $2/3$, it finds a $(k, \boldsymbol{\Delta}, \Gamma)$-decomposition of P.*
2. *Otherwise, it always outputs* NO.

Proof. Let $R : \mathbb{Z}^+ \to \mathbb{Z}^+$ be defined as $R(m) = r(C_2^{r,d}(m+k)) + C_2^{r,d}(m+k) + p$ for a function $r : \mathbb{Z}^+ \to \mathbb{Z}^+$ to be fixed later. First, we apply Corollary corollary 2 to the factor defined by $\{P\}$ so that with probability 9/10, we find an R-regular polynomial factor $\mathcal{B}$ of degree d defined by polynomials $P_1, P_2, \ldots, P_C : \mathbb{F}_p^n \to \mathbb{F}_p$ such that $P(x) = G(\mathcal{B}(x))$ for some $G : \mathbb{F}_p^C \to \mathbb{F}_p$. Here, $C \leqslant C^{R,d}(1) = O(1)$.

If $n \leqslant Cd$, then we can decide whether f is $(k, \boldsymbol{\Delta}, \Gamma)$-structured by brute force in $O(1)$ time.

Otherwise, we are in the case $n > Cd$. From each P_i, pick a monomial m_i with degree equal to $\deg(P_i)$. Since $n > Cd$, there exists $i_0 \in [n]$ such that x_{i_0} does not appear in any of the m_i's. Let $P_1', P_2', \ldots, P_C'$ be $P_1|_{x_{i_0}=0}, P_2|_{x_{i_0}=0}, \ldots, P_C|_{x_{i_0}=0}$ respectively, and let $\mathcal{B}'$ be the factor defined by these polynomials. Clearly, $\deg(P_i') = \deg(P_i)$ for each $i \in [C]$. Moreover, by Subspace Restriction Lemma 3, $\mathcal{B}'$ is $(R-p)$-regular.

Recursively, decide $(k, \boldsymbol{\Delta}, \Gamma)$-structure for the polynomial $P' \overset{\text{def}}{=} P|_{x_{i_0}=0}$ on $n-1$ variables. Note that:

$$P'(x) = G(P_1'(x), P_2'(x), \ldots, P_C'(x)).$$

If P' is not $(k, \boldsymbol{\Delta}, \Gamma)$-structured, then clearly P cannot be, and the algorithm can output NO. Otherwise, suppose that:

$$P'(x) = \Gamma(S_1(x), S_2(x), \ldots, S_k(x))$$

where $\deg(S_1), \ldots, \deg(S_k)$ are at most $\Delta_1, \ldots, \Delta_k$ respectively. We need to show how to extract $(k, \boldsymbol{\Delta}, \Gamma)$-structure for P from this decomposition for P'.

Use Corollary 2 to find, with probability at least 9/10, an r-regular refinement $\mathcal{B}'$ of the factor defined by $\{P'_1, \ldots, P'_C, S_1, \ldots, S_k\}$. Note that the rank of $\mathcal{B}'$ is at least $r(|\mathcal{B}'|)$, while the rank of the factor defined by $\{P'_1, \ldots, P'_C\}$ is at least $R(C) - p = r(C_2^{r,d}(C + k)) + C_2^{r,d}(C + k) \geqslant r(|\mathcal{B}'|) + |\mathcal{B}'|$. Because of the last part of Corollary 2, $\mathcal{B}'$ is a syntactic refinement of of $\{P'_1, \ldots, P'_C\}$. That is, we obtain a polynomial factor $\mathcal{B}' = \{P'_1, \ldots, P'_C, S'_1, \ldots, S'_D\}$ which has degree d and rank $> r(C + D)$, where $C + D \leqslant C_2^{r,d}(C + k)$ and where $S_i(x) = G_i(P'_1(x), \ldots, P'_C(x), S'_1(x), \ldots, S'_D(x))$ for some function $G_i : \mathbb{F}_p^{C+D} \to \mathbb{F}_p$. Thus, we have that for all x:

$$\begin{aligned} &G(P'_1(x), \ldots, P'_C(x)) \\ &= \Gamma(G_1(P'_1(x), \ldots, P'_C(x), S'_1(x), \ldots, S'_D(x)), \ldots, G_k(P'_1(x), \ldots, P'_C(x), S'_1(x), \ldots, S'_D(x)) \end{aligned}$$

Note that by Corollary 2, we find the functions $G_1, \ldots, G_k$ explicitly.

Let $\gamma(m) = \frac{1}{2p^m}$, and suppose $r(m) > R_4^{d,\gamma}(m)$. Then, by Theorem 4, Lemma 2 and Corollary 1, we see that $\mathcal{B}'(x)$ acquires every possible value in its range. Thus, we have the identity:

$$G(a_1, \ldots, a_C) = \Gamma(G_1(a_1, \ldots, a_C, b_1, \ldots, b_D), \ldots, G_k(a_1, \ldots, a_C, b_1, \ldots, b_D))$$

for *all* $a_1, \ldots, a_C, b_1, \ldots, b_D \in \mathbb{F}_p$. In particular:

$$\begin{aligned} P(x) = G(P_1(x), \ldots, &P_C(x)) \\ &= \Gamma(G_1(P_1(x), \ldots, P_C(x), 0, \ldots, 0), \ldots, G_k(P_1(x), \ldots, P_C(x), 0, \ldots, 0)) \end{aligned}$$

Define $Q_i(x) = G_i(P_1(x), \ldots, P_C(x), 0, \ldots, 0)$ for each $i \in [k]$. Now, suppose $r(m) > R_4^{d,\alpha_2^d}(m)$. By Lemma 2 and Theorem 2, since $\deg(P_i) = \deg(P'_i)$, it follows that $\deg(Q_i) \leqslant \deg(S_i) \leqslant \Delta_i$ for each $i \in [k]$. Then, our $(k, \boldsymbol{\Delta}, \Gamma)$-decomposition is given by:

$$P(x) = \Gamma(Q_1(x), \ldots, Q_k(x))$$

Hence, set $r = \max(R_4^{d,\gamma}, R_4^{d,\alpha_2^d})$.

In order to see the guarantees in the theorem statement, consider repeating the above algorithm infinitely until a $(k, \boldsymbol{\Delta}, \Gamma)$-decomposition is discovered for P. If P is not $(k, \boldsymbol{\Delta}, \Gamma)$-structured, then any candidate $(k, \boldsymbol{\Delta}, \Gamma)$-decomposition discovered (due to the error probability in Corollary 2) can be ruled out in $O(n^d)$ time. Otherwise, if P is $(k, \boldsymbol{\Delta}, \Gamma)$-structured the expected time before a valid $(k, \boldsymbol{\Delta}, \Gamma)$-decomposition is discovered will be the expected time for discovering a decomposition for P' plus expected $O(n^d)$ time for finding valid regular refinements. Thus, the expected time to find a $(k, \boldsymbol{\Delta}, \Gamma)$-decomposition for P is $O(n^{d+1})$. Therefore, if we stop repeating the algorithm after $O(n^{d+1})$ time steps, our desired result is true by Markov's theorem.

Theorem 6 can be derandomized using existing pseudorandom generators for low-degree polynomials [Vio09] to yield Theorem 1. This idea was suggested by Shachar Lovett. Due to space constraints, we omit the proof here and refer the reader to the full version [Bha14].

Acknowledgments. Part of this work was done when visiting the Simons Institute and MIT during December, 2013. Many thanks to Ronitt Rubinfeld for useful discussions, to Pablo Parrilo for asking about tensor rank and to Neeraj Kayal, Amir Shpilka and Madhu Sudan for pointers to previous work.

References

ADL$^+$94. Alon, N., Duke, R.A., Lefmann, H., Rödl, V., Yuster, R.: The algorithmic aspects of the regularity lemma. J. Algorithms 16(1), 80–109 (1994)

BCSX11. Bhattacharyya, A., Chen, V., Sudan, M., Xie, N.: Testing linear-invariant non-linear properties. Theory Comput 7(1), 75–99 (2011)

BFH$^+$13. Bhattacharyya, A., Fischer, E., Hatami, H., Hatami, P., Lovett, S.: Every locally characterized affine-invariant property is testable. In: Proc. 45th Annual ACM Symposium on the Theory of Computing, pp. 429–436 (2013)

BFL13. Bhattacharyya, A., Fischer, E., Lovett, S.: Testing low complexity affine-invariant properties. In: Proc. 24th ACM-SIAM Symposium on Discrete Algorithms, pp. 1337–1355 (2013), http://arxiv.org/abs/1201.0330v2

BGS10. Bhattacharyya, A., Grigorescu, E., Shapira, A.: A unified framework for testing linear-invariant properties. In: Proc. 51st Annual IEEE Symposium on Foundations of Computer Science, pp. 478–487 (2010)

Bha14. Bhattacharyya, A.: Polynomial decompositions in polynomial time. Technical report (February 2014), http://eccc.hpi-web.de/report/2014/018/

BHT13. Bhattacharyya, A., Hatami, P., Tulsiani, M.: Algorithmic regularity for polynomials and applications. Technical report (November 2013), http://arxiv.org/abs/1311.5090

DLM$^+$07. Diakonikolas, I., Lee, H.K., Matulef, K., Onak, K., Rubinfeld, R., Servedio, R.A., Wan, A.: Testing for concise representations. In: Proc. 48th Annual IEEE Symposium on Foundations of Computer Science, pp. 549–558 (2007)

dW08. de Wolf, R.: A Brief Introduction to Fourier Analysis on the Boolean Cube. Graduate Surveys, vol. 1. Theory of Computing Library (2008)

Gow98. Gowers, W.T.: A new proof of Szeméredi's theorem for arithmetic progressions of length four. Geom. Funct. Anal. 8(3), 529–551 (1998)

Gow01. Gowers, W.T.: A new proof of Szeméredi's theorem. Geom. Funct. Anal. 11(3), 465–588 (2001)

GT08. Green, B., Tao, T.: An inverse theorem for the Gowers U^3-norm. Proc. Edin. Math. Soc. 51, 73–153 (2008)

GT09. Green, B., Tao, T.: The distribution of polynomials over finite fields, with applications to the Gowers norms. Contrib. Discrete Math. 4(2) (2009)

GT10. Green, B., Tao, T.: Linear equations in primes. Ann. of Math. 171, 1753–1850 (2010)

GTZ. Green, B., Tao, T., Ziegler, T.: An inverse theorem for the Gowers U^{s+1}-norm. In: Ann. of Math. (to appear)

GTZ11. Green, B., Tao, T., Ziegler, T.: An inverse theorem for the Gowers U^4-norm. Glasgow Math. J. 53(1), 1–50 (2011)

HK05. Host, B., Kra, B.: Nonconventional ergodic averages and nilmanifolds. Ann. of Math. 161(1), 397–488 (2005)

Kal95. Kaltofen, E.: Effective Noether irreducibility forms and applications. J. Comp. Sys. Sci. 50(2), 274–295 (1995)

KL08. Kaufman, T., Lovett, S.: Worst case to average case reductions for polynomials. In: Proc. 49th Annual IEEE Symposium on Foundations of Computer Science, pp. 166–175 (2008)

KS09. Karnin, Z.S., Shpilka, A.: Reconstruction of generalized depth-3 arithmetic circuits with bounded top fan-in. In: Proc. 24th Annual IEEE Conference on Computational Complexity, pp. 274–285 (2009)

KSS14. Kopparty, S., Saraf, S., Shpilka, A.: Equivalence of polynomial identity testing and deterministic multivariate polynomial factorization. Technical Report 001, Electronic Colloquium on Computational Complexity (January 2014), http://eccc.hpi-web.de/report/2014/001/

NN93. Naor, J., Naor, M.: Small-bias probability spaces: efficient constructions and applications. SIAM J. on Comput. (4), 838–856 (1993), Earlier version in STOC 1990

Sze78. Szemerédi, E.: Regular partitions of graphs. In: Bremond, J.C., Fournier, J.C., Las Vergnas, M., Sotteau, D. (eds.) Proc. Colloque Internationaux CNRS 260 – Problèmes Combinatoires et Théorie des Graphes, pp. 399–401 (1978)

Tao12. Tao, T.: Higher Order Fourier Analysis. Graduate Studies in Mathematics, vol. 142. American Mathematical Society (2012)

TZ10. Tao, T., Ziegler, T.: The inverse conjecture for the Gowers norm over finite fields via the correspondence principle. Analysis & PDE 3(1), 1–20 (2010)

TZ12. Tao, T., Ziegler, T.: The inverse conjecture for the Gowers norm over finite fields in low characteristic. Ann. Comb. 16(1), 121–188 (2012)

Vio09. Viola, E.: The sum of D small-bias generators fools polynomials of degree D. Computational Complexity 18(2), 209–217 (2009)

VW08. Viola, E., Wigderson, A.: Norms, XOR lemmas, and lower bounds for polynomials and protocols. Theory Comput 4(7), 137–168 (2008)

vzGK85. von zur Gathen, J., Kaltofen, E.: Factorization of multivariate polynomials over finite fields. Mathematics of Computation 45(171), 251–261 (1985)

Fault-Tolerant Approximate Shortest-Path Trees*

Davide Bilò[1], Luciano Gualà[2], Stefano Leucci[3], and Guido Proietti[3,4]

[1] Dipartimento di Scienze Umanistiche e Sociali, Università di Sassari, Italy
[2] Dipartimento di Ingegneria dell'Impresa, Università di Roma "Tor Vergata", Italy
[3] Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica, Università degli Studi dell'Aquila, Italy
[4] Istituto di Analisi dei Sistemi ed Informatica, CNR, Roma, Italy
davide.bilo@uniss.it, guala@mat.uniroma2.it, {stefano.leucci,guido.proietti}@univaq.it

Abstract. The resiliency of a network is its ability to remain *effectively* functioning also when any of its nodes or links fails. However, to reduce operational and set-up costs, a network should be small in size, and this conflicts with the requirement of being resilient. In this paper we address this trade-off for the prominent case of the *broadcasting* routing scheme, and we build efficient (i.e., sparse and fast) *fault-tolerant approximate shortest-path trees*, for both the edge and vertex *single-failure* case. In particular, for an n-vertex non-negatively weighted graph, and for any constant $\varepsilon > 0$, we design two structures of size $O(\frac{n \log n}{\varepsilon^2})$ which guarantee $(1+\varepsilon)$-stretched paths from the selected source also in the presence of an edge/vertex failure. This favorably compares with the currently best known solutions, which are for the edge-failure case of size $O(n)$ and stretch factor 3, and for the vertex-failure case of size $O(n \log n)$ and stretch factor 3. Moreover, we also focus on the unweighted case, and we prove that an ordinary (α, β)-spanner can be slightly augmented in order to build efficient fault-tolerant approximate *breadth-first-search trees*.

1 Introduction

Broadcasting a message from a source node to every other node of a network is one of the most basic communication primitives. Since this operation should be performed by making use of a both sparse and fast infrastructure, the natural solution is to root at the source node a *shortest-path tree* (SPT) of the underlying graph. However, the SPT, as any tree-based network topology, is highly sensitive to a link/node malfunctioning, which will unavoidably cause the disconnection of a subset of nodes from the source.

To be readily prepared to react to any possible (transient) failure in a SPT, one has then to enrich the tree by adding to it a set of edges selected from the underlying graph, so that the resulting structure will be 2-edge/vertex-connected

* This work was partially supported by the Research Grant PRIN 2010 "ARS TechnoMedia", funded by the Italian Ministry of Education, University, and Research.

w.r.t. the source. Thus, after an edge/vertex failure, these edges will be used to build up the alternative paths emanating from the root, each one of them in replacement of a corresponding original shortest path which was affected by the failure. However, if these paths are constrained to be *shortest*, then it can be easily seen that for a non-negatively real weighted and undirected graph G of n nodes and m edges, this may require as much as $\Theta(m)$ additional edges, also in the case in which $m = \Theta(n^2)$. In other words, the set-up costs of the strengthened network may become unaffordable. Thus, a reasonable compromise is that of building a *sparse* and *fault-tolerant* structure which *accurately approximates* the shortest paths from the source, i.e., that contains paths which are longer than the corresponding shortest paths by at most a multiplicative *stretch* factor, for any possible edge/vertex failure. The aim of this paper is to show that very efficient structures of this sort do actually exist.

Related work. Let s denote a distinguished source vertex of a non-negatively real weighted and undirected graph $G = (V(G), E(G))$. We say that a spanning subgraph H of G is an *Edge-fault-tolerant α-Approximate SPT* (in short, α-EASPT), with $\alpha > 1$, if it satisfies the following condition: For each edge $e \in E(G)$, all the distances from s in the subgraph $H - e = (V(H), E(H) \setminus \{e\})$ are α-stretched w.r.t. the corresponding distances in $G - e$. When *vertex failures* are considered, then the EASPT is correspondingly called VASPT.

Our work is inspired by the paper of Parter and Peleg [13], which were concerned with the same problem but on *unweighted* graphs (and so they were focusing on the construction of an *edge-fault-tolerant α-approximate Breadth-First Search tree* (in short, α-EABFS). In that paper the authors present a 3-EABFS having at most $4n$ edges.[1] Moreover, the authors also present a set of lower and upper bounds to the size of (α, β)-EABFS, i.e., edge-fault-tolerant structures for which the length of a path is stretched by at most a factor of α plus an additive term of β. Finally, assuming at most $f = O(1)$ edge failures can take place, they show the existence of a $(3(f+1), (f+1)\log n)$-EABFS of size $O(fn)$.

On the other hand, if one wants to have an *exact* edge-fault-tolerant SPT (say ESPT), then as we said before this may require $\Theta(n^2)$ edges. This is now in contrast with the unweighted case, where it can be shown the existence (see [12]) of an *edge/vertex-fault-tolerant BFS* (say EBFS/VBFS) of size $O(n \cdot \min\{ecc(s), \sqrt{n}\})$, where $ecc(s)$ denotes the eccentricity of s in G. In the same paper, the authors also exhibit a corresponding lower bound of $\Omega(n^{3/2})$ for the size of a EBFS. Moreover, they also treat the *multisource* case, i.e., that in which we look for a structure which incorporates an EBFS rooted at each vertex of a set $S \subseteq V(G)$. For this, they show the existence of a solution of size $O(\sqrt{|S|} \cdot n^{3/2})$, which is tight. Finally, the authors provide an $O(\log n)$-approximation algorithm for constructing an optimal (in terms of size) EBFS (also for the multisource case), and they show this is tight.

[1] Notice that this result is obtained through a rather involved algorithm that suitably enriches a BFS of G rooted at the source node, but, as we will point out in more detail later, a 3-EASPT of size at most $2n$ (and then, *a fortiori*, a 3-EABFS of the same size), can actually be obtained as a by-product of the results given in [11].

As far as the vertex-failure problem is concerned, in [3] the authors study the related problem of computing *distance sensitivity oracles* (DSO) structures. Designing an efficient DSO means to compute, with a *low* preprocessing time, a *compact* data structure which is functional to *quickly* answer to some distance query following a component failure. Classically, DSO cope with single edge/vertex failures, and they have to answer to a point-to-point post-failure (approximate) distance query, or they have to report a point-to-point replacement short(est) path. In particular, in [3] the vertex-failure case w.r.t. a SPT is analyzed, and the authors compute in $O(m \log n + n^2 \log n)$ time a DSO of size $O(n \log n)$, that returns a 3-stretched replacement path in time proportional to the path's size. As the authors specify in the paper, this DSO can be used to build a 3-VASPT of size $O(n \log n)$, and a $(1+\varepsilon)$-VABFS of size $O(\frac{n}{\varepsilon^3} + n \log n)$. Actually, we point out that the latter structure can be easily sparsified so as to obtain a $(1+\varepsilon)$-EABFS of size $O(\frac{n}{\varepsilon^3})$: in fact, its $O(n \log n)$ size term is associated with an auxiliary substructure that, in the case of edge failures, can be made of linear size. This result is of independent interest, since it qualifies itself as the best current solution for the EABFS problem.

Our results. Our main result is the construction in polynomial time[2] of a $(1+\varepsilon)$-VASPT of size $O(\frac{n \log n}{\varepsilon^2})$, for any $\varepsilon > 0$. This substantially improves on the 3-VASPT of size $O(n \log n)$ given in [3]. To obtain our result, we perform a careful selection of edges that will be added to an initial SPT. The somewhat surprising outcome of our approach is that if we accept to have slightly stretched fault-tolerant paths, then we can drastically reduce the $\Theta(n^2)$ size of the structure that we would have to pay for having fault-tolerant *shortest* paths! Actually, the analysis of the stretch factor and of the structure's size induced by our algorithm is quite involved. Thus, for clarity of presentation, we give our result in two steps: first, we show an approach to build a $(1+\varepsilon)$-EASPT of size $O(\frac{n \log n}{\varepsilon^2})$, then we outline how this approach can be extended to the vertex-failure case.

Furthermore, we also focus on the unweighted case, and we exhibit an interesting connection between a fault-tolerant BFS and an (α, β)*-spanner*. An (α, β)-spanner of a graph G is a spanning subgraph H of G such that *all* the intra-node distances in H are stretched by at most a multiplicative factor of α and an additive term of β w.r.t. the corresponding distances in G. We show how an ordinary (α, β)-spanner of size $\sigma = \sigma(n, m)$ can be used to build in polynomial time an (α, β)-EABFS and an (α, β)-VABFS of size $O(\sigma)$ and $O(\sigma + n \log n)$, respectively. As a consequence, the EABFS problem is easier than the corresponding (non fault-tolerant) spanner problem, and we regard this as an interesting hardness characterization. Notice also that for all the significant values of α and β, the size of an (α, β)-spanner is $\omega(n \log n)$, which essentially means that the VABFS problem is easier than the corresponding spanner problem as well. This bridge between the two problems is useful for building sparse $(1, \beta)$-VABFS structures

[2] We do not insist on the time efficiency in building our structures, since the focus of our paper, consistently with the literature, is on the trade-off between their size and their stretch factor.

by making use of the vast literature on additive $(1, \beta)$-spanners. For instance, the $(1, 4)$-spanner of size $O(n^{\frac{7}{5}} \operatorname{polylog}(n))$ given in [6], and the $(1, 6)$-spanner of size $O(n^{\frac{4}{3}})$ given in [2], can be used to build corresponding vertex-fault-tolerant structures. Another interesting implication arises for the multisource `EABFS` problem. Indeed, given a set of multiple sources $S \subseteq V(G)$, the (α, β)-spanner of size σ can be used to build a multisource (α, β)-`EABFS` of size $O(n \cdot |S| + \sigma)$. This allows to improve, for $|S| = \omega(n^{\frac{1}{15}} \operatorname{polylog}(n))$, the multisource $(1, 4)$-`EABFS` of size $O(n^{\frac{4}{3}} \cdot |S|)$ given in [13]: indeed, it suffices to plug-in in our method the $(1, 4)$-spanner of size $O(n^{\frac{7}{5}} \operatorname{polylog}(n))$ given in [6].

Other related results. Besides fault-tolerant (approximate) SPT and BFS, there is a large body of literature on fault-tolerant short(est) paths in graphs. A natural counterpart of the structures considered in this paper, as we have seen before, are the DSO. For recent achievements on DSO, we refer the reader to [4,8], and more in particular to [3,10], where single-source distances are considered. Another setting which is very close in spirit to ours is that of *fault-tolerant spanners*. In [7], for weighted graphs and any integer $k \geq 1$, the authors present a $(2k - 1, 0)$-spanner resilient to f vertex (resp., edge) failures of size $O(f^2 \cdot k^{f+1} \cdot n^{1+1/k} \cdot \log^{1-1/k} n)$ (resp., $O(f \cdot n^{1+1/k})$). This was later improved through a randomized construction in [9]. On the other hand, for the unweighted case, in [5] the authors present a general result for building a $(1, O(f \cdot (\alpha + \beta)))$-spanner resilient to f edge failures, by unioning an ordinary $(1, \beta)$-spanner with a fault-tolerant $(\alpha, 0)$-spanner resilient against up to f edge faults. Finally, we mention that in [1] it was introduced the resembling concept of *resilient spanners*, i.e., spanners such that whenever any edge in G fails, then the relative distance increases in the spanner are very close to those in G, and it was shown how to build a resilient spanner by augmenting an ordinary spanner.

2 Notation

We start by introducing our notation. For the sake of brevity, we give it for the case of edge failures, but it can be naturally extended to the node failure case.

Given a non-negatively real weighted, undirected, and 2-edge-connected graph G, we will denote by $w_G(e)$ or $w_G(u, v)$ the weight of the edge $e = (u, v) \in E(G)$. We also define $w(G) = \sum_{e \in E(G)} w(e)$. Given an edge $e = (u, v)$, we denote by $G - e$ or $G - (u, v)$ (resp., $G + e$ or $G + (u, v)$) the graph obtained from G by removing (resp., adding) the edge e. Similarly, for a set F of edges, $G - F$ (resp., $G + F$) will denote the graph obtained from G by removing (resp., adding) the edges in F.

We will call $\pi_G(x, y)$ a shortest path between two vertices $x, y \in V(G)$, $d_G(x, y)$ its (weighted) length, and $T_G(s)$ a SPT of G rooted at s. Whenever the graph G and/or the vertex s are clear from the context, we might omit them, i.e., we will write $\pi(u)$ and $d(u)$ instead of $\pi_G(s, u)$ and $d_G(s, u)$, respectively. When considering an edge (x, y) of an SPT we will assume x and y to be the closest and the furthest endpoints from s, respectively.

Algorithm 1. Algorithm for building an $(1+\varepsilon)$-EASPT

```
Input  : A graph $G$, $s \in V(G)$, $\varepsilon > 0$
Output: A $(1+\varepsilon)$-EASPT of $G$ rooted at $s$
1 $H \leftarrow$ compute a 3-EASPT of size $O(n)$ using the algorithm in Sect. 3.1.1 of [11].
2 for $e \in E(T_G(s))$ in preorder w.r.t. $T_G(s)$ do
3     for $t \in V(G)$ in preorder w.r.t. $T_G^{-e}(s)$ do
4         if $d_H^{-e}(t) > (1+\varepsilon)d_G^{-e}(t)$ then    /* vertex $t$ is bad for edge $e$ */
5             Select a set of edges $S \subseteq E(\pi_G^{-e}(t))$ (see details after Lemma 1)
6             $H \leftarrow H + S$
7 return $H$
```

Given an edge $e \in E(G)$, we define $\pi_G^{-e}(x,y)$, $d_G^{-e}(x,y)$ and $T_G^{-e}(s)$ to be, respectively, a shortest path between x and y, its length, and a SPT in the graph $G-e$. Moreover, if P is a path from x to y and Q is a path from y to z, with $x, y, z \in V(G)$, we will denote by $P \circ Q$ the path from x to z obtained by concatenating P and Q.

Given G, a vertex $s \in V(G)$, and an edge $e=(u,v) \in E(T_G(s))$, we denote by $U_G(e)$ and $D_G(e)$ the partition of $V(G)$ induced by the two connected components of $T(G)-e$, such that $U_G(e)$ contains s and u, and $D_G(e)$ contains v. Then, $C_G(e) = \{(x,y) \in E(G) : x \in U_G(e), y \in D_G(e)\}$ will denote the *cutset* of e, i.e., the set of edges crossing the cut $(U_G(e), D_G(e))$.

For the sake of simplicity we consider only edge weights that are strictly positive. However our entire analysis also extends to non-negative weights. Throughout the rest of the paper we will assume that, when multiple shortest paths exist, ties will be broken in a consistent manner. In particular we fix a SPT $T = T_G(s)$ of G and, given a graph $H \subseteq G$ and $x, y \in V(H)$, whenever we compute the path $\pi_H(x,y)$ and ties arise, we will prefer the edges in $E(T)$. We will also assume that if we are considering a shortest path $\pi_H(x,y)$ between x and y passing through vertices x' and y', then $\pi_H(x',y') \subseteq \pi_H(x,y)$.

3 A $(1+\varepsilon)$-EASPT Structure

First, we give a high-level description of our algorithm for computing a $(1+\varepsilon)$-EASPT (see Algorithm 1). We build our structure, say H, by starting from an SPT T rooted at s which is suitably augmented with at most $n-1$ edges in order to make it become a 3-EASPT. Then, we enrich H incrementally by considering the tree edge failures in preorder, and by checking the disconnected vertices. When an edge e fails and a vertex t happens to be too stretched in $H-e$ w.r.t. its distance from s in $G-e$, we add a suitable subset of edges to H, selected from the new shortest path to t. This is done so that we not only adjust the distance of t, but we also improve the stretch factor of a *subset* of its predecessors. This is exactly the key for the efficiency of our method, since altogether, up to a logarithmic factor, we maintain constant in an amortized sense the ratio between the size of the set of added edges and the overall distance improvement.

Let us now provide a detailed description of our algorithm. To build the initial 3-EASPT, it augments T by making use of a *swap algorithm* devised in [11]. More precisely, in that paper the authors were concerned with the problem of reconnecting in a best possible way (w.r.t. to a set of distance criteria) the two subtrees of an SPT undergoing an edge failure, through a careful selection of a *swap edge*, i.e., an edge with an endvertex in each of the two subtrees. In particular, they show that if we select as a swap edge for $e = (u, v)$ – with u closer to the source s than v – the edge that lies on a shortest path in $G - e$ from s to v, then the distances from the source towards all the disconnected vertices is stretched at most by a factor of 3.[3] Therefore, a 3-EASPT of size at most $2n$ can be obtained by simply adding to a SPT rooted at s a such swap edge for each corresponding tree edge, and interestingly this improves the 3-EASPT of size at most $4n$ provided in [13].

Then, our algorithm works in $n - 1$ *phases*, where each phase considers an edge of T w.r.t. to a fixed preorder of the edges, say $e_1, \ldots, e_{n-1}$. In the h-th phase, the algorithm considers the failure of e_h, and when a vertex t happens to be too stretched in H w.r.t. $d^{-e_h}(t)$, then we say that t is *bad* for e_h and we add a suitable subset S of edges to H. These edges are selected from $\pi^{-e_h}(t)$ and they always include the last edge of $\pi^{-e_h}(t)$. We now show that this suffices to prove the correctness of the algorithm:

Lemma 1. *The structure H returned by the algorithm is a $(1 + \varepsilon)$-EASPT.*

Proof. Let $\widetilde{H}$ be the structure built by the algorithm just before a bad vertex t for an edge e_h is considered. Assume by induction that, for every vertex z in $T_G^{-e_h}(s)$ already considered in phase h, we have $d_{\widetilde{H}}^{-e_h}(z) \leq (1 + \varepsilon)d^{-e_h}(t)$. Let $f = (z, t)$ be the last edge of $\pi^{-e_h}(t)$ and recall that f is always added to $\widetilde{H}$. Hence we have:

$$d_H^{-e_h}(t) \leq d_{\widetilde{H}+f}^{-e_h}(t) \leq d_{\widetilde{H}}^{-e_h}(z) + w(f) \leq (1 + \varepsilon)d^{-e_h}(z) + w(f)$$
$$\leq (1 + \varepsilon)(d^{-e_h}(z) + d^{-e_h}(z, t)) = (1 + \varepsilon)d^{-e_h}(t). \qquad \square$$

It remains to describe the edge selection process and to analyze the size of our final structure. Let H_0 be the initial 3-EASPT structure. Let us fix the failed edge $e = (u, v)$ and a single bad vertex t for e. We call H' the structure built by the algorithm just before t is considered. Let $f = (x, y)$ be the unique edge in $C_G(e) \cap E(\pi_G^{-e}(t))$. Consider the subpath of $\pi_G^{-e}(t)$ going from x to t and let $x_0, x_1, \ldots, x_r$ be its vertices, in order. We consider the set $Z = \{x_i : (x_{i-1}, x_i) \notin E(H_0)\}$, we name its vertices $z_1, \ldots, z_k$ with $k = |Z| - 1$, in order and we let $z_0 = x$ (see Figure 1). We define $\alpha_i = \frac{d_{H'}^{-e}(z_i)}{d^{-e}(z_i)}$. It follows from the definitions and from Lemma 1 that we have $\alpha_0 = 1$, $\alpha_j \leq (1 + \varepsilon)$ for $1 \leq j < k$ and $\alpha_k > 1 + \varepsilon$.

Think of the edges in $\pi^{-e}(t)$ as being directed towards t for a moment. In the following we will describe how to select the set S of edges used by the algorithm.

[3] Actually, in [11] it is not explicitly claimed the 3-stretch factor, but this is implicitly obtained by the qualitative analysis of the swap procedure therein provided.

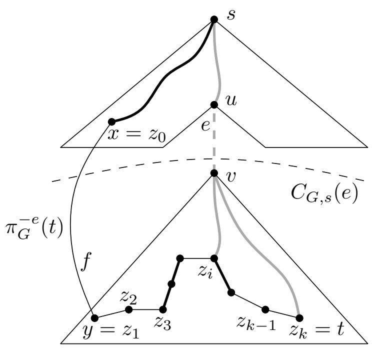


Fig. 1. Edge selection phase of Algorithm 1 when a bad vertex t for the failing edge e is considered. Bold edges belong to H_0 while the black path is $\pi_G^{-e}(t)$.

In particular, we will select $\eta \geq 1$ edges entering into the last η vertices in Z. This choice of S will ensure that the overall decrease of the values α_i in $H' + S$ will be at least $\frac{\varepsilon}{\mathcal{H}_n}\eta$ where $\mathcal{H}_n$ denotes the n-th *harmonic number.*

We exploit the fact that, after adding the set S, each "new value" α_i with $i > k - \eta = j$, will not be larger than α_j as we will show in the following.

Consider the sequence $\gamma_0, \dots, \gamma_k$ where $\gamma_i = 1 + \frac{\varepsilon}{\mathcal{H}_k}(\mathcal{H}_k - \mathcal{H}_{k-i})$. Notice that the sequence is monotonically increasing from $\gamma_0 = 1$ to $\gamma_k = 1+\varepsilon$. Let $0 \leq j < k$ be the largest index such that $\alpha_j \leq \gamma_j$. Notice that j always exists as $\alpha_0 = \gamma_0$ and that $\alpha_k > \gamma_k$. We set $\eta = k - j$ so that the set S is defined accordingly. Let $U = \{z_{j+1}, \dots, z_k\}$ be the set of vertices for which an incoming edges has been added in S.

For every vertex $z \in U$ we define the following path in $H' + S$: $P(z) = \pi_{H'}^{-e}(z_j) \circ \pi(z_j, z)$. Notice that $\pi(z_j, z)$ is entirely contained in $H' + S$. We define $\alpha'_i = \frac{w(P(z_i))}{d^{-e}(z_i)}$, and note that α'_i is an upper bound to the stretch of z in $H' + S$.

Lemma 2. *For $i > j$, $\alpha'_i \leq \alpha_j < \alpha_i$.*

Proof. By definition of j, we have $\alpha_j \leq \gamma_j < \gamma_i < \alpha_i$. Now we prove $\alpha'_i \leq \alpha_j$:

$$\alpha'_i = \frac{w(P(z))}{d^{-e}(z_i)} = \frac{d_{H'}^{-e}(z_j) + d(z_j, z_i)}{d^{-e}(z_i)} \leq \frac{\alpha_j d^{-e}(z_j) + d^{-e}(z_j, z_i)}{d^{-e}(z_i)} \leq \frac{\alpha_j d^{-e}(z_i)}{d^{-e}(z_i)} = \alpha_j.$$

□

We now lower-bound the overall decrease of the values α'_i's w.r.t. the corresponding α_i's by using the following inequalities:

$$\sum_{z \in U} \left(\frac{d_{H'}^{-e}(z)}{d^{-e}(z)} - \frac{w(P(z))}{d^{-e}(z)} \right) = \sum_{i=j+1}^{k} (\alpha_i - \alpha'_i) \geq \sum_{i=j+1}^{k} (\alpha_i - \alpha_j) \geq \sum_{i=j+1}^{k} (\gamma_i - \gamma_j)$$

$$= \frac{\varepsilon}{\mathcal{H}_k} \sum_{i=j+1}^{k} (\mathcal{H}_{k-j} - \mathcal{H}_{k-i}) = \frac{\varepsilon}{\mathcal{H}_k}(k - j) \geq \frac{\varepsilon}{\mathcal{H}_n}\eta.$$

where in the last but one step we used the well-known equality that for every $j \le k$, $\sum_{i=j+1}^{k} (\mathcal{H}_{k-j} - \mathcal{H}_{k-i}) = k - j$.

The above selection procedure is repeated by the algorithm for every failed edge e_h and for every corresponding bad vertex. We now focus on the h-th phase of the algorithm. Let U_h be the union of all the sets U used when considering the bad vertices of the phase h. Moreover let $V_h = \bigcup_{i=1}^{h} U_i$ and notice that $V_0 = \emptyset$. For a vertex $z \in U_h$, let $P_h(z)$ be the *last* path $P(z)$ built by the algorithm, as defined above. Let H_h (resp., H'_h) be the structure built by the algorithm at the end (resp., start) of the phase h and let m_h be the number of new edges added during the phase h. By summing over all the bad vertices for edge e_h, we have:

Lemma 3. $\displaystyle\sum_{z\in U_h} \left(\frac{d^{-e_h}_{H'_h}(z)}{d^{-e_h}(z)} - \frac{w(P_h(z))}{d^{-e_h}(z)} \right) \ge m_h \frac{\varepsilon}{\mathcal{H}_n}$.

Now, let us define a function $\phi_h(z)$ for every $z \in V$:

$$\phi_h(z) = \begin{cases} 0 & \text{if } z \notin V_h \\ w(P_h(z)) & \text{if } z \in U_h \\ \phi_{h-1}(z) & \text{if } z \in V_h \setminus U_h \end{cases}$$

The proofs of next three lemmas are postponed to the full version of the paper.

Lemma 4. *For every $z \in U_h$ we have $d^{-e_h}_{G}(z) < \frac{2}{\varepsilon} d_G(z)$.*

Lemma 5. *For $z \in V_{h-1}$, $\phi_{h-1}(z) \ge d^{-e_h}_{H'_h}(z)$.*

Lemma 6. *For $z \in U_h$, $d^{-e_h}_{H'_h}(z) \ge w(P_h(z))$.*

We now prove the following:

Lemma 7. $\displaystyle\sum_{z\in U_h} \frac{\phi_{h-1}(z)}{d(z)} - \sum_{z\in U_h} \frac{\phi_h(z)}{d(z)} \ge m_h \frac{\varepsilon}{\mathcal{H}_n} - |U_h \setminus V_{h-1}| \frac{6}{\varepsilon}$.

Proof. By Lemmas 3–6, and since the initial structure H_0 is a 3-`EASPT`, we have:

$$\sum_{z\in U_h} \frac{\phi_{h-1}(z)}{d(z)} - \sum_{z\in U_h} \frac{\phi_h(z)}{d(z)} \ge \sum_{z\in U_h \cap V_{h-1}} \left(\frac{\phi_{h-1}(z)}{d(z)} - \frac{\phi_h(z)}{d(z)} \right) + \sum_{z\in U_h \setminus V_{h-1}} \left(\frac{\phi_{h-1}(z)}{d(z)} - \frac{\phi_h(z)}{d(z)} \right)$$

$$\ge \sum_{z\in U_h \cap V_{h-1}} \left(\frac{d^{-e_h}_{H'_h}(z)}{d(z)} - \frac{\phi^{-e}_h(z)}{d(z)} \right) + \sum_{z\in U_h \setminus V_{h-1}} - \frac{\phi_h(z)}{d(z)}$$

$$= \sum_{z\in U_h \cap V_{h-1}} \left(\frac{d^{-e_h}_{H'_h}(z)}{d(z)} - \frac{w(P_h(z))}{d(z)} \right) + \sum_{z\in U_h \setminus V_{h-1}} \left(\frac{d^{-e_h}_{H'_h}(z)}{d(z)} - \frac{w(P_h(z))}{d(z)} \right) - \sum_{z\in U_h \setminus V_{h-1}} \frac{d^{-e_h}_{H'_h}(z)}{d(z)}$$

$$\ge \sum_{z\in U_h \cap V_{h-1}} \left(\frac{d^{-e_h}_{H'_h}(z)}{d^{-e_h}(z)} - \frac{w(P_h(z))}{d^{-e_h}(z)} \right) + \sum_{z\in U_h \setminus V_{h-1}} \left(\frac{d^{-e_h}_{H'_h}(z)}{d^{-e_h}(z)} - \frac{w(P_h(z))}{d^{-e_h}(z)} \right) - \sum_{z\in U_h \setminus V_{h-1}} \frac{3d^{-e_h}(z)}{d(z)}$$

$$\ge \sum_{z\in U_h} \left(\frac{d^{-e_h}_{H'_h}(z)}{d^{-e_h}(z)} - \frac{w(P_h(z))}{d^{-e_h}(z)} \right) - \sum_{z\in U_h \setminus V_{h-1}} \frac{3\frac{2}{\varepsilon}d(z)}{d(z)} \ge m_h \frac{\varepsilon}{\mathcal{H}_n} - |U_h \setminus V_{h-1}| \frac{6}{\varepsilon}. \qquad \square$$

We now define a global potential function Φ:

$$\Phi(h) = \sum_{z \in V_h} \frac{\phi_h(z)}{d(z)}$$

for $0 < h \leq n-1$. Notice that we trivially have $\Phi(h) \geq 0$.

Theorem 1. *The structure H returned by the algorithm is a $(1+\varepsilon)$-EASPT of size $O(\frac{n \log n}{\varepsilon^2})$.*

Proof. The fact that H is a $(1+\varepsilon)$-`EASPT` follows from Lemma 1. Concerning the size of H, since H_0 contains $O(n)$ edges, we only focus on bounding the number $\mu = \sum_{h=1}^{n-1} m_h$ of edges in $E(H) \setminus E(H_0)$. Using Lemma 7, we can write:

$$\begin{aligned}
\Phi(i) &= \sum_{z \in V_{i-1} \setminus U_i} \frac{\phi_i(z)}{d(z)} + \sum_{z \in U_i} \frac{\phi_i(z)}{d(z)} \\
&= \sum_{z \in V_{i-1} \setminus U_i} \frac{\phi_{i-1}(z)}{d(z)} + \sum_{z \in U_i} \frac{\phi_{i-1}(z)}{d(z)} - \left(\sum_{z \in U_i} \frac{\phi_{i-1}(z)}{d(z)} - \sum_{z \in U_i} \frac{\phi_i(z)}{d(z)} \right) \\
&\leq \sum_{z \in V_{i-1} \setminus U_i} \frac{\phi_{i-1}(z)}{d(z)} + \sum_{z \in V_{i-1} \cap U_i} \frac{\phi_{i-1}(z)}{d(z)} + \sum_{z_i \in U_i \setminus V_{i-1}} \frac{\phi_{i-1}(z)}{d(z)} - m_h \frac{\varepsilon}{\mathcal{H}_n} + |U_h \setminus V_{h-1}| \frac{6}{\varepsilon} \\
&\leq \Phi(i-1) + 0 - m_h \frac{\varepsilon}{\mathcal{H}_n} + |U_h \setminus V_{h-1}| \frac{6}{\varepsilon}.
\end{aligned}$$

Unfolding the previous recurrence relation we obtain:

$$0 \leq \Phi(n-1) \leq |V_{n-1}| \frac{6}{\varepsilon} - \frac{\varepsilon}{\mathcal{H}_n} \sum_{h=0}^{n-1} m_h \leq n \frac{6}{\varepsilon} - \frac{\varepsilon}{\mathcal{H}_n} \mu$$

which we finally solve for μ to get $\mu = O(\frac{n \log n}{\varepsilon^2})$. □

4 A $(1+\varepsilon)$-VASPT Structure

In this section we extend our previous $(1+\varepsilon)$-`EASPT` structure to deal with vertex failures. In order to do so we will build a different subgraph H_0 having suitable properties that we will describe. Then we will use the natural extension of Algorithm 1 where we consider (in preorder) vertex failures instead of edge failures. We now describe the construction of H_0 and then argue how the previous analysis can be adapted to show the same bound on the size of H.

The structure H_0 is initially equal to T and it is augmented by using a technique similar to the one shown in [3]: the SPT T of G is suitably decomposed into ancestor-leaf vertex-disjoint paths. Then, for each path, an approximate structure is built. This structure will provide approximate distances towards any vertex of the graph when any vertex along the path fails. The union of T with all those structures will form H_0.

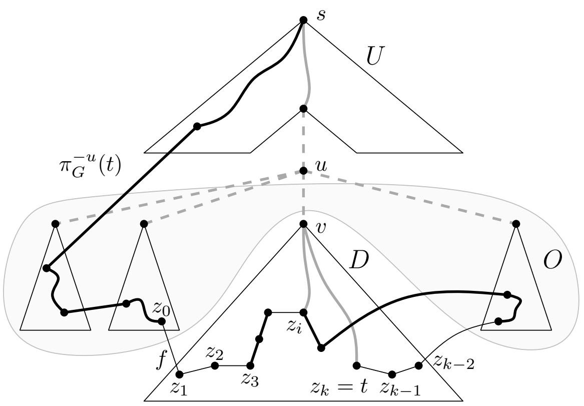


Fig. 2. Edge selection phase of the vertex-version of Algorithm 1 when a bad vertex t for the failing vertex u is considered. Bold edges belong to H_0 while the black path is $\pi_G^{-u}(t)$. Notice that all z_is belong to the down set D.

Fix a path Q of the previous decomposition starting from a vertex q, and let T_q be the subtree of T rooted at q. Moreover, let $u \in V(Q)$ be a failing vertex, and let v be the next vertex in Q.[4] We partition the vertices of the forest $T - u$ into three sets: (i) the *up set* U containing all the vertices of the tree rooted at s, (ii) the *down set* D containing all the vertices of the tree rooted at v, and (iii) the *others set* O containing all the remaining vertices (see Figure 2).

We want to select a set of edges to add to H. In order to do so, we construct a SPT T' of $G - u$ and we imagine that its edges are directed towards the leaves. We select all the edges of $E(T') \setminus E(T)$ that do not lead to a vertex in D, plus the unique edge of $\pi^{-u}(v)$ that crosses the cut induced by the sets $U \cup O$ and D. Notice that $T - u$ contains all the paths in T' towards the vertices in U, and that each vertex has at most one incoming edge in T'. This implies that the number of selected edges is at most $|O| + 1$.

The above procedure is repeated for all the failing vertices of Q, in order. As the sets O associated with the different vertices are disjoint we have that, while processing Q, at most $|V(T_q)| + |Q| = O(|V(T_q)|)$ edges are selected. We use the path decomposition described in [3] that can be recursively defined as follows: given a tree, we select a path Q from the root to a leaf such that the removal of Q splits the tree into a forest where the size of each subtree is at most half the size of the original tree. We than proceed recursively on each subtree. Using this approach, the size of the entire structure H_0 can be shown to be $O(n \log n)$ [3].

We now prove some useful properties of the structure H_0. First of all, observe that, by construction and similarly to the edge-failure case, we immediately have:

[4] W.l.o.g. we are assuming that the failing vertex u is not a leaf, as otherwise $T - u$ is already a SPT of $G - u$.

Lemma 8. *Consider a failed vertex u and another vertex $z \neq u$. We have: (i) $d^{-u}_{H_0}(v) = d^{-u}(v)$, and (ii) for $z \in D$, it holds $d^{-u}_{H_0}(z) \leq 3d^{-u}(z)$.*

Moreover, we also have the following (proof postponed to the full version of the paper):

Lemma 9. *Consider a failed vertex u. During the execution of the vertex-version of Algorithm 1, every bad vertex t for u will be in D.*

At this point, the same analysis given for the case of edge failures can be retraced for vertex failures as well. We point out that Lemma 9 ensures that every bad every for u is in the same subtree as v. Also notice that all the vertices z_i's are, by definition, in the same subtree as well (see Figure 2). The above, combined with Lemma 8 (i), is needed by the proof of Lemma 4, while Lemma 8 (ii) is used in the proof of Lemma 7. Hence we have:

Theorem 2. *The vertex-version of Algorithm 1 computes a $(1 + \varepsilon)$-VASPT of size $O(\frac{n \log n}{\varepsilon^2})$.*

5 Relation with (α, β)-Spanners in Unweighted Graphs

In this section we turn our attention to the unweighted case, and we provide two polynomial-time algorithms that augment an (α, β)-spanner of G so to obtain an (α, β)-EABFS/VABFS. We present the algorithm for the vertex-failure case and show how it can be adapted to the edge-failure case.

The algorithm first augments the structure H_0 computed so as explained in Section 4 and then adds its edges to the (α, β)-spanner of G. The structure H_0 is augmented as follows. The vertices of the BFS of G rooted at s are visited in preorder. Let u be the vertex visited by the algorithm and let D be the set of vertices of the tree defined so as explained in Section 4 w.r.t the path decomposition computed for H_0. For every $t \in D$, the algorithm checks whether $\pi^{-u}_G(s,t)$ contains no vertex of $D \setminus \{t\}$ and $d^{-u}_G(s,t) < d^{-u}_{H_0}(s,t)$. If this is the case, then the algorithm augments H_0 with the edge of $\pi^{-u}_G(s,t)$ incident to t.

The following observation is crucial to prove the algorithm correctness.

Fact 1. *For every vertex u and every vertex $t \in V(G) \setminus \{u\}$ such that $\pi^{-u}_G(t)$ contains a vertex in D, let x and y be the first and last vertex of $\pi^{-u}_G(t)$ that belong to D, respectively. We have $d^{-u}_{H_0}(x) = d^{-u}_G(x)$ and $d^{-u}_{H_0}(y,t) = d^{-u}_G(y,t)$.*

We can now give the following (proof postponed to the full version of the paper):

Theorem 3. *Given an unweighted graph G with n vertices and m edges, a source vertex $s \in V(G)$, and an (α, β)-spanner for G of size $\sigma = \sigma(n,m)$, the algorithm computes an (α, β)-VABFS w.r.t. s of size $O\big(\sigma + n \log n\big)$.*

Now, we adapt the algorithm to prove a similar result for the (α, β)-EABFS. The algorithm first augments a BFS tree T of G rooted at s and then adds its edges to the (α, β)-spanner of G. The tree T is augmented by visiting its edges in preorder. Let e be the edge visited by the algorithm. For every $t \in D_G(e)$, the algorithm checks whether $\pi_G^{-e}(s,t)$ contains no vertex of $D_G(e) \setminus \{t\}$ and $d_G^{-e}(s,t) < d_T^{-e}(s,t)$. If this is the case, then the algorithm augments T with the edge of $\pi_G^{-e}(s,t)$ incident to t. In the full version of the paper it will be shown that the proof of Theorem 3 can be adapted to prove the following:

Theorem 4. *Given an unweighted graph G with n vertices and m edges, a source vertex $s \in V(G)$, and an (α, β)-spanner for G of size σ, the algorithm computes an (α, β)-EABFS w.r.t. s of size less than or equal to $\sigma + 3n$.*

References

1. Ausiello, G., Franciosa, P.G., Italiano, G.F., Ribichini, A.: On Resilient Graph Spanners. In: Bodlaender, H.L., Italiano, G.F. (eds.) ESA 2013. LNCS, vol. 8125, pp. 85–96. Springer, Heidelberg (2013)
2. Baswana, S., Kavitha, T., Mehlhorn, K., Pettie, S.: Additive spanners and (α, β)-spanners. ACM Trans. on Algorithms 7, A.5 (2010)
3. Baswana, S., Khanna, N.: Approximate shortest paths avoiding a failed vertex: near optimal data structures for undirected unweighted graphs. Algorithmica 66(1), 18–50 (2013)
4. Bernstein, A., Karger, D.R.: A nearly optimal oracle for avoiding failed vertices and edges. In: Proc. of the 41st Symp. on the Theory of Computing (STOC 2009), pp. 101–110. ACM Press (2009)
5. Braunschvig, G., Chechik, S., Peleg, D.: Fault Tolerant Additive Spanners. In: Golumbic, M.C., Stern, M., Levy, A., Morgenstern, G. (eds.) WG 2012. LNCS, vol. 7551, pp. 206–214. Springer, Heidelberg (2012)
6. Chechik, S.: New additive spanners. In: Proc. of the 24th Symp. on Discrete Algorithms (SODA 2013), pp. 498–512. ACM Press (2013)
7. Chechik, S., Langberg, M., Peleg, D., Roditty, L.: Fault-tolerant spanners for general graphs. In: Proc. of the 41st Symp. on the Theory of Computing (STOC 2009), pp. 435–444. ACM Press (2009)
8. Chechik, S., Langberg, M., Peleg, D., Roditty, L.: f-Sensitivity Distance Oracles and Routing Schemes. In: de Berg, M., Meyer, U. (eds.) ESA 2010, Part I. LNCS, vol. 6346, pp. 84–96. Springer, Heidelberg (2010)
9. Dinitz, M., Krauthgamer, R.: Fault-tolerant spanners: better and simpler. In: Proc. of the 30th Symp. on Principles of Distributed Computing (PODC 2011), pp. 169–178. ACM Press (2011)
10. Grandoni, F., Williams, V.V.: Improved distance sensitivity oracles via fast single-source replacement paths. In: Proc. of the 53rd Annual IEEE Symp. on Foundations of Computer Science (FOCS 2012), pp. 748–757 (2012)
11. Nardelli, E., Proietti, G., Widmayer, P.: Swapping a failing edge of a single source shortest paths tree is good and fast. Algorithmica 36(4), 361–374 (2003)
12. Parter, M., Peleg, D.: Sparse fault-tolerant BFS trees. In: Bodlaender, H.L., Italiano, G.F. (eds.) ESA 2013. LNCS, vol. 8125, pp. 779–790. Springer, Heidelberg (2013)
13. Parter, M., Peleg, D.: Fault tolerant approximate BFS structures. In: Proc. of the 25th Symp. on Discrete Algorithms (SODA 2014), pp. 1073–1092. ACM Press (2014)

Fast Witness Extraction Using a Decision Oracle*

Andreas Björklund[1], Petteri Kaski[2], and Łukasz Kowalik[3]

[1] Department of Computer Science, Lund University, Sweden
[2] Helsinki Institute for Information Technology HIIT, Department of Information and Computer Science, Aalto University, Finland
[3] Institute of Informatics, University of Warsaw, Poland

Abstract. The gist of many (NP-)hard combinatorial problems is to decide whether a universe of n elements contains a *witness* consisting of k elements that match some prescribed pattern. For some of these problems there are known advanced algebra-based FPT algorithms which solve the decision problem but do not return the witness. We investigate techniques for turning such a YES/NO-decision oracle into an algorithm for extracting a single witness, with an objective to obtain practical scalability for large values of n. By relying on techniques from combinatorial group testing, we demonstrate that a witness may be extracted with $O(k \log n)$ queries to either a deterministic or a randomized set inclusion oracle with one-sided probability of error. Furthermore, we demonstrate through implementation and experiments that the algebra-based FPT algorithms are practical, in particular in the setting of the k-path problem. Also discussed are engineering issues such as optimizing finite field arithmetic.

1 Introduction

The gist of many (NP-)hard combinatorial problems is to *decide* whether a universe of n elements contains a *witness* consisting of k elements that match some prescribed pattern. In the positive case this is naturally followed by the task of *extracting* the elements of one such witness.

As a result of advances in fixed-parameter tractability, many such hard problems are now known to admit algorithms that run in linear (or low-order polynomial) time in the size of the universe n, and where the complexity of the problem can be isolated to the size of the witness k. That is, the running times obtained are of the form $O(f(k) \cdot n)$ for some rapidly growing function $f(k)$ of k. This makes such algorithms ideal candidates for practical applications that must consider large inputs, that is, large values of n. For example, a recent randomized algorithm for the k-sized graph motif problem runs in time $O(2^k k^2 (\log k)^2 \cdot e)$, where e is the number of edges in the input graph [2].

* P.K. supported by the Academy of Finland, grants 252083 and 256287. Ł.K. supported by National Science Centre of Poland, grant number UMO-2013/09/B/ST6/03136.

Despite scalability to large inputs, some such advanced parameterized algorithms (like the ones for graph motif [2] or for k-path [1]) have an inherent handicap from a concrete algorithm engineering perspective. *They only solve the decision problem.* In applications, however, one needs access to the witnesses, which puts forth the question whether one can efficiently extract a witness or list all witnesses, using the algorithm for the decision problem as an *oracle* (black-box subroutine), and without losing the scalability to large inputs.

This paper studies the question of efficiently turning a decision oracle into an algorithm for witness extraction over the universe $U = \{1, 2, \ldots, n\}$. Let $\mathcal{F} \subseteq 2^U$ be the (unknown) family of witnesses. We focus on the following oracle:

Inclusion oracle. Given a query set $Y \subseteq U$, the oracle answers (either YES or NO) whether there exists at least one witness $W \in \mathcal{F}$ such that $W \subseteq Y$. We can motivate this type of oracle by observing that most problems have natural self-reducibility that we can use to narrow down the universe from U to Y (e.g. take the subgraph induced by the set Y of vertices) and then run the decision algorithm.

In the oracle setting there are at least two natural ways to measure the efficiency of witness extraction.

Number of oracle queries. This measure has been extensively studied in the domain of *combinatorial group testing* [7], where the canonical task is to identify k *defective* items from a population of n items, with the objective of minimizing the number of tests[1] (oracle queries) required to identify all the defectives. While this measure does not reflect accurately the amount of computing resources invested in our context—indeed, different oracle queries in general do not use the same amount of resources—the group testing perspective enables information-theoretic lower bounds and supplies useful algorithmic techniques for extraction.

Total running time. Assuming we have bounds on the running time of the oracle as a function of n and k, we can bound the running time of extraction of witnesses by taking the sum of the running times of the oracle queries. It turns out that we get fair control over the total running time already if we know that the running time of the oracle scales at least linearly in n.

The objectives of this paper are threefold. (a) First, we draw from techniques in classical group testing to arrive at efficient witness extraction algorithms for inclusion oracles both in deterministic and in randomized settings with one-sided error. (b) Second, we show examples of parameterized problems which can be solved efficiently *in practice* by a combination of an FPT decision oracle and a group-testing algorithm; in particular, for the k-path problem our experimental results show that one can find a 14-vertex witness in a 2000-vertex graph within a minute on a typical laptop. (c) Third, we discuss some non-obvious choices we made during the implementation: namely the choice of the GF(2^q) arithmetic

[1] In the setting of classical group testing, a single test on a set of items determines whether the set contains at least one defective item.

implementation; we believe our findings might be useful for implementations of other algorithms applying GF(2^q) arithmetic.

To set up a trivial baseline for performance comparisons, it is not difficult to see that $\Theta(n)$ queries to an inclusion oracle suffice to extract a witness—simply delete points from the universe one by one, with each deletion followed by an oracle query on the remaining points. If the oracle answers NO, we know the deleted point was essential and insert it back. When the process finishes the points that remain form a witness. This, however, is not particularly efficient since each oracle query costs at least $O(f(k) \cdot n)$ time, raising the total running time to $O(f(k) \cdot n^2)$ and making the approach impractical for large n.

Our Results on Extraction. We begin by transporting techniques from group testing [7] to arrive at more efficient witness extraction. Our first contribution merely amounts to observing that the so-called *bisecting algorithm* [6] can be translated to work with an inclusion oracle and in the presence of one or more witnesses. We also observe that taking into account the total running time of the algorithm, the baseline cost of a factor $O(n)$ in running time can be lowered to $O(k)$ if the running time of the oracle is at least linear in n, which is the case in most applications. These observations are summarized in Theorem 1.1.

Let $\mathcal{F}$ be a nonempty family witnesses, each of size at most k, over an n-element universe, $n, k \geq 1$. We say that a function $g : \mathbb{N} \to \mathbb{N}$ is *at least linear* if for all $n_1, n_2 \in \mathbb{N}$ it holds that $g(n_1) + g(n_2) \leq g(n_1 + n_2)$.

Theorem 1.1 (Deterministic Extraction). *There exists an algorithm that extracts a witness in $\mathcal{F}$ without knowledge of k using at most*

$$Q(n, k) = 2k\left(\log_2 \frac{n}{k} + 2\right)$$

queries to a deterministic inclusion oracle. Moreover, suppose the oracle runs in time $T(n, k) = O(f(k)g(n))$ for a function g that is at least linear. Then, there exists an algorithm that extracts a witness in $\mathcal{F}$ in time $O(k \cdot T(2n, k)) = O(f(k) \cdot k \cdot g(2n))$.

Currently the fastest known parameterized algorithms in many cases use randomization. Thus in practice one must be able to cope with decision oracles that may give erroneous answers, for example it is typically the case that the decision algorithm produces false negatives with at most some small probability, but false positives do not occur [1,2,13,12].

Let us assume that the probability of a false negative is $p \leq \frac{1}{4}$. Beyond the absence of false positives, a further observation to our advantage is that typically *witnesses may be checked*, deterministically, and essentially at no computational cost compared with the execution of even one oracle query. That is, we have available a subroutine that takes a candidate witness $W \subseteq U$ as input and returns whether $W \in \mathcal{F}$. We make this assumption in what follows. Thus having access to a randomized inclusion oracle enables deterministic extraction, but with randomized running time. These observations are summarized in Theorem 1.2.

Theorem 1.2 (Las Vegas Extraction). *There exists an algorithm that extracts a witness in $\mathcal{F}$ without knowledge of k using in expectation at most $O(k \log n)$ queries to a randomized inclusion oracle that has no false positives but may output a false negative with probability at most $p \leq \frac{1}{4}$. Moreover, suppose the oracle runs in time $T(n, k) = O(f(k)g(n))$ for a function g that is at least linear. Then, there exists an algorithm that extracts a witness in $\mathcal{F}$ in time $O(k \cdot T(2n, k) + (k \log k) \cdot T(2k, k))$.*

An Example Application: k-Path. The k-path problem is one of the basic NP-complete problems, a natural parameterized version of the Hamiltonian Path problem. In this problem we are given an undirected connected graph $G = (V, E)$, and a natural number k. The goal is to find a simple path on k vertices in G. Denote by $n = |V|$ and $m = |E|$. In terms of dependence on k, the currently fastest algorithm is due to Björklund, Husfeldt, Kaski, and Koivisto [1] and can be tuned to run in $1.66^k k^{O(1)} m$ time. It uses algebraic tools and only solves the corresponding decision problem. We applied a simplified version of this algorithm, slightly easier to implement, which runs in $O(2^k km)$ time, assuming that finite field arithmetic operations take constant time (cf. [4]). The algorithm evaluates a certain polynomial of degree $d = 2k - 1$ over the finite field $\mathrm{GF}(2^q)$, which turns out to be a generating function of all witnesses. The algorithm is randomized, and it may return a false negative. The failure probability is bounded by $\frac{2k-1}{2^q}$, hence by choosing q large enough we can assume it is at most $\frac{1}{4}$, as required by Theorem 1.2.

Our universe U is the set of edges of the input graph and we are extracting witnesses with exactly $k - 1$ edges. By Theorem 1.2 we obtain an algorithm with expected running time $O(2^k k^2 \cdot m)$ for witness extraction.

However, when we consider actual *implementation* the above approach should be refined as follows. First set the universe U to be the set of *vertices* and find the set of k vertices S which contains a k-vertex path. Next, set the universe U to be the set of *edges* in the induced graph $G[S]$, and find the witness. By Theorem 1.2, for dense graphs this can give a factor two speed-up.

Further applications in FPT algorithms will be presented in a full version of this work.

Related and Previous Work. The relations between the time complexity of decision problems and their search versions were studied by Fellows and Langston [9].

Independently of our work, Hassidim, Keller, Lewenstein, and Roditty [11] presented a *randomized* algorithm that extracts a witness for the (weighted) k-path problem using $O(k \log n)$ calls to a decision oracle, *in expectation.* Their approach is to discard random subsets (of size n/k) of the vertex set as long as the resulting instance still contains the solution. The bisecting algorithm [6] that we extend in this paper can be seen as a cleaner version of this idea. First, in the bisecting algorithm larger sets get discarded. Second, the bisecting algorithm is deterministic. Hassidim et al. do not analyze how the time of their algorithm is influenced by the fact that the oracle is randomized. From an asymptotic perspective this is not needed because one can repeat each oracle call multiple times to reduce the error probability below an arbitrary threshold. However, in

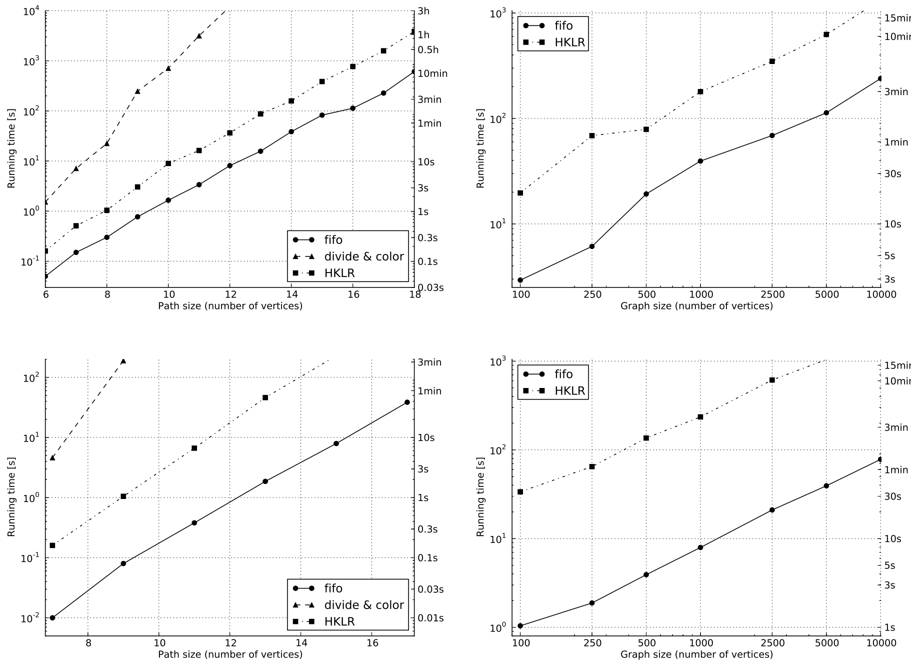


Fig. 1. Running times of various algorithms for a graph with exactly one witness (upper charts) and $\Omega(n^2)$ witnesses (lower charts). Each running time on the graph is the median of 5 runs for the same input instance. The left charts: a 1000-vertex graph and $k \in \{6, 7, \ldots, 18\}$. The right charts: $k = 14$ (upper) or $k = 15$ (lower) and the number of vertices varies. Running times on a 2.53-GHz Intel Xeon CPU.

practice this is an unnecessary (though only constant-factor) slow-down, which we seek to avoid in what follows.

Implementation and Experiments. We implemented in C the $O(2^k km)$-time decision algorithm for the k-path problem and the algorithm from Theorem 1.2, which we call 'fifo' on the charts. The crucial part of implementation of the decision oracle is the finite field arithmetic. Somewhat unexpectedly, we found that to optimize the running time, a *different* method should be chosen depending on whether we use the oracle just once (e.g. check whether there is a witness) or whether it is used in combination with the algorithm from Theorem 1.2 to find a witness. Details can be found in Section 4.

We run a series of experiments on a single 2.53-GHz Intel Xeon CPU. We compare the fifo algorithm with two other natural candidates. The first is the witness extraction algorithm of Hassidim et al. [11] combined with the $O(2^k km)$-time inclusion oracle, called 'HKLR' on the charts. The second is the $O(4^k k^{2.7} m)$-time algorithm of Chen et al. [3] called 'Divide-and-Color'. It is not based on algebraic tools and finds the witness while solving the decision problem. Note that there are many more algorithms/heuristics for k-path problem which would be much faster on particular instances. A natural heuristic is computing the DFS tree. If the tree has depth at least k the witness is found and otherwise the graph has pathwidth at most k. On the other hand, when the pathwidth p is very small (say, $p \leq \frac{k}{2}$), the $(2+\sqrt{2})^p n^{O(1)}$ algorithm of Cygan et al. [5] should be fast. However, in this work we want to focus on algorithms with best guarantees *in the worst case.* Disregarding the detailed memory layout of the input graph, all the three algorithms we compare are oblivious to the topology of the graph apart from the parameters m and k. In our experiments we use two types of trees with $m = n - 1$ as the input graphs. The first type (with a unique witness) consists of $\lfloor (k-1)/2 \rfloor$-vertex paths joined at a common endvertex; when k is odd two of the paths are extended by an edge, when k is even one path is extended by two edges and one path by one edge. The second type (with $\Omega(n^2)$ witnesses) has k odd and all paths are extended by an edge.

The results can be seen on Fig. 1. We see that both fifo and HKLR are much faster than Divide-and-Color even for very small values of k. For 1000-vertex graphs our algorithm fifo finds ($\leq$ 10)-vertex patterns below 1 second and ($\leq$ 20)-vertex patterns below 1 hour. HKLR is considerably slower and the difference is more visible when there are many witnesses.

2 Extracting a Witness Using a Deterministic Oracle

The objective of this section is to prove Theorem 1.1. Accordingly, we assume we have available a deterministic inclusion oracle. Our strategy is to translate an existing algorithm developed for group testing into the setting of witness extraction (Algorithm 1 and Lemma 2.1), and then analyze its performance with respect to the total running time, including the oracle queries (Lemma 2.2).

Let us first review the setting of classical group testing, and then indicate how to translate classical algorithms to the setting of witness extraction. In group testing, we do not have a family of witnesses, but rather a *single* unknown set $D \subseteq U$ consisting of *defective* items. Furthermore, instead of an inclusion oracle (that would test whether $D \subseteq Y$ for a query Y) we have an *intersection oracle* that answers whether $D \cap Y \neq \emptyset$ for a query Y. That is, a query tells us whether the query set Y has at least one defective item.

Characteristic to classical group testing algorithms is that they proceed to shrink down the size of the universe U while maintaining the invariant $D \subseteq U$ until D has been identified (that is, $D = U$). Indeed, whenever the (intersection) oracle answers NO, we know that the query Y is disjoint from D, and thus can safely delete all points in Y from U without violating the invariant.

In our setting we have to work with an inclusion oracle and cope with the possibility of the family $\mathcal{F}$ containing more than one witness. Fortunately, it turns out that the setting is not substantially different from group testing. Indeed, in analogy with group testing, we will also proceed to narrow down the universe U but seek to maintain a slightly different invariant, namely "there exists a $W \in \mathcal{F}$ such that $W \subseteq U$". In this setting we can narrow down the universe by the following basic procedure: for a subset $A \subseteq U$ we query the inclusion oracle with

Algorithm 1. ExtractInclusion(U)

```
1  Initialize an empty FIFO queue $\mathcal{Q}$;
2  Let $W \leftarrow \emptyset$;
3  Insert $U$ into $\mathcal{Q}$;
4  while $\mathcal{Q}$ is not empty do
5      Remove the first set $A$ from $\mathcal{Q}$;
6      if $|A| = 1$ then
7          Let $W \leftarrow W \cup A$;
8      else
9          Partition $A$ into $A_1$ and $A_2$ arbitrarily so that $||A_1| - |A_2|| \leq 1$;
10         if Includes($U \setminus A_1$) then
11             Let $U \leftarrow U \setminus A_1$;
12             Insert $A_2$ into $\mathcal{Q}$;
13         else
14             if Includes($U \setminus A_2$) then
15                 Let $U \leftarrow U \setminus A_2$;
16                 Insert $A_1$ into $\mathcal{Q}$;
17             else
18                 Insert both $A_1$ and $A_2$ into $\mathcal{Q}$;
19 return $W$
```

$Y = U \setminus A$. If the answer is YES, we know that we can safely remove A from U while maintaining the invariant. This basic analogy enables one to transport group testing algorithms into the setting of witness extraction.

In what follows we focus on a translation of one such algorithm, the *bisecting algorithm* [6]. One of its advantages is that it does not need to know the number of defective items in advance, and hence in particular it is suitable for our applications where we want to allow the witnesses to potentially differ in size. Moreover, this particular algorithm is convenient in our further modifications for the randomized oracle model (Sect. 3). We give the pseudocode of a "witness extraction" version of the bisecting algorithm in pseudocode as Algorithm 1.

The correctness of Algorithm 1 follows from the fact that our invariant "there exists a $W \in \mathcal{F}$ such that $W \subseteq U$" is always satisfied. We remark that Algorithm 1 has a further minor difference with the original bisection algorithm in that whenever it partitions a set A into A_1 and A_2 then A_1 and A_2 are almost of the same size ($||A_1|-|A_2|| \leq 1$), whereas the original algorithm $|A_1| = 2^{\lceil \log |A| \rceil - 1}$ and $|A_2| = |A| - |A_1|$. Du and Hwang [6] showed that the bisection algorithm performs $O\big(k \log \frac{n}{k}\big)$ queries. Below we present a self-contained analysis.

Lemma 2.1. *Algorithm 1 makes at most $2k\big(\log_2 \frac{n}{k} + 2\big)$ oracle queries.*

Proof. We can model the execution of Algorithm 1 with a tree $\mathcal{T}$ whose nodes are the subsets A that have appeared in the queue $\mathcal{Q}$ during execution. A node A is a child of node B if and only if A was obtained by bisecting B. In particular

$\mathcal{T}$ is a binary tree with at most k leaves and two types of internal nodes: the *partition nodes* with two children correspond to splitting a set into two halves, and the *cut nodes* with one child correspond to cutting-off a half of a set. Each internal node in $\mathcal{T}$ is associated with 1 or 2 queries.

Let us order $\mathcal{T}$ arbitrarily so that every partition node has a left child and a right child; let us furthermore call the only child of a cut node the left child. For every leaf v form a path P_v up in the tree by first including v into the path and including each subsequent node into P_v as long as we arrived into the node from the left child of the node. Such paths P_v clearly form a partition of nodes in $\mathcal{T}$.

For every cut node x, let D_x denote the subset of vertices that was discarded. For a leaf v let S_v denote the union of all the sets D_x on path P_v. For any cut nodes x and y on P_v, if x is an ancestor of y then $|D_x| \geq 2|D_y| - 1$. It follows that there are at most $\lceil \log_2 |S_v| \rceil$ cut nodes on P_v. Hence the total number of cut nodes is at most $\sum_v \lceil \log_2 |S_v| \rceil \leq k\bigl(\log_2 \frac{n}{k} + 1\bigr)$ where the sum is over the at most k leaves v in $\mathcal{T}$ and the inequality follows from Jensen's inequality (and the fact that the sets S_v form a partition of $U \setminus W$, where W is the returned witness). Since $\mathcal{T}$ is a binary tree, the number of partition nodes is at most $k - 1$. Thus there are at most $k\bigl(\log_2 \frac{n}{k} + 2\bigr)$ nodes and at most $2k\bigl(\log_2 \frac{n}{k} + 2\bigr)$ queries. □

A routine information-theoretic argument shows that Lemma 2.1 is optimal up to constants, that is, at least $\log_2 \binom{n}{k} \geq k \log_2 \frac{n}{k}$ queries (bits of information) are needed to identify a unique witness of size k in a universe of size n. This observation can be strengthened to the randomized setting via the Yao principle—in expectation at least $\frac{k}{2} \log_2 \frac{n}{k}$ queries are required.

We now proceed to analyze Algorithm 1 with a more natural complexity measure, namely the total time of the extraction procedure, taking into account the time used by the oracle queries. Recall that a function $g : \mathbb{N} \to \mathbb{N}$ is at least linear if for all $n_1, n_2 \in \mathbb{N}$ we have $g(n_1) + g(n_2) \leq g(n_1 + n_2)$.

Lemma 2.2. *Suppose the time complexity of the inclusion oracle on a query set of size n is $T(n,k) = O(f(k)g(n))$, where g is at most linear. Then, the running time of Algorithm 1 is $O(k \cdot T(2n,k))$.*

Proof. We follow the notation introduced in the proof of Lemma 2.1. Because there are at most $k-1$ partition nodes, the total time spent at these nodes is $O(k \cdot T(n,k))$. Hence it remains to analyze the time spent at the cut nodes. It suffices to show that for every leaf v of the tree $\mathcal{T}$ the total time spent at the cut nodes in path P_v is $O(T(n,k))$. Observe that at every cut node the size of the universe decreases by a factor of 2. Hence this time is at most $T(n,k) + T(n/2,k) + T(n/4,k) + \ldots + T(1,k) \leq T(2n,k)$ where the last inequality uses the assumption that g is at most linear. □

Lemma 2.1 and Lemma 2.2 now establish Theorem 1.1.

3 Extracting a Witness Using a Randomized Oracle

The objective of this section is to prove Theorem 1.2. Accordingly, we assume we have available a randomized inclusion oracle that has no false positives but may output a false negative with probability at most $p \leq \frac{1}{4}$. The outcomes of queries are assumed to be mutually independent as random events.

We start with two simple observations regarding Algorithm 1 in the context of a randomized oracle. First, since the oracle does not have false positives, the set W output by Algorithm 1 is always a superset of a witness. Second, by Theorem 1.1 we know that the algorithm makes at most $Q(n,k)$ queries to extract a witness *in the event no false negatives occur in the first $Q(n,k)$ queries.* By the union bound, the probability of this event is at least $1 - pQ(n,k)$. This gives us a Monte Carlo algorithm that fails with probability at most $pQ(n,k)$.

Recalling that we assume we have access to a subroutine that checks whether a given set $W \subseteq U$ satisfies $W \in \mathcal{F}$, we would clearly like to transform the Monte Carlo algorithm into a Las Vegas algorithm that always extracts a witness, and the cost of randomization is only paid in terms of the running time.

The Las Vegas algorithm now operates in two stages. Let us call this algorithm Algorithm 2. In the first stage, we simply run Algorithm 1 and obtain a set W as output. In the second stage, we insert each element of W into an empty queue $\mathcal{Q}$. Next, as long as W is not a witness, we (1) remove an element e from the head of $\mathcal{Q}$, (2) if $\textsc{Includes}(W \setminus \{e\})$ returns NO then we insert e at the tail of $\mathcal{Q}$ and otherwise we remove e from W. Finally, we return W.

Given that only false negatives may occur, Algorithm 2 is obviously correct and always returns a witness. It remains to analyze the expected number of queries and the expected running time of Algorithm 2.

Lemma 3.1. *Algorithm 2 makes in expectation $O(k \log n)$ queries to the randomized inclusion oracle.*

Proof. First we bound the expected number of queries in the first stage. Recall the tree model of the execution of Algorithm 1 in the proof of Lemma 2.1. Let us study the model in the presence of false negatives. A false negative at line 10 of Algorithm 1 causes the algorithm to view the set A_1 as necessary and continue processing it even if it could in be dropped in reality. Similarly, a false negative at line 14 causes the algorithm to view A_2 as necessary. In particular, each false negative gets inserted into the queue $\mathcal{Q}$ and hence into the tree $\mathcal{T}$.

Now let us study an arbitrary subtree of $\mathcal{T}$ rooted at a false negative node. We observe that all such nodes either remain false negative nodes, or become exhausted as YES nodes or singleton nodes. (That is, no node in the subtree is a true negative.) Let us study the process that creates such a subtree and for convenience ignore the possibility of singleton nodes exhausting the process. Let X be the random variable that tracks the size of the subtree. Because the left and right child nodes of each node are independently false negatives with probability p, we observe that the expectation of X satisfies $\mathrm{E}[X] = 1 + 2p\mathrm{E}[X]$. That is, $\mathrm{E}[X] = 1/(1-2p)$. Because $p \leq \frac{1}{4}$, we have $\mathrm{E}[X] \leq 2$. Since each false negative has to interact with true negative and positive nodes, the expected number of queries in the first stage is, by linearity of expectation, at most $3Q(n,k)$ by Lemma 2.1.

Let W_0 denote W at the beginning of the second stage. For purposes of analysis we divide the second stage into two sub-stages. The first sub-stage finishes when $|W| \leq 2k$. Assume that there was at least one query in the first sub-stage, that is, $|W_0| > 2k$. Let Z be the total number of queries in the first sub-stage. Then $Z = Z_1 + Z_2 + Z_3$ where Z_1 is the number of false negative queries, Z_2 is the number of positive queries and Z_3 is the number of true negative queries. First observe that Z_1 has the negative binomial distribution, that is, $Z_1 \sim \mathrm{NB}(|W_0| - 2k, p)$, and hence $\mathrm{E}[Z_1 \mid |W_0|] = (|W_0| - 2k)\frac{p}{1-p} \leq |W_0| - 2k$. It follows that $\mathrm{E}[Z_1] \leq \mathrm{E}[|W_0|] - 2k \leq 3Q(n,k)$. Now note that that Z_2 is bounded by $|W_0|$, which is bounded by the number of queries in the first stage, so $\mathrm{E}[Z_2] \leq 3Q(n,k)$. Call an element e of W *false* if $W \setminus \{e\}$ contains a witness and *true* otherwise. Since there are at most k true elements, as long as $|W| > 2k$ the number of true elements is bounded by the number of false elements (if W contains more than one witness then all elements of W may be false). If $e \in W$ is a true element then the query $W \setminus \{e\}$ always returns NO (a true negative); if e is false then the query $W \setminus \{e\}$ may return either YES (a true positive) or NO (a false negative). Since elements of W are tested in queue order, $Z_3 \leq Z_1 + Z_2$ and hence $\mathrm{E}[Z_3] \leq 6Q(n,k)$.

Finally consider the second sub-stage, when $|W| \leq 2k$. Let t be the number of false elements in W, $t \leq 2k$. The algorithm iterates through the queue until there is no false element in W. The number of times we iterate over the whole queue is the maximum of t independent random variables, each of geometric distribution with success probability $1 - p$, which by $p \leq \frac{1}{4}$ is in expectation at most $1 + H_t/\ln(1/p) \leq 2H_{2k} \leq 3\ln 2k$ (cf. [8]). Since in each iteration the algorithm performs at most $2k$ queries, the expected number of queries in the second sub-stage is then at most $6k \ln 2k$.

The expected number of queries is thus at most $15Q(n,k) + 6k\ln 2k$. □

Theorem 1.2 is now established by Lemma 3.1 and the following lemma, whose proof is relegated to a full version of this work.

Lemma 3.2. *Suppose the time complexity of the randomized inclusion oracle on a query set of size n is $T(n,k) = O(f(k)g(n))$, where g is at most linear. Then, the running time of Algorithm 2 is $O(kT(2n,k) + k\log kT(2k,k))$.*

4 Implementation of Finite Field Arithmetic

The most critical subroutines of the k-path inclusion oracle we implemented are operations of addition and multiplication in a finite field $\mathrm{GF}(2^q)$. The choice of q is important: the oracle returns a false negative with probability at most $\frac{2k-1}{2^q}$. We can assume that $k \leq 30$, for otherwise the oracle runs too long. It follows that to guarantee low error probability, say, at most 2^{-20}, it suffices to pick $q = 26$.

Let us recall that elements of $\mathrm{GF}(2^q)$ correspond to polynomials of degree at most $q - 1$ with coefficients from $\mathrm{GF}(2)$. Such a polynomial is conveniently represented as a q-bit binary number. The addition in $\mathrm{GF}(2^q)$ corresponds to addition of two polynomials, that is, the symmetric difference (xor) of the binary representations. Multiplication is performed by (a) multiplying the polynomials and (b) returning the remainder of the division of the result by a primitive

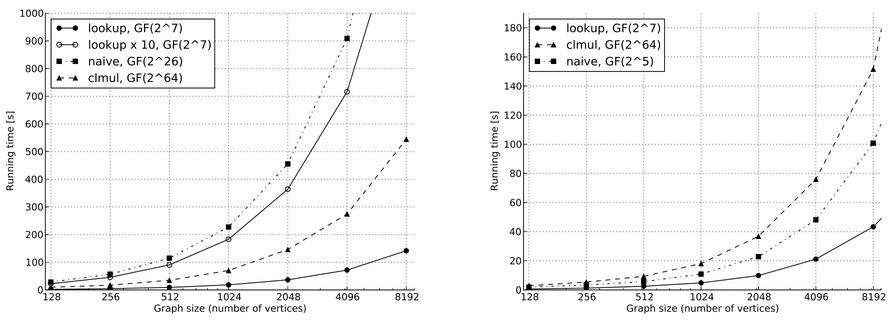


Fig. 2. Comparison of three implementations of GF(2^q) arithmetic. Left: (single run of) k-path decision oracle for instances with *no* solution. Right: fifo algorithm using k-path decision oracle for instances with exactly one solution (each running time on the graph is the median of 5 runs for the same input instance). The pattern size is fixed as $k = 16$ and the number of vertices n varies. Running times on a 2.53-GHz Intel Xeon CPU.

degree-q polynomial; this is easily implemented in $O(q)$ word operations. We refer to this implementation as 'naive'.

One can observe that step (a) above corresponds to carry-less multiplication of two binary numbers, that is, the usual multiplication without generating carries ($011 \times 011 = 101$). Such multiplication of two 64-bit numbers is available as a single instruction (PCLMULQDQ) on a number of modern Intel and AMD architectures. Using the fact that there is an only 5-term primitive polynomial of degree 64, step (b) can be implemented using bit shifts and xors [10]. We refer to this implementation as 'clmul'.

The third natural option is to *precompute* the whole multiplication table (using the naive algorithm) before running the oracle. This takes $4^q\lceil q/8\rceil$ bytes of memory, so can be considered only for small values of q, say $q \leq 12$ (even for $q = 12$ the precomputation time is negligible at substantially less than a second). We refer to this implementation as 'lookup'.

The left chart of Fig. 2 shows the comparison of the three implementations of GF(2^q) arithmetic used in a *single run* of the decision oracle. For 'naive' we use $q = 26$ and for 'clmul' $q = 64$. For 'lookup' we use $q = 7$ because for smaller values of q the running time is roughly the same; nevertheless since in the tests we look for a pattern of size 16, it gives just a bound of $\frac{1}{4}$ for error probability. To squeeze the probability down to 2^{-20} one can run the oracle 10 times and return the conjunction of the results. We see that although 'lookup' is faster than 'clmul' when the oracle is called once, it is much slower when we repeat the oracle call 10 times (note also that clmul provides error probability 2^{-59}). The 'naive' method is worse than the other two.

Note however that, if we aim at *finding* a witness, by Theorem 1.2 it suffices to guarantee that error probability is at most $\frac{1}{4}$, hence for $k \leq 16$ we can pick $q = 7$.

The advantage of our witness extraction algorithm fifo is that even if it gets a false answer from the oracle, it will discover the mistake in the future. Indeed, the right chart of Fig. 2 shows that using GF(2^7) with 'lookup' outperforms using GF(2^{64}) with 'clmul', roughly by a factor of four. The value $q = 7$ here is carefully chosen. One one hand, we want q to be large to get small error probability for a single query and thus small variance of the whole extraction running time. On the other hand, at our machine the multiplication table for $q = 8$ does not fit into L1 cache (of size 32K) what results in increase in the median running time. In the table below we show statistics for 200 runs of the extraction algorithm ($n = 1000$, $k = 12$) using 'lookup' for $q = 5, 6, \ldots, 12$ and 'clmul' ($q = 64$). Clearly, for $q = 8, 9, \ldots, 12$ we get increased number of cache misses (the 256K L2 cache could fit the table only for $q \leq 8$).

logarithm of the field size	5	6	7	8	9	10	11	12	64 (clmul)
median [sec]	4.58	4.38	4.39	4.69	6.15	7.30	9.55	15.94	15.92
maximum [sec]	12.96	8.53	7.61	7.57	9.97	11.18	9.65	18.05	16.77
standard deviation [sec]	1.16	0.68	0.61	0.22	0.30	0.34	0.50	0.43	0.25

References

1. Björklund, A., Husfeldt, T., Kaski, P., Koivisto, M.: Narrow sieves for parameterized paths and packings. arXiv 1007.1161 (2010)
2. Björklund, A., Kaski, P., Kowalik, Ł.: Probably optimal graph motifs. In: Proc. STACS 2013, pp. 20–31 (2013)
3. Chen, J., Kneis, J., Lu, S., Mölle, D., Richter, S., Rossmanith, P., Sze, S.H., Zhang, F.: Randomized divide-and-conquer: Improved path, matching, and packing algorithms. SIAM Journal on Computing 38(6), 2526–2547 (2009)
4. Cygan, M., Fomin, F.V., Kowalik, L., Lokshtanov, D., Marx, D., Pilipczuk, M., Pilipczuk, M., Saurabh, S.: Parameterized Algorithms. Springer (to appear)
5. Cygan, M., Kratsch, S., Nederlof, J.: Fast hamiltonicity checking via bases of perfect matchings. In: Proc. STOC 2013, pp. 301–310. ACM (2013)
6. Du, D.Z., Hwang, F.K.: Competitive group testing. Discrete Appl. Math. 45(3), 221–232 (1993)
7. Du, D.Z., Hwang, F.K.: Combinatorial Group Testing and Its Applications. Series on Applied Mathematics, vol. 12. World Scientific Publishing Co. Inc. (2000)
8. Eisenberg, B.: On the expectation of the maximum of IID geometric random variables. Statistics & Probability Letters 78(2), 135 (2008)
9. Fellows, M.R., Langston, M.A.: On search decision and the efficiency of polynomial-time algorithms. In: Proc. STOC 1989, pp. 501–512. ACM (1989)
10. Gueron, S., Kounavis, M.: Efficient implementation of the Galois Counter Mode using a carry-less multiplier and a fast reduction algorithm. Information Processing Letters 110(14), 549–553 (2010)
11. Hassidim, A., Keller, O., Lewenstein, M., Roditty, L.: Finding the minimum-weight k-path. In: Dehne, F., Solis-Oba, R., Sack, J.-R. (eds.) WADS 2013. LNCS, vol. 8037, pp. 390–401. Springer, Heidelberg (2013)
12. Koutis, I.: Constrained multilinear detection for faster functional motif discovery. Inform. Process. Lett. 112(22), 889–892 (2012)
13. Williams, R.: Finding paths of length k in $O^*(2^k)$ time. Inform. Process. Lett. 109(6), 315–318 (2009)

Complexity of Higher-Degree Orthogonal Graph Embedding in the Kandinsky Model[⋆]

Thomas Bläsius[1], Guido Brückner[1], and Ignaz Rutter[1,2]

[1] Faculty of Informatics, Karlsruhe Institute of Technology, Karlsruhe
[2] Department of Applied Mathematics, Charles University, Prague

Abstract. We show that finding orthogonal grid embeddings of *plane graphs* (planar with fixed combinatorial embedding) with the minimum number of bends in the so-called Kandinsky model (allowing vertices of degree > 4) is NP-complete, thus solving a long-standing open problem. On the positive side, we give an efficient algorithm for several restricted variants, such as graphs of bounded branch width and a subexponential exact algorithm for general plane graphs.

1 Introduction

Orthogonal grid embeddings are a fundamental topic in computer science and the problem of finding suitable grid embeddings of planar graphs is a subproblem in many applications, such as graph visualization [19] and VLSI design [17,21]. Aside from the area requirement, the typical optimization goal is to minimize the number of bends on the edges (which heuristically minimizes the area). Traditionally, grid embeddings have been studied for *4-planar graph* (max-deg 4), which is natural since it allows to represent vertices by grid points and edges by internally disjoint chains of horizontal and vertical segments on the grid. For plane graphs, Tamassia showed that the number of bends can be efficiently minimized [14]; the running time was recently reduced to $O(n^{1.5})$ [7]. In contrast, if the combinatorial embedding is not fixed, it is NP-complete to decide whether a 0-embedding (a k*-embedding* is a planar grid embedding with at most k bends per edge) exists [14], thus also showing that bend minimization is NP-complete and hard to approximate. In contrast, a 2-embedding exists for every graph except the octahedron [2]. Recently it was shown that the existence of a 1-embedding can be tested efficiently [4]. The problem is FPT if some subset of k edges has to have 0 bends [5]. If there are no 0-bend edges, it is even possible to minimize the number of bends in the embedding, not counting the first bend on each edge [6].

These results only apply to graphs of maximum degree 4. There have been several suggestions for possible generalizations to allow vertices of higher degree [16,20]. For example, it is possible to represent higher-degree vertices by rectangles. The disadvantage is that the vertices may be stretched arbitrarily in order to avoid bends. In particular, a visibility representation (existing for every planar graph) can be interpreted as a 0-embedding in this model. It is thus natural to forbid stretching of vertices.

⋆ Partially supported by grant WA 654/21-1 of the German Research Foundation (DFG). Partly done within GRADR – EUROGIGA project no. 10-EuroGIGA-OP-003. Ignaz Rutter was supported by a fellowship within the Postdoc-Program of the German Academic Exchange Service (DAAD).

Fößmeier and Kaufmann [13] proposed a generalization of planar orthogonal grid embeddings, the so-called Kandinsky model (originally called podevsnef), that overcomes this problem and guarantees that vertices are represented by boxes of uniform size. Essentially their model allows to map vertices to grid points on a coarse grid, while routing the edges on a much finer grid. The vertices are then interpreted as boxes on the finer grid, thus allowing several edges to emanate from the same side of a vertex; see Sect. 2. Fößmeier and Kaufmann model the bend minimization in the fixed combinatorial embedding setting by a flow network similar to the work of Tamassia [18] but with additional constraints that limit the total amount of flow on some pairs of edges. Fößmeier et al. [12] show that every planar graph admits a 1-embedding in this model. Concerning bend minimization, reductions of the mentioned flow networks to ordinary minimum cost flows have been claimed both for general bend minimization [13] and for bend minimization when every edge may have at most one bend [12].

Eiglsperger [10] pointed out that the reductions to minimum cost flow are flawed and gave an efficient 2-approximation. Bertolazzi et al. [1] introduced a restricted variant of the Kandinsky model (requiring more bends), for which bend minimization can be done in polynomial time. Although the Kandinsky model has been later vastly generalized, e.g., to apply to the layout of UML class diagrams [11], the fundamental question about the complexity of bend minimization in the Kandinsky model has remained open for almost two decades.

Contribution and Outline. We show that the bend minimization problem in the Kandinsky model is NP-complete (no matter if we allow or forbid so-called empty faces). This also holds if each edge may have at most one bend; see Sect. 3. As an intermediate step, we show NP-hardness of the problem ORTHOGONAL 01-EMBEDDABILITY, which asks whether a plane graph (with maximum degree 4) admits a grid embedding when requiring some edges to have exactly one and the remaining edges to have zero bends. This result is interesting on its own, as it can serve as tool to show hardness of other grid embedding problems. In particular, it gives a simpler proof for the hardness of deciding 0-embeddability (maximum degree 4) for graphs with a variable embedding.

We then study the complexity of the problem subject to structural graph parameters in Sect. 4. For graphs with branch width k, we obtain an algorithm with running time $2^{O(k \log n)}$. For fixed branch width this yields a polynomial-time algorithm ($O(n^3)$ for series-parallel graphs), for general plane graphs the result is an exact algorithm with subexponential running time $2^{O(\sqrt{n} \log n)}$.

For detailed proofs, we refer to the full version of this paper [3].

2 Preliminaries

Kandinsky Embedding. Let G be a plane graph. An *orthogonal embedding* of G maps vertices to grid points and edges to paths in the grid such that the resulting drawing is planar and respects the combinatorial embedding of G; see Fig. 1a. Clearly, G admits an orthogonal embedding if and only if it is 4-planar. The Kandinsky model [13] overcomes this limitation. A *Kandinsky embedding* of G (Fig. 1b) maps each vertex to a box of constant size centered at a grid point and each edge to a path in a finer grid such that

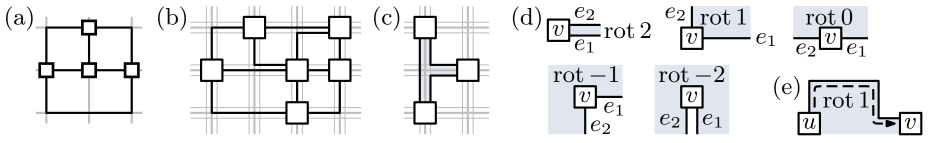


Fig. 1. (a) An orthogonal embedding of the K_4. (b) A Kandinsky embedding of the wheel of size 5. (c) A Kandinsky embedding with an empty face. (d–e) The rotation of a vertex (d) and an edge (e) in a face f (shaded blue).

the resulting drawing is planar, respects the combinatorial embedding of G, and has no *empty faces*. A face is empty if it does not include a grid cell of the coarser grid; see Fig. 1c.

One can declare a bend on an edge uv to be *close* to v if it is the last bend on uv (traversing uv from u to v). A bend cannot simultaneously be close to u and to v. Kandinsky embeddings have the *bend-or-end property* [13], requiring that a 0° angle between edges uv and vw in the face f implies that at least one of the edges uv and vw has a bend close to v forming a 270° angle in f.

Kandinsky Representation. A Kandinsky embedding of a planar graph can be specified in three stages. First, its topology is fixed by choosing a combinatorial embedding. Second, its shape in terms of angles between edges and sequences of bends on edges is fixed. Third, the geometry is fixed by specifying coordinates for vertices and bend points. In analogy to combinatorial embeddings as equivalence classes of planar drawings with the same topology, one can define *Kandinsky representations* as equivalence classes of Kandinsky embeddings with the same topology and the same shape. This approach was first introduced for orthogonal embeddings [18] and extended to Kandinsky embeddings [13].

Let Γ be a Kandinsky embedding. Let f be a face with an edge e_1 in its boundary and let e_2 be the successor of e_1 in clockwise direction (counter-clockwise if f is the outer face). Let further v be the vertex between e_1 and e_2 and let α be the angle at v in f. We define the *rotation* $\mathrm{rot}_f(e_1, e_2)$ between e_1 and e_2 to be $\mathrm{rot}_f(e_1, e_2) = 2 - \alpha/90°$; see Fig. 1d. The rotation $\mathrm{rot}_f(e_1, e_2)$ can be interpreted as the number of right turns between the edges e_1 and e_2 at the vertex v in the face f. We also write $\mathrm{rot}_f(v)$ instead of $\mathrm{rot}_f(e_1, e_2)$ if the edges are clear from the context and call it the *rotation of* v *in* f.

The shape of every edge can also be described in terms of its rotation. Let $e = uv$ be an edge incident to a face f such that v is the clockwise successor of u along the boundary of f (counter-clockwise if f is the outer face). The *rotation* $\mathrm{rot}_f(e)$ of e in f is the number of right bends minus the number of left bends one encounters, when traversing e from u to v; see Fig. 1e.

Let uv, vw be a path of length 2 in the face f. If uv and vw form an angle of 0° ($\mathrm{rot}_f(v) = 2$), at least one of the edges uv or vw has a bend close to v with rotation -1 in f (bend-or-end property). We represent the information of which bends are close to vertices as follows. If uv has a bend close to v, we define the *rotation* $\mathrm{rot}_f(uv[v])$ *at the end* v of uv to be 1 (-1) if it has rotation 1 (-1) in f. If uv has no bend close to v, we set $\mathrm{rot}_f(uv[v]) = 0$.

A set of values for the rotations is a Kandinsky representation (i.e., there is a corresponding embedding) if and only if it satisfies the following properties [13].

(1) The sum over all rotations in a face is 4 (-4 for the outer face).
(2) For every edge uv with incident faces f_ℓ and f_r, we have $\mathrm{rot}_{f_\ell}(uv) + \mathrm{rot}_{f_r}(uv) = 0$, $\mathrm{rot}_{f_\ell}(uv[u]) + \mathrm{rot}_{f_r}(uv[u]) = 0$, and $\mathrm{rot}_{f_\ell}(uv[v]) + \mathrm{rot}_{f_r}(uv[v]) = 0$.
(3) The sum of rotations around a vertex v is $2 \cdot \deg(v) - 4$.
(4) The rotations at vertices lie in the range $[-2, 2]$.
(5) If $\mathrm{rot}_f(uv, vw) = 2$ then $\mathrm{rot}_f(uv[v]) = -1$ or $\mathrm{rot}_f(vw[v]) = -1$.

If the face is clear from the context, we often omit the subscript in rot_f. One can assume that all bends on an edge (except for bends close to vertices) have the same direction. It follows that the actual number of bends of uv can be computed from $\mathrm{rot}(uv)$, $\mathrm{rot}(uv[u])$, and $\mathrm{rot}(uv[v])$.

Let f be a face of G and let u and v be two vertices on the boundary of f. By $\pi_f(u, v)$ we denote the path from u to v on the boundary of f in clockwise direction (counter-clockwise for the outer face). The rotation $\mathrm{rot}_f(\pi)$ of a path π in the face f is the sum of all rotations of edges and inner vertices of π in f.

An orthogonal embedding is basically a Kandinsky embedding without $0°$ angles at vertices. Thus, we can define *orthogonal representations* [18] (equivalence class of orthogonal embeddings), by forbidding rotation 2 at vertices.

3 Complexity

Let $\mathcal{S}$ be an instance of 3-SAT. In its *variable-clause graph*, the variables and clauses are represented by vertices and there is an edge xc connecting a variable x with a clause c if and only if $x \in c$ or $\neg x \in c$. The NP-hard problem PLANAR MONOTONE 3-SAT [8] restricts the instances of 3-SAT as follows. Every clause contains only positive or only negative literals. Moreover, the variable-clause graph admits a planar embedding such that the edges connecting a variable x to its positive clauses appear consecutively around x.

The problem ORTHOGONAL 01-EMBEDDABILITY is defined as follows. Let $G = (V, E)$ be a 4-plane graph having its edges $E = E_0 \dot\cup E_1$ partitioned into 0-*edges* (E_0) and 1-*edges* (E_1). Decide whether G admits an *orthogonal* 01-*representation* such that every edge in E_i has exactly i bends. In the following, we always consider the variant of ORTHOGONAL 01-EMBEDDABILITY where we allow to fix angles at vertices. Fixing the angles at vertices does not make the problem harder since augmenting a vertex v to have degree 4 by adding degree-1 vertices incident to v has the same effect as fixing the angles at v.

We first reduce PLANAR MONOTONE 3-SAT to ORTHOGONAL 01-EMBEDDABILITY, which is further reduced to KANDINSKY BEND MINIMIZATION.

3.1 Orthogonal 01-Embeddability

In the reduction from PLANAR MONOTONE 3-SAT, the decision of setting a variable to `true` or `false` is encoded in the bend-direction of a 1-edge. We show how to build gadgets for variables (outputting a positive and negative literal) and for clauses

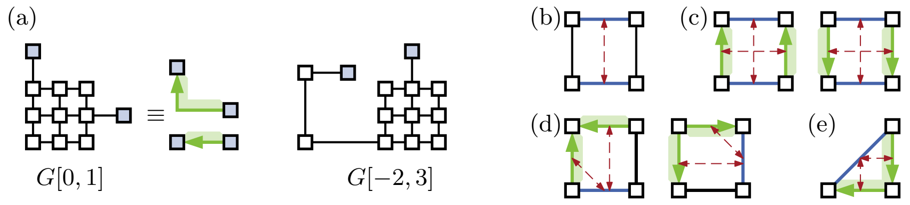


Fig. 2. (a) The interval gadgets $G[0,1]$ ($\equiv$ 01-edge) and $G[-2,3]$ (s and t are blue). (b–e) Edges are color-coded; 0-edges are black; 1-edges are blue; 01-edges are green and directed such that they may bend right but not left. The building blocks are (b) the box; (c) the bendable box; (d) the merger; (e) the splitter.

(admitting drawings if and only if at least one input edge encodes the value `true`). To carry the decision of one variable to several clauses we need gadgets that impose the bend direction from one edge on multiple edges (literal duplicator). Finally, we build bendable pipes to carry the information (in a flexible way) to the clause gadgets. We first present some basic building blocks.

Building Blocks. An interval gadget $G[\rho_1, \rho_2]$ is a graph with two designated degree-1 vertices (its *endpoints*) s and t on the outer face. It has the property that $\mathrm{rot}(\pi(s,t)) \in [\rho_1, \rho_2]$ for any orthogonal embedding. The construction is similar to the tendrils used by Garg and Tamassia [14]; Fig. 2a shows $G[0,1]$ and $G[-2,3]$. Note that $G[0,1]$ behaves like an edge that may have one bend, but only into a fixed direction. In the following, we draw copies of $G[0,1]$ as directed green edges and refer to them as 01-*edges*.

All our gadgets are based on the building blocks shown in Fig. 2b–e. We require that the angles at the vertices in the internal face f are 90° (rotation 1). Note that, apart from the 0-edges, all edges of the building blocks admit precisely two possible rotation values in each face. Thus, each edge attains its maximum rotation value in one of its faces and the minimum rotation in the other. It can be shown that in any orthogonal 01-representation the rotation values of some edge pairs are not independent but are linked in the sense that exactly one of them must attain its minimum (maximum) rotation value in f. In Fig. 2b–e such dependencies are displayed as red dashed arrows.

Gadget Constructions. Our gadgets will always have 1-edges on the outer face, whose bend directions represent truth values (as output or as input). We again use red dashed arrows to indicate which edges have to bend consistently. It follows that when there is a path of such red arrows from one edge to another edge, then they are synchronized.

The *variable gadget* for a variable x consists of a single box. The two 1-bend edges are called *positive* and *negative output*. The variable gadget has exactly two different representations; see Fig. 3a. We interpret a rotation of -1 and 1 of the positive output in the outer face as $x =$ `true` and $x =$ `false`, respectively.

The *literal duplicator* is formed by a splitter, which is glued to two mergers via its 01-edges; see Fig. 3b. It has one *input edge* and two *output edges* and transfers the state of the input to both outputs in every orthogonal 01-representation (red dashed paths),

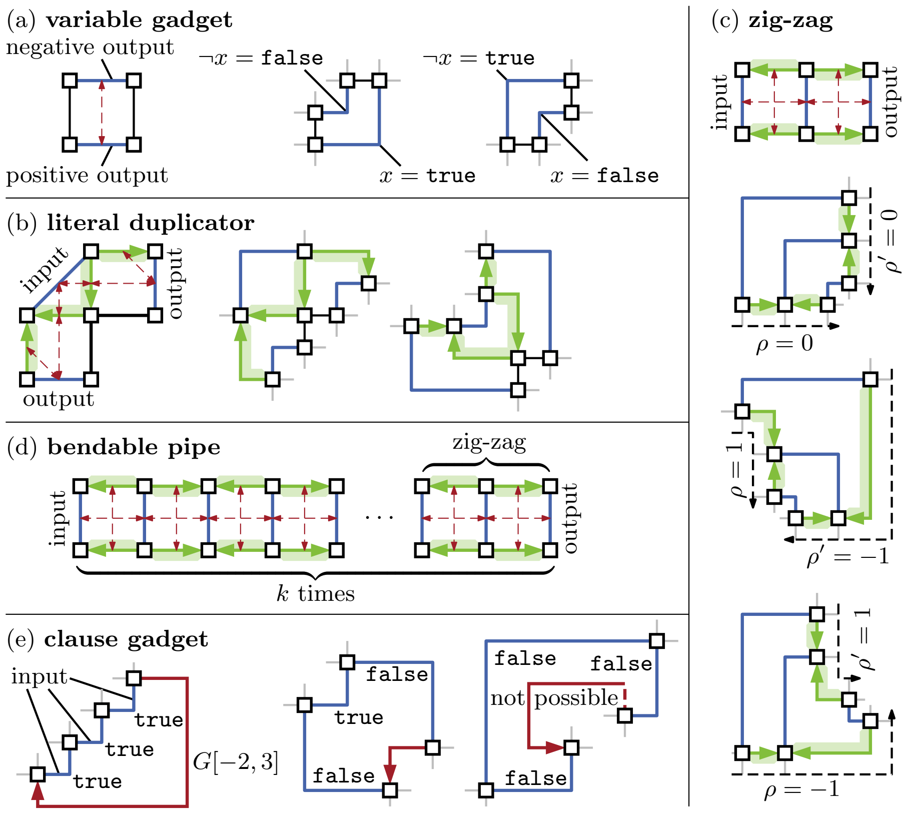


Fig. 3. The different gadgets we use in our construction

i.e., the output edges have rotation -1 (1) in the outer face if and only if the input edge has rotation 1 (-1). The literal duplicator admits orthogonal 01-representations for both inputs `true` and `false`; see Fig. 3b.

A *zig-zag* consists of the two bendable boxes glued along a pair of 1-edges; see Fig. 3c. It has an input and an output edge, and in any valid drawing the information encoded in the input is transmitted to the output. Moreover, the decision which of the bendable boxes bend their 01-edges can be taken independently. Thus, the zig-zag allows to choose the rotations ρ, ρ' of the paths between the input and the output edge with $\rho = -\rho'$ for each $\rho \in \{-1, 0, 1\}$; see the drawings in Fig. 3c. A *k-bendable pipe* is obtained by concatenating k zig-zags; see Fig. 3d. It has the same properties as a zig-zag, except that the rotation ρ can be in the interval $[-k, k]$. In a high-level view, a bendable pipe looks like a flexible edge that transfers information between its endpoints.

The *clause gadget* is a cycle of length 4, consisting of three 1-edges, the *input edges*, and the interval gadget $G[-2, 3]$; see Fig. 3e. The inner face lies to the right of $G[-2, 3]$ (i.e., the rotation of $G[-2, 3]$ in the inner face is in $[-2, 3]$) and the angles at vertices in inner faces are fixed to $90°$. Interpreting a rotation of -1 (of 1) of an input edge in the inner face as `true` (as `false`), we get a valid embedding if and only if not all inputs are `false`; see Fig. 3e.

Putting Things Together. Let S be an instance of PLANAR MONOTONE 3-SAT. To obtain the graph $G(S)$, we create a variable gadget for every variable and a clause gadget for every clause, duplicate the literals (using the literal duplicator) outputted by the variable gadget as often as they occur in clauses, and connect the resulting output edges with the corresponding input edges of the clauses using bendable pipes of sufficient length. Note that $G(S)$ is planar if we adhere to the planar embedding of the variable-clause graph of S.

If $G(S)$ admits an orthogonal 01-representation, the drawings of the variable gadgets imply a truth assignment for the variables in S. Moreover, it satisfies S, since a non-satisfied clause would imply an orthogonal 01-representation of a clause gadget with value `false` on every input edge. Conversely, a satisfying truth assignment of S, completely fixes the orthogonal 01-representation of each gadget, except for the rotations along the bendable pipes. One needs to show that these representations can be plugged together to a representation of the whole graph $G(S)$, which is the case if the bendable pipes are sufficiently long.

Theorem 1. ORTHOGONAL 01-EMBEDDABILITY *is NP-complete.*

In fact, we even showed NP-hardness for the case where all angles at vertices incident to 1-edges are fixed. Moreover, it can be seen that both variants remain hard if the combinatorial embedding is fixed up to the choice of an outer face.

It can be shown that ORTHOGONAL 01-EMBEDDABILITY remains NP-hard for subdivisions of triconnected graphs [3], which have a unique combinatorial embedding. Replacing in such an instance every 1-edge with a copy of the interval gadget $G[1,1]$ and releasing the combinatorial embedding gives an equivalent instance of 0-EMBEDDABILITY (variable embedding) where mirroring the embedding of $G[1,1]$ corresponds to bending a 1-edge in different directions. This simplifies the hardness proof by Garg and Tamassia [14].

3.2 Kandinsky Bend Minimization

The reduction from ORTHOGONAL 01-EMBEDDABILITY to KANDINSKY BEND MINIMIZATION consists of two basic building blocks. In an orthogonal embedding, $0°$ angles between edges are forbidden. We show how to enforce this for Kandinsky embeddings. Moreover, we construct a subgraph whose Kandinsky embeddings behave like the embeddings of an edge with exactly one bend.

The graph B in Fig. 4a is called *corner blocker*. The vertex v is its *attachment vertex*. Clearly, B admits a Kandinsky representation with two bends. It can be shown that there is no representation with fewer bends and that three bends are necessary if the angle at v is $0°$.

Let v be a vertex with incident edges e_1 and e_2. Assume we attach two corner blockers B_1 and B_2 and embed them as in Fig. 4b. Then the angle between e_1 and e_2 cannot be $0°$ without causing B_1 or B_2 to have three bends. By *nesting* corner blockers

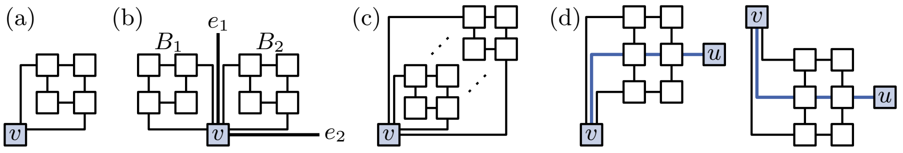


Fig. 4. (a) The corner blocker. (b) Two corner blockers enforcing at least 90° angles between e_1 and e_2 (c) Nesting corner blockers. (d) The one-bend gadget.

(Fig. 4c), one can increase this cost arbitrarily. Hence, we can force angles between edges to be at least 90° by adding (nested) corner blockers.

The graph Γ in Fig. 4d with the two *endvertices* u and v is called *one-bend gadget*. The path π from u to v (blue in Fig. 4d) is the *bending path* of Γ. A Kandinsky representation of Γ *blocks no corner* if all three edges incident to v leave v on the same side. Clearly, Γ admits Kandinsky representations blocking no corner with three bends and rotation 1 and -1 on π. We can show that an optimal representation of Γ blocking no corner requires three bends and rotation either -1 or 1 on π. Thus, Γ behaves like a 1-edge.

Let $G = (V, E = E_0 \dot{\cup} E_1)$ (with combinatorial embedding) be an instance of ORTHOGONAL 01-EMBEDDABILITY. We assume that all angles at vertices incident to a 1-edge in G are fixed. Starting with G, we construct a graph G' that serves as instance of KANDINSKY BEND MINIMIZATION. Let v be a vertex of G. If the angles at v are not fixed, we add a nested corner blocker for every face incident to v, which forbids 0° angles between edges of G incident to v. If the angles are fixed, we add $\alpha/90°$ nested corner blockers into a face with angle α, which enforces the correct angles. Then we replace every 1-edge uv in G with a one-bend gadget Γ. As the angles around v were fixed (v is incident to a 1-edge), Γ is forced to block no corner. Hence, Γ has at least three bends in every Kandinsky representation of G' and its bending path has rotation 1 or -1.

We show that G admits an orthogonal 01-representation if and only if G' has a Kandinsky representation with $2b + 3|E_1|$ bends (b is the number of corner blockers). Given an orthogonal 01-representation of G, one can add the corner blockers (two bends each) and replace 1-edges by one-bend gadgets (three bends each). Conversely, given a Kandinsky representation of G' with $2b + 3|E_1|$ bends, removing the corner blockers and replacing the one-bend gadgets by edges with one bend gives an orthogonal 01-representation. The construction still works when allowing empty faces or restricting edges to have at most one bend (or both).

Theorem 2. KANDINSKY BEND MINIMIZATION *is NP-complete.*

4 A Subexponential Algorithm

In this section, we give a subexponential algorithm for computing optimal Kandinsky representations of plane graphs. We use dynamic programming on sphere cut decompositions, which are special types of branch decompositions [9]. Assume graph G is decomposed into subgraphs G_1 and G_2. It may be possible to merge Kandinsky representations $\mathcal{K}_1$ and $\mathcal{K}_2$ of G_1 and G_2 into a representation of G. We show (Sect. 4.1) which properties of $\mathcal{K}_1$ are important when trying to merge it with $\mathcal{K}_2$ and derive classes of Kandinsky representations whose members behave equivalently. If we know optimal Kandinsky representations of G_1 and G_2 for each of these equivalence classes, we find an optimal representation of G by trying to merge every pair of representations of G_1 and G_2. We bound the number of combinations one has to consider in Sect. 4.2. Iteratively applying this merging step in a sphere cut decomposition results in our Algorithm (Sect. 4.3).

4.1 Interfaces of Kandinsky Representations

Consider two edge-disjoint graphs G_1 and G_2 sharing a set of *attachment vertices*. Let the union G of G_1 and G_2 be plane. We say that G_1 and G_2 are *glueable* if both graphs are connected and there is a simple closed curve that separates G_1 from G_2; see Fig. 5a–c. We also say that G_1 (G_2) is a *glueable subgraph* of G. A sphere cut decomposition of width k basically recursively decomposes a plane graph into glueable subgraphs with at most k attachment vertices.

Let $\mathcal{K}$ be a Kandinsky representation of G with restriction $\mathcal{K}_1$ to G_1. Let $\mathcal{K}_1'$ be another representation of G_1. *Replacing* $\mathcal{K}_1$ *with* $\mathcal{K}_1'$ *in* $\mathcal{K}$ means to set every rotation in $\mathcal{K}$ involving only edges in G_1 to its value in $\mathcal{K}_1'$ (other values remain unchanged). The result is not necessarily a Kandinsky representation. We say that $\mathcal{K}_1$ and $\mathcal{K}_1'$ have *the same interface* if replacing $\mathcal{K}_1$ with $\mathcal{K}_1'$ (and vice versa) in any Kandinsky representation of G yields a Kandinsky representation of G; see Fig. 5d. In the following we derive a combinatorial description of an interface.

Consider two glueable subgraphs G_1 and G_2 of a plane graph G. Let $v_0, \dots, v_\ell$ be the attachment vertices in the order they appear on the simple closed curve separating G_1 from G_2. Let f be the face of G_1 containing G_2 and let C_f be its facial cycle (C_f contains $v_0, \dots, v_\ell$ in that order). The v_i decompose C_f into the *interface paths* $\pi_{01}, \pi_{12}, \dots, \pi_{\ell 0}$ with $\pi_{ij} = \pi(v_i, v_j)$. For an attachment vertex v_i, denote the last edge of the path $\pi_{i-1\,i}$ by e_i^{in} and the first edge of the path $\pi_{i\,i+1}$ by e_i^{out} (indices are considered modulo $\ell + 1$); see Fig. 5e.

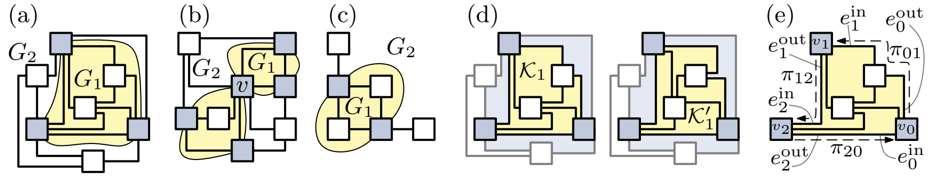


Fig. 5. (a) Decomposition of a graph into glueable subgraphs G_1 and G_2 (attachment vertices are shaded blue). (b) A non-glueable decomposition (a closed curve separating G_1 from G_2 cannot be simple as v must be visited twice). (c) Non-glueable decomposition (G_2 is disconnected). (d) Graph G with glueable subgraph G_1 (yellow). Faces shared by G_1 and G_2 are blue. The Kandinsky representations $\mathcal{K}_1$ and $\mathcal{K}_1'$ are interchangeable. (e) Some notation.

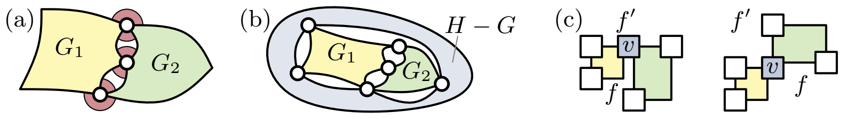


Fig. 6. (a) Merging G_1 and G_2. Shared rotations are marked red. (b) A merging step of width 5. (c) Two ways to choose the shared rotations.

The representations $\mathcal{K}_1$ and $\mathcal{K}_1'$ of G_1 have *compatible interface paths* if each $\pi_{i\,i+1}$ has the same rotation in $\mathcal{K}_1$ and $\mathcal{K}_1'$. They have *the same attachment rotations* if for every attachment vertex v_i, the rotation $\mathrm{rot}(e_i^{\mathrm{in}}, e_i^{\mathrm{out}})$ is the same. In Fig. 5e, interface paths π_{01}, π_{12}, and π_{20} have rotations -1, 1, and 0, and the attachment rotations at v_0, v_1, and v_2 are -1, -1, and -2, respectively.

For an attachment vertex v_i, the rotations at the end v_i of the edges e_i^{in} and e_i^{out} ($\mathrm{rot}(e_i^{\mathrm{in}}[v_i])$ and $\mathrm{rot}(e_i^{\mathrm{out}}[v_i])$) indicate whether $0°$ angles at v_i are allowed. For both rotations, we define the $0°$ *flag* to be `true` if a $0°$ angle is allowed (rotation -1) and `false` otherwise (rotations $0, 1$). Possible values for the $0°$ flags in Fig. 5e are `true` for $e_0^{\mathrm{out}}[v_0]$ and for $e_1^{\mathrm{in}}[v_1]$ and `false` for all other flags.

Lemma 1. *Two Kandinsky representations have the same interface iff they have compatible interface paths, the same attachment rotations, and the same* $0°$ *flags.*

It follows that each interface class is uniquely described by this information. We simply call it the *interface* of G_1 (G_2) in G.

4.2 Merging Two Kandinsky Representations

Let $\mathcal{K}_1$ and $\mathcal{K}_2$ be Kandinsky representations of G_1 and G_2, respectively, and let $G = G_1 \cup G_2$. We say that $\mathcal{K}_1$ and $\mathcal{K}_2$ can be *merged* if there exists a Kandinsky representation $\mathcal{K}$ of G whose restriction to G_1 and G_2 is $\mathcal{K}_1$ and $\mathcal{K}_2$, respectively. Note that the only rotations in $\mathcal{K}$ that occur neither in $\mathcal{K}_1$ nor in $\mathcal{K}_2$ are rotations at attachment vertices between an edge of G_1 and an edge of G_2. We call these rotations the *shared rotations*; see Fig. 6a. Thus, merging $\mathcal{K}_1$ and $\mathcal{K}_2$ is the process of choosing values for the shared rotation such that the resulting set of rotations is a Kandinsky representation of G.

In the following, we consider the case where G itself is a glueable subgraph of a larger graph H. We call this the *merging step* $G = G_1 \sqcup G_2$. Note that G_1 and G_2 are also glueable subgraphs of H. Note further that the interface of G_1 (G_2) in G can be deduced from the interface of G_1 (G_2) in H. When dealing with a merging step, we always consider the interfaces of G_1 and G_2 in H. The *width* of a merging step is the maximum number of attachment vertices of G_1, G_2, and G in H; see Fig. 6b for an example.

If the Kandinsky representations $\mathcal{K}_1$ and $\mathcal{K}_2$ can be merged, then every Kandinsky representation $\mathcal{K}_1'$ with the same interface as $\mathcal{K}_1$ can be merged in the same way (i.e., with the same shared rotations) with $\mathcal{K}_2$. Moreover, the resulting Kandinsky representations $\mathcal{K}$ and $\mathcal{K}'$ of G have the same interface. Thus, the only choices that matter when merging two representations are to choose shared rotations and interfaces for G_1 and G_2. A choice of shared rotations and interfaces is *compatible* if these interfaces can be merged using the chosen rotations.

We bound the number of compatible combinations, depending on the width k of the merging step and the *maximum rotation* ρ. The maximum rotation of a graph H is ρ if H admits an optimal Kandinsky representation such that the absolute rotations of the interface paths in every glueable subgraph of H are at most ρ; the maximum rotation ρ of a merging step refers to the maximum rotation of the whole graph H.

A simple bound can be obtained as follows. There are $2k$ interface paths, each admitting up to $(2\rho+1)$ possible rotations, giving $(2\rho+1)^{2k}$ combinations. Every attachment vertex has its attachment rotation in $[-2, 2]$ and two binary $0°$-flags, yielding another 20^{2k} combinations. Finally, each shared rotation (two per attachment) may be chosen from $[-2, 2]$, yielding again 5^{2k} combinations. That are $(2\rho + 1)^{2k}10000^k$ combinations in total. By a careful consideration which combinations are actually meaningful this number can be reduced greatly.

Lemma 2. *In a merging step $G = G_1 \sqcup G_2$ of width k with maximum rotation ρ, there are at most $(2\rho + 1)^{\lfloor 1.5k \rfloor -1} \cdot 330^k$ compatible choices for the shared rotations and the interfaces of G_1 and G_2.*

Let G be a glueable subgraph of H. The *cost* of an interface class is the minimum cost (e.g., number of bends) of the Kandinsky representations it contains. The *cost table* of G is a table containing the cost of each interface class of G.

Lemma 3. *Let $G = G_1 \sqcup G_2$ be a merging step of width k with maximum rotation ρ. Given the cost tables of G_1 and G_2, the cost table of G can be computed in $O(k \cdot (2\rho + 1)^{\lfloor 1.5k \rfloor -1} \cdot 330^k)$ time.*

4.3 The Algorithm

The previous three lemmas together with an optimal sphere cut decomposition (computable in $O(n^3)$ time [15,9]) can be used to prove the following theorem.

Theorem 3. *An optimal Kandinsky representation of a plane graph G can be computed in $O(n^3 + n \cdot k \cdot (2\rho + 1)^{\lfloor 1.5k \rfloor -1} \cdot 330^k)$ time, where k is the branch width and ρ the maximum rotation of G.*

To obtain the following corollaries, we bound ρ in terms of the optimal bend number and the maximum face size and use upper bounds of 2 and $O(\sqrt{n})$ on the branch width of series-parallel and planar graphs, respectively.

Corollary 1. *Let G be a plane graph with maximum face-degree Δ_F, and branch width k. An optimal Kandinsky representation can be computed in $O(n^3 + n \cdot k \cdot (2m + 2\Delta_F - 3)^{\lfloor 1.5k \rfloor -1} \cdot 330^k)$ time. An optimal b-bend Kandinsky representation can be computed in $O(n^3 + n \cdot k \cdot ((2b + 2) \cdot \Delta_F - 2b - 3)^{\lfloor 1.5k \rfloor -1} \cdot 330^k)$ time.*

Corollary 2. *For series-parallel and general plane graphs an optimal Kandinsky representation can be computed in $O(n^3)$ and $2^{O(\sqrt{n} \log n)}$ time, respectively.*

Acknowledgments. We thank Therese Biedl for discussions.

References

1. Bertolazzi, P., Di Battista, G., Didimo, W.: Computing orthogonal drawings with the minimum number of bends. IEEE Trans. Comput. 49(8), 826–840 (2000)
2. Biedl, T., Kant, G.: A better heuristic for orthogonal graph drawings. Comput. Geom. Theory Appl. 9, 159–180 (1998)
3. Bläsius, T., Brückner, G., Rutter, I.: Complexity of higher-degree orthogonal graph embedding in the kandinsky model. CoRR abs/1405.2300 (2014)
4. Bläsius, T., Krug, M., Rutter, I., Wagner, D.: Orthogonal graph drawing with flexibility constraints. Algorithmica 68(4), 859–885 (2014)
5. Bläsius, T., Lehmann, S., Rutter, I.: Orthogonal graph drawing with inflexible edges. CoRR abs/1404.2943 (2014)
6. Bläsius, T., Rutter, I., Wagner, D.: Optimal orthogonal graph drawing with convex bend costs. In: Fomin, F.V., Freivalds, R., Kwiatkowska, M., Peleg, D. (eds.) ICALP 2013, Part I. LNCS, vol. 7965, pp. 184–195. Springer, Heidelberg (2013)
7. Cornelsen, S., Karrenbauer, A.: Accelerated bend minimization. J. Graph Algorithms Appl. 16(3), 635–650 (2012)
8. de Berg, M., Khosravi, A.: Optimal binary space partitions for segments in the plane. Int. J. Comput. Geometry Appl. 22(3), 187–206 (2012)
9. Dorn, F., Penninkx, E., Bodlaender, H., Fomin, F.: Efficient exact algorithms on planar graphs: Exploiting sphere cut decompositions. Algorithmica 58(3), 790–810 (2010)
10. Eiglsperger, M.: Automatic Layout of UML Class Diagrams: A Topology-Shape-Metrics Approach. Ph.D. thesis, Universität Tübingen (2003)
11. Eiglsperger, M., Gutwenger, C., Kaufmann, M., Kupke, J., Jünger, M., Leipert, S., Klein, K., Mutzel, P., Siebenhaller, M.: Automatic layout of UML class diagrams in orthogonal style. Information Visualization 3(3), 189–208 (2004)
12. Fößmeier, U., Kant, G., Kaufmann, M.: 2-visibility drawings of planar graphs. In: North, S. (ed.) GD 1996. LNCS, vol. 1190, pp. 155–168. Springer, Heidelberg (1997)
13. Fößmeier, U., Kaufmann, M.: Drawing high degree graphs with low bend numbers. In: Brandenburg, F.J. (ed.) GD 1995. LNCS, vol. 1027, pp. 254–266. Springer, Heidelberg (1996)
14. Garg, A., Tamassia, R.: On the computational complexity of upward and rectilinear planarity testing. SIAM J. Comput. 31(2), 601–625 (2001)
15. Gu, Q.P., Tamaki, H.: Optimal branch-decomposition of planar graphs in $O(n^3)$ time. ACM Trans. Alg. 4(3), 30:1–30:13 (2008)
16. Klau, G.W., Mutzel, P.: Quasi-orthogonal drawing of planar graphs. Research Report MPI-I-98-1-013, Max-Planck-Institut für Informatik (1998)
17. Leiserson, C.E.: Area-efficient graph layouts (for VLSI). In: FOCS 1980, pp. 270–281 (1980)
18. Tamassia, R.: On embedding a graph in the grid with the minimum number of bends. SIAM J. Comput. 16(3), 421–444 (1987)
19. Tamassia, R. (ed.): Handbook of Graph Drawing and Visualization. No. 81 in Discrete Mathematics and Its Applications. Chapman and Hall/CRC (2013)
20. Tamassia, R., Battista, G.D., Batini, C.: Automatic graph drawing and readability of diagrams. IEEE Trans. Syst., Man, Cybern., Syst. 18, 61–79 (1988)
21. Valiant, L.G.: Universality considerations in VLSI circuits. IEEE Trans. Comput. 30(2), 135–140 (1981)

A Subexponential Parameterized Algorithm for Proper Interval Completion[⋆]

Ivan Bliznets[1], Fedor V. Fomin[1,2], Marcin Pilipczuk[2], and Michał Pilipczuk[2]

[1] St. Petersburg Department of Steklov Institute of Mathematics, Russia
[2] Department of Informatics, University of Bergen, Norway
{fomin,Marcin.Pilipczuk,Michal.Pilipczuk}@ii.uib.no

Abstract. In the Proper Interval Completion problem we are given a graph G and an integer k, and the task is to turn G using at most k edge additions into a proper interval graph, i.e., a graph admitting an intersection model of equal-length intervals on a line. The study of Proper Interval Completion from the viewpoint of parameterized complexity has been initiated by Kaplan, Shamir and Tarjan [FOCS 1994; SIAM J. Comput. 1999], who showed an algorithm for the problem working in $\mathcal{O}(16^k \cdot (n+m))$ time. In this paper we present an algorithm with running time $k^{\mathcal{O}(k^{2/3})} + \mathcal{O}(nm(kn+m))$, which is the first subexponential parameterized algorithm for Proper Interval Completion.

1 Introduction

A graph G is an *interval graph* if it admits a model of the following form: each vertex is associated with an interval on the real line, and two vertices are adjacent if and only if the associated intervals overlap. If moreover the intervals can be assumed to be of equal length, then G is a *proper interval graph*; equivalently, one may require that no associated interval is contained in another. Interval and proper interval graphs appear naturally in molecular biology in the problem of *physical mapping*, where one is given a graph with vertices modeling contiguous intervals (called *clones*) in a DNA sequence, and the edges indicate which intervals overlap. Based on this information one would like to reconstruct the layout of the clones. We refer to [12] for further discussion on biological applications of (proper) interval graphs.

The biological motivation was the starting point of the work of Kaplan et al. [12], who initiated the study of (proper) interval graphs from the point of view of parameterized complexity. It is namely natural to expect that some information about overlaps will be lost, and hence the model will be missing a small number of edges. Thus we arrive at the problems of Interval Completion (IC) and Proper Interval Completion (PIC): given a graph G and an

[⋆] The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement n. 267959 and the Government of the Russian Federation (grant 14.Z50.31.0030). A full version of this paper is available at arXiv [4].

integer k, one is asked to add at most k edges to G to obtain a (proper) interval graph. Both problems are NP-hard [17], and hence it is natural to ask for an FPT algorithm parameterized by the number of additions. For PIC Kaplan et al. [12] presented an algorithm with running time $\mathcal{O}(16^k \cdot (n+m))$, while fixed-parameterized tractability of IC was proved much later by Villanger et al. [16]. Recently, Liu et al. [14] obtained an $\mathcal{O}(4^k + nm(n+m))$-time algorithm for PIC.

The approach of Kaplan et al. [12] is based on a characterization by *forbidden induced subgraphs*, also studied by Cai [5]: proper interval graphs are exactly graphs that are chordal (do not contain any induced cycle C_ℓ for $\ell \geq 4$) and additionally exclude three small graphs as induced subgraphs: a *claw*, a *tent*, and a *net* (see e.g. [2]). Thus, in PIC we may apply a basic branching strategy: Whenever a forbidden induced subgraph is encountered, we branch into several possibilities of how it is going to be destroyed in the optimal solution. A cycle C_ℓ can be destroyed only by adding $\ell - 3$ edges to triangulate it, and there are roughly $4^{\ell-3}$ different ways to do so. Since there is only a constant number of ways of destroying a small subgraph, the whole branching procedure runs in $c^k n^{\mathcal{O}(1)}$ time, for some constant c.

The approach via forbidden induced subgraphs has driven the research on the parameterized complexity of graph modification problems ever since the work of Cai [5]. Of particular importance was the work on polynomial kernelization; recall that a *polynomial kernel* for a parameterized problem is a polynomial-time preprocessing routine that reduces the size of the instance at hand to polynomial in the parameter. While many natural completion problems admit polynomial kernels, there are also examples where no such kernel exists under plausible complexity assumptions [13]. In particular, PIC admits a kernel with $\mathcal{O}(k^3)$ vertices which is computable in $\mathcal{O}(nm(kn+m))$ time [2], while the kernelization status of IC remains a notorious open problem.

The turning point came recently, when Fomin and Villanger [9] proposed an algorithm for Chordal Completion (aka Fill-in), that runs in *subexponential parameterized time*, more precisely $k^{\mathcal{O}(\sqrt{k})} n^{\mathcal{O}(1)}$. As observed by Kaplan et al. [12], the approach via forbidden induced subgraphs leads to an FPT algorithm for Fill-in with running time $16^k n^{\mathcal{O}(1)}$. However, in order to achieve a subexponential running time one needs to completely abandon this route, as even branching on obstacles as small as, say, induced C_4-s, leads to running time $2^k n^{\mathcal{O}(1)}$. To circumvent this, Fomin and Villanger proposed the approach of gradually building the structure of a chordal graph in a dynamic programming manner. The crucial observation was that the number of 'building blocks' (in their case, *potential maximal cliques*) is subexponential in a YES-instance, and thus the dynamic program operates on a subexponential space of states.

This research direction was continued by Ghosh et al. [10] and by Drange et al. [7], who identified several more graph classes for which completion problems have subexponential parameterized complexity: threshold graphs, split graphs, pseudo-split graphs, and trivially perfect graphs. Let us remark here that problems admitting subexponential parameterized algorithms are very scarce, since for most natural parameterized problems existence of such algorithms can be refuted under the Exponential Time Hypothesis (ETH) [11]. Up to very recently, the only natural positive examples were problems on specifically constrained inputs, like H-minor free graphs [6] or tournaments [1]. Thus, completion problems admitting subexponential parameterized algorithms can be regarded as 'singular points on the complexity landscape'. Indeed, Drange et al. [7] complemented their work with a number of lower bounds excluding (under ETH) subexponential parameterized algorithms for completion problems to many related graphs classes, e.g. cographs.

Interestingly, threshold, trivially perfect and chordal graphs, which are currently our main examples, correspond to graph parameters *vertex cover*, *treedepth*, and *treewidth* in the following sense: the parameter is equal to the minimum possible maximum clique size in a completion to the graph class (± 1). It is therefore natural to ask if INTERVAL COMPLETION and PROPER INTERVAL COMPLETION, which likewise correspond to *pathwidth* and *bandwidth*, also admit subexponential parameterized algorithms.

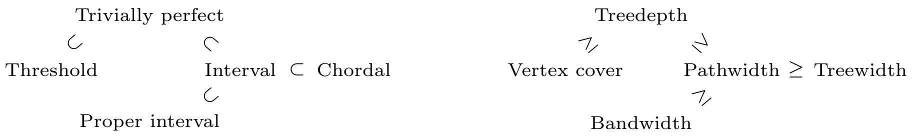


Fig. 1. Graph classes and corresponding graph parameters. Inequalities on the right side are with ± 1 slackness.

Our Results. In this paper we answer the question about PROPER INTERVAL COMPLETION in affirmative by proving the following theorem:

Theorem 1. PROPER INTERVAL COMPLETION *is solvable in time* $k^{\mathcal{O}(k^{2/3})} + \mathcal{O}(nm(kn+m))$.

In a companion paper [3] we also present an algorithm for INTERVAL COMPLETION with running time $k^{\mathcal{O}(\sqrt{k})}n^{\mathcal{O}(1)}$, which means that the completion problems for all the classes depicted on Fig. 1 in fact do admit subexponential parameterized algorithms. We now describe briefly our techniques employed to prove Theorem 1, and main differences with the work on interval graphs [3].

From a space-level perspective, both the approach of this paper and of [3] follows the route laid out by Fomin and Villanger in [9]. That is, we enumerate a subexponential family of potentially interesting building blocks, and then try to arrange them into a (proper) interval model with a small number of missing edges using dynamic programming (DP for short). In both cases, a natural candidate for this building block is the concept of a *cut*: given an interval model of a graph, imagine a vertical line placed at some position x that pins down intervals containing x. A *potential cut* is then a subset of vertices that becomes a cut in some minimal completion to a (proper) interval graph of cost at most k. The starting point of both this work and of [3] is enumeration of potential cuts. Using different structural insights into the classes of interval and proper interval graphs, one can show that in both cases the number of potential cuts is at most $n^{\mathcal{O}(\sqrt{k})}$, and they can be enumerated efficiently. Since in the case of proper interval graphs we can start with a cubic kernel given by Bessy and Perez [2], this immediately gives $k^{\mathcal{O}(\sqrt{k})}$ potential cuts for the PIC problem. In the interval case the question of existence of a polynomial kernel is widely open, and the need of circumventing this obstacle causes severe complications in [3].

Afterwards the approaches diverge completely, as it turns out that in both cases the potential cuts are insufficient building blocks to perform dynamic programming, however for very different reasons. For Interval Completion the problem is that the cut itself does not define what lies on the left and on the right of it. Even worse, there can be an exponential number of possible left/right alignments when the graph contains many modules that neighbor the same clique. To cope with this problem, the approach taken in [3] remodels the dynamic programming routine so that, in some sense, the choice of left/right alignment is taken care of inside the dynamic program. However, this leads to extremely complicated definition of a DP state and its relations.

Curiously, in the proper interval setting the left/right choice can be easily guessed along with a potential cut at basically no extra cost. Hence, the issue causing the most severe problems in the interval case is non-existent. The problem, however, is in the *ordering* of intervals in the cut: while performing a natural left-to-right DP that builds the model, we need to ensure that intervals participating in a cut begin in the same order as they end. Therefore, apart from the cut itself and a partition of the other vertices into left and right, a state would also need to include the ordering of the vertices of the cut; as the cut may be large, we cannot afford constructing a state for every possible ordering.

Instead we remodel the dynamic program, this time by introducing two layers. We first observe that the troublesome ordering may be guessed expeditiously providing that the cut in question has only a sublinear in k number of incident edge additions. Hence, in the first layer of dynamic programming we aim at chopping the optimally completed model using such cheap cuts, and to conclude the algorithm we just need to be able to compute the best possible completed model between two border cuts that are cheap, assuming that all the intermediate cuts are expensive. This task is performed by the layer-two dynamic program. The main observation is that since all the intermediate cuts are expensive, there cannot be many disjoint such cuts and, consequently, the space between the border cuts is in some sense 'short'. As the border cuts can be large, it is natural to start partitioning the space in between 'horizontally' instead of 'vertically' — shortness of this space guarantees that the number of sensible 'horizontal' separations is subexponential. The horizontal partitioning method that we employ resembles the classic $\mathcal{O}^\star(10^n)$-time exact algorithm for bandwidth of Feige [8].

2 Preliminaries

In most cases, we follow standard graph notation.

For integers a, b, we denote $[a, b] = \{a, a+1, \dots, b\}$. An ordering σ of a subset $X \subseteq V(G)$ is an injective function $\sigma : X \to [1, |V(G)|]$, and an ordering of G

is simply an ordering of $V(G)$. Note that an ordering of G is a bijection. We sometimes treat an ordering σ of $X \subseteq V(G)$ as an ordering of $G[X]$ as well, implicitly identifying $\sigma(X)$ with $[1, |X|]$ in the monotonous way.

We use n and m to denote the number of vertices and edges of the input graph, respectively. Moreover, for an input graph G we fix some arbitrary order $\preceq$ of $V(G)$ and with every ordering σ of $X = \{x_1 \prec x_2 \prec \ldots \prec x_{|X|}\} \subseteq V(G)$ we associate a sequence $(\sigma(x_1), \sigma(x_2), \ldots, \sigma(x_{|X|}))$. The *lexicographically minimum* ordering from some family of orderings of a fixed set X is the ordering with lexicographically minimum associated sequence.

A graph G is a *proper interval graph* if it admits an intersection model, where each vertex is assigned a closed interval on a line such that no interval is a proper subset of another one. In our work it is more convenient to use an equivalent combinatorial object, called an *umbrella ordering.*

Definition 2 (Umbrella Ordering). *Let G be a graph and $\sigma : V(G) \to [1, n]$ be an ordering. We say that σ satisfies the* umbrella property *for a triple $a, b, c \in V(G)$ if $ac \in E(G)$ and $\sigma(a) < \sigma(b) < \sigma(c)$ implies $ab, bc \in E(G)$. Furthermore, σ is an* umbrella ordering *if it fulfills the umbrella property for all $a, b, c \in V(G)$.*

It is known that a graph is a proper interval graph if and only if it admits an umbrella ordering [15]. Observe that we may equivalently define an umbrella ordering σ as such an ordering that for every $ab \in E(G)$ with $\sigma(a) < \sigma(b)$, the vertices in $[\sigma(a), \sigma(b)]$ in σ form a clique in G, or, alternatively, if and only if for every $a, a', b', b \in V(G)$ such that $\sigma(a) \leq \sigma(a') < \sigma(b') \leq \sigma(b)$, if $ab \in E(G)$, then also $a'b' \in E(G)$.

For a graph G, a *completion* of G is a set $F \subseteq \binom{V(G)}{2} \setminus E(G)$ such that $G + F := (V(G), E(G) \cup F)$ is a proper interval graph. The Proper Interval Completion problem asks for a completion of G of size not exceeding a given budget k. For a completion F of G and $v \in V(G)$, we denote by $F(v)$ the set of edges of F incident with v and for $X \subseteq V(G)$, we define $F(X) = \bigcup_{v \in X} F(v)$.

However, for our purposes it is more convenient to work with orderings rather than completions. Consider an ordering σ and define F^σ to be the set of these pairs $uv \notin E(G)$ such that there exists an edge $u'v' \in E(G)$ with $\sigma(u') \leq \sigma(u) < \sigma(v) \leq \sigma(v')$. It is straightforward to verify the following.

Lemma 3. *The graph $G^\sigma := G + F^\sigma$ is a proper interval graph, and σ is its umbrella ordering. Moreover, F^σ is the unique inclusion-wise minimal completion of G among completions F for which σ is an umbrella ordering of $G + F$.*

The *canonical ordering* of a graph G is the lexicographically minimum ordering among orderings σ with minimum possible $|F^\sigma|$. For a canonical ordering σ, the set F^σ is called the *canonical completion.* If additionally $|F^\sigma| \leq k$, the canonical ordering σ is also called the *canonical solution.*

Our starting point for the proof of Theorem 1 is the polynomial kernel for Proper Interval Completion due to Bessy and Perez.

Theorem 4 ([2]). Proper Interval Completion *admits a kernel with $\mathcal{O}(k^3)$ vertices computable in time $\mathcal{O}(nm(kn + m))$.*

The algorithm of Theorem 1 starts with applying the kernelization algorithm of Theorem 4; all further computation will take $k^{\mathcal{O}(k^{2/3})}$ time, yielding the promised time bound. Hence, in the rest of the paper we assume that we are given a PIC instance (G, k) with $n = |V(G)| = \mathcal{O}(k^3)$, and we are looking for the canonical solution of G provided that (G, k) is a YES-instance. Moreover, by standard arguments we may assume that G is connected.

3 Expensive Vertices

We first deal with vertices that are incident with many edges of F^σ. Formally, we set a threshold $\tau := (2k)^{1/3}$ and say that a vertex v is *expensive* with respect to σ if it is incident with more than τ edges of F^σ, and *cheap* otherwise. As there are at most $(2k)^{2/3} = \tau^2$ expensive vertices, we may afford guessing a lot of information about expensive vertices within the promised time bound.

More formally, we branch into $k^{\mathcal{O}(k/\tau)} = k^{\mathcal{O}(k^{2/3})}$ subcases corresponding to the guesses about the expensive vertices in the canonical solution σ. We consider all possible

– sets $V_\$ \subseteq V(G)$ of size at most τ^2 of expensive vertices w.r.t. σ, and
– for each $V_\$$, all possible quadruples (v, p_v, p_v^L, p_v^R), where $v \in V_\$$, and p_v, p_v^L, p_v^R are integers such that $p_v = \sigma(v)$, $p_v^L = \min\{\sigma(w) : w \in N_{G^\sigma}[v]\}$ and $p_v^R = \max\{\sigma(w) : w \in N_{G^\sigma}[v]\}$.

In each branch, we look for the canonical solution to the instance (G, k), assuming that the aforementioned guess is the correct one. The *correct branch* is the one where this assumption is indeed true.

In each branch, some consistency checks are in place. For instance, the mapping $v \mapsto p_v$ should be injective, $p_{v_1} < p_{v_2}$ should imply $p_{v_1}^L \le p_{v_2}^L$ and $p_{v_1}^R \le p_{v_2}^R$, etc. We omit the full description of these checks in this extended abstract.

Observe that, in the correct branch, a vertex $v \in V_\$$ has degree exactly $p_v^R - p_v^L$ in the graph G^σ, with its closed neighborhood placed on positions $[p_v^L, p_v^R]$. This motivates us to define the following

$$\begin{aligned} F_\$ &= \{v_1 v_2 : v_1, v_2 \in V_\$ \wedge v_1 \neq v_2 \wedge v_1 v_2 \notin E(G) \wedge v_1 \in [p_{v_2}^L, p_{v_2}^R]\}, \\ c_\$ &= -|F_\$| + \sum_{v \in V_\$} \left((p_v^R - p_v^L) - \deg_G(v)\right). \end{aligned}$$

Let us observe that $F_\$$ is the set comprising edges of F^σ with both endpoints in $V_\$$, i.e. $F_\$ = F^\sigma \cap \binom{V_\$}{2}$, and that $c_\$$ is the number of edges of F^σ incident with $V_\$$, i.e. $c_\$ = |F^\sigma(V_\$)|$. Both notions are meaningful for every branch.

Lemma 5. *Let σ' be an ordering of $V(G)$ and F be a completion of G such that (i) σ' is an umbrella ordering of $G + F$, and (ii) for every $v \in V_\$$, we have $\sigma'(v) = p_v$ and $\sigma'(N_{G+F}[v]) = [p_v^L, p_v^R]$. Then $F \cap \binom{V_\$}{2} = F_\$$ and $|F(V_\$)| = c_\$$.*

We infer that the guesses made so far impose a fixed cost of $c_\$$ edges and it is tempting to consider the remaining instance $(G \setminus V_\$, k - c_\$)$. However, the guessed values impose some constraints on this remaining instance. First, if $uv \in E(G)$ for some expensive v and cheap u, we need to have $\sigma(u) \in [p_v^L, p_v^R]$. Second, due to the umbrella property, for any expensive v, all vertices placed on positions $[p_v^L, p_v]$ become a clique, whereas no edge of G connects a vertex placed before position p_v^L and a vertex placed on or after position p_v; similar constraints are imposed for positions p_v and p_v^R.

Luckily, all these constraints can be modelled as (i) prescribing for each cheap u a set $\Sigma_u \subseteq [1, n]$ of allowed positions, and (ii) prescribing some pairs of positions to be necessarily adjacent or necessarily nonadjacent in the ordering σ. It turns out that the aforementioned additional constraints only slightly increase the technical level of further reasonings, and none of them adds any significant difficulty. Hence, in this extended abstract we ignore them, and assume that in the canonical ordering σ there are *no* expensive vertices.

4 Sections

For any position p in the canonical ordering σ we define a *section* $A_p = \{v \in V(G) : \sigma(v) < p\}$, and additionally $A_\infty = V(G)$. We are now going to show the vital combinatorial result: in the absence of expensive vertices, there is only subexponential number of candidates for sections of σ.

Theorem 6. *In $k^{\mathcal{O}(\tau)}$ time one can enumerate a family $\mathcal{S}$ of $k^{\mathcal{O}(\tau)}$ subsets of $V(G)$ such that every section of the canonical solution σ is in $\mathcal{S}$.*

The main idea in the proof of Theorem 6 is to investigate *twin classes* in graph G^σ. Recall that two vertices x and y are *true twins* if $N[x] = N[y]$; in particular, this implies that they are adjacent. The relation of being true twins is an equivalence relation, and equivalence classes of this relation are called *twin classes.* Observe that by the definition of the umbrella ordering, in σ the vertices of each twin class of G^σ occupy consecutive positions. We show the following bound on the number of candidates for twin classes.[1]

Theorem 7. *In $k^{\mathcal{O}(\tau)}$ time one can enumerate a family $\mathcal{T}$ of $k^{\mathcal{O}(\tau)}$ triples $(L, \Lambda, \sigma_\Lambda)$, where $L, \Lambda \subseteq V(G)$ and σ_Λ is an ordering of Λ, with the following property. For every twin class Λ of G^σ, if L is the set of vertices of G placed before Λ in the ordering σ, and $\sigma|_\Lambda$ is the ordering σ restricted to Λ, then $(L, \Lambda, \sigma|_\Lambda) \in \mathcal{T}$.*

We remark that it is easy to derive Theorem 6 from Theorem 7: We first output the section $V(G)$ and then, for each $(L, \Lambda, \sigma_\Lambda) \in \mathcal{T}$ and $p \in [1, n]$, we output $L \cup \{u \in \Lambda : \sigma_\Lambda(u) < p\}$. Observe that if a section $A_p \neq V(G)$ is consistent with σ, then A_p is output for the position p and the triple $(L, \Lambda, \sigma|_\Lambda) \in \mathcal{T}$ where Λ is the twin class of vertex $\sigma^{-1}(p)$.

[1] We care about the order inside twin classes because inside a single twin class we may have different restrictions imposed by the guesses on expensive vertices made in the previous section.

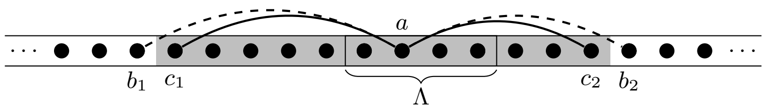


Fig. 2. The guessed vertices a, b_1, b_2, c_1 and c_2 with respect to a twin class Λ. The gray area denotes $N_{G^\sigma}[\Lambda]$.

To prove Theorem 7, we describe a branching algorithm that produces $k^{\mathcal{O}(\tau)}$ subcases and, in each subcase, produces one triple $(L, \Lambda, \sigma_\Lambda)$. We fix one twin class Λ of G^σ and argue that the algorithm in one of the branches produces $(L, \Lambda, \sigma|_\Lambda)$, where L is defined as in Theorem 7.

The algorithm first guesses the following five vertices, see also Fig. 2.

1. a is a vertex of Λ,
2. b_1 is the rightmost vertex outside $N_{G^\sigma}[\Lambda]$ in σ that lies before Λ, or $b_1 = \perp$ if no such vertex exists;
3. c_1 is the leftmost vertex of $N_{G^\sigma}[\Lambda]$ in σ;
4. c_2 is the rightmost vertex of $N_{G^\sigma}[\Lambda]$ in σ;
5. b_2 is the leftmost vertex outside $N_{G^\sigma}[\Lambda]$ in σ that lies after Λ, or $b_2 = \perp$ if no such vertex exists.

Moreover, for each $u \in \{a, b_1, b_2, c_1, c_2\} \setminus \{\perp\}$ the algorithm guesses $F^\sigma(u)$. This leads us to $k^{\mathcal{O}(\tau)}$ subcases, as all vertices of G are cheap. The crucial step in deducing the triple $(L, \Lambda, \sigma_\Lambda)$ is the following lemma (we take $N_{G^\sigma}[\perp] = \emptyset$).

Lemma 8. *In the branch where the guesses are correct, for every $u \in N_{G^\sigma}[a]$ the following holds*

1. *If $u \in N_{G^\sigma}[b_1]$ or $u \notin N_{G^\sigma}[c_2]$, then $u \notin \Lambda$ and u lies before Λ in σ;*
2. *If $u \in N_{G^\sigma}[b_2]$ or $u \notin N_{G^\sigma}[c_1]$, then $u \notin \Lambda$ and u lies after Λ in σ;*
3. *If none of the above happens, then $u \in \Lambda$.*

Proof. By the definition of b_1, b_2, c_1 and c_2, we have that every vertex $u \in \Lambda$ is in $N_{G^\sigma}[c_1]$ and $N_{G^\sigma}[c_2]$, but does not belong to $N_{G^\sigma}[b_1]$ and to $N_{G^\sigma}[b_2]$. Consequently, all vertices of Λ fall into the third category of the statement.

We now show that every vertex of $N_{G^\sigma}[a] \setminus \Lambda$ falls into one of the first two categories, depending on its position in the ordering σ. By symmetry, we may only consider a vertex $u \in N_{G^\sigma}[a] \setminus \Lambda$ that lies before Λ in σ. The umbrella property together with $a \notin N_{G^\sigma}[b_2]$ imply that $u \notin N_{G^\sigma}[b_2]$, and because $ac_1 \in E(G^\sigma)$, we have that $uc_1 \in E(G^\sigma)$. Consequently, u does not fall into the second category in the statement of the lemma.

As $u \notin \Lambda$ and $u \in N_{G^\sigma}[a]$, either $N_{G^\sigma}(u) \setminus N_{G^\sigma}[a]$ is not empty or $N_{G^\sigma}(a) \setminus N_{G^\sigma}[u]$ is not empty. In the first case, let $uw \in E(G^\sigma)$ but $aw \notin E(G^\sigma)$. Since also $ua \in E(G^\sigma)$, by the umbrella property it easily follows that w lies before u in the ordering σ, so in particular before Λ. By the definition of b_1, b_1 exists and $\sigma(b_1) \geq \sigma(w)$. By the umbrella property, $b_1u \in E(G^\sigma)$ and hence $u \in N_{G^\sigma}[b_1]$.

In the second case, assume $uw \notin E(G^\sigma)$ but $aw \in E(G^\sigma)$. Again, since $ua \in E(G^\sigma)$, by the umbrella property it easily follows that w lies after Λ in the ordering σ, so in particular after u. By the definition of c_2 and the existence of w, $c_2 \notin \Lambda$ and $\sigma(c_2) \geq \sigma(w)$. By the umbrella property, $c_2u \notin E(G^\sigma)$ and $u \notin N_{G^\sigma}[c_2]$. Hence, u falls into the first category and the lemma is proven. $\square$

The knowledge of a and $F^\sigma(a)$ allows us to compute $N_{G^\sigma}[\Lambda] = N_{G^\sigma}[a]$. Lemma 8 allows us further to partition $N_{G^\sigma}[\Lambda]$ into Λ, the vertices of $N_{G^\sigma}(\Lambda)$ that lie before Λ in the ordering σ, and the ones that lie after Λ.

We are left with the vertices outside $N_{G^\sigma}[\Lambda]$. Let C be a connected component of $G \setminus N_{G^\sigma}[\Lambda]$. As no vertex of C is incident with Λ in G^σ, by the properties of an umbrella ordering we infer that all vertices of $N_G[C]$ lie before Λ in the ordering σ or all vertices of $N_G[C]$ lie after Λ. As G is assumed to be connected, $N_G(C)$ contains a vertex of $N_{G^\sigma}(\Lambda)$, and we can deduce whether $C \subseteq L$ or $L \cap C = \emptyset$.

Finally, as Λ is a twin class in G^σ, the ordering σ sorts Λ according to $\preceq$. Thus, $\sigma|_\Lambda$ depends only on the position $p = \min \sigma(\Lambda)$, which we simply guess.

We remark here that there are some slight difficulties if we have some additional constraint imposed by the guesses of the previous section. First, for a connected component C of $G \setminus N_{G^\sigma}[\Lambda]$ it may happen that $N_G(C) \subseteq V_\$$. In this case, however, we may deduce whether $C \subseteq L$ or $C \cap L = \emptyset$ from the guessed positions of the expensive vertices and the position p of the first vertex of Λ. Second, the ordering $\sigma|_\Lambda$ needs to respect the prescribed allowed positions Σ_u for $u \in \Lambda$. Luckily, with this constraint the task of finding $\sigma|_\Lambda$ boils down to a task of finding a lexicographically minimum perfect matching in an auxiliary bipartite graph, which is solvable in polynomial time.

5 Dynamic Programming

Layer One: Jumps and Jump Sets. Armed with Theorem 6, we proceed to design a dynamic programming algorithm that constructs the canonical solution σ. We first develop a natural left-to-right DP that splits the graphs G and G^σ 'vertically'. For each position p, we define the *jump* and *jump set* X_p as

$$\begin{aligned} \texttt{jump}(p) &= \min\{q : q > p \wedge \sigma^{-1}(p)\sigma^{-1}(q) \notin E(G^\sigma)\}, \\ X_p &= \sigma^{-1}([p, \texttt{jump}(p) - 1]) = A_{\texttt{jump}(p)} \setminus A_p. \end{aligned}$$

See also Fig. 3. The separation property of a jump is provided by the following direct consequence of the property of the umbrella ordering.

Lemma 9. *For each $p \in [1, n]$, X_p is a clique in G^σ and no edge of G^σ connects a vertex of A_p with a vertex of $V(G) \setminus A_{\texttt{jump}(p)}$.*

It is tempting to define a DP state as a 'jump set with a history' $J := (A, X)$, where its value is an ordering $\sigma_J := A \cup X \to [1, |A \cup X|]$ that first places the vertices of A and then of X, with the intention that X is a jump set after the remaining vertices are placed (i.e., X is a clique in $G^{\sigma_J}[A \cup X]$, but no edge

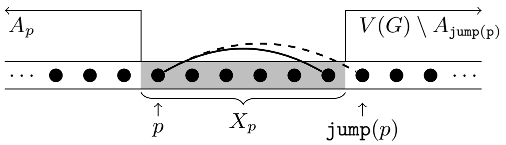


Fig. 3. A jump at position p and a corresponding jump set. The jump set X_p, denoted with gray, induces a clique in G^σ, and no edge of G^σ connects A_p with $V(G) \setminus A_{\mathtt{jump}(p)}$.

connects the first vertex of X in σ_J with $V(G)\setminus(A\cup X)$). However, this approach fails for the following reason: the internal ordering of the vertices of X affects both the ordering of A and the ordering of $V(G) \setminus (A \cup X)$, and hence needs to be stored in the DP state as well. Luckily, the ordering of X can be deduced from the knowledge of $F^\sigma(X)$, that is, the completion edges incident with X.

Lemma 10. *For each $p \in [1, n]$, if $u_1, u_2 \in X_p$ and $\sigma(u_1) \leq \sigma(u_2)$, then*

$$N_{G^\sigma}(u_1) \cap A_p \supseteq N_{G^\sigma}(u_2) \cap A_p \text{ and } N_{G^\sigma}(u_1) \setminus A_{\mathtt{jump}(p)} \subseteq N_{G^\sigma}(u_2) \setminus A_{\mathtt{jump}(p)}.$$

It is easy to observe that, after satisfying the conditions of Lemma 10, we may proceed greedily. That is, the ordering σ sorts the vertices of X according to Lemma 10, breaking ties using order $\preceq$ to preserve lexicographical minimality.

It is not clear (if possible at all) how to provide a subexponential number of candidate orderings of X but we can do it in the case when $F^\sigma(X)$ is small. More precisely, we say that a jump set X is *cheap* if it is incident to at most $2k/\tau$ edges of F^σ, and *expensive* otherwise. Consequently, we may enumerate $k^{\mathcal{O}(2k/\tau)} = k^{\mathcal{O}(k^{2/3})}$ candidate triples (A, X, σ_X) for $(A_p, X_p, \sigma|_{X_p})$ for each *cheap* jump sets X_p. The layer one DP treats such triples as states, and finds for each such triple (A, X, σ_X) the optimal ordering $(A \cup X) \mapsto [1, |A \cup X|]$ that is consistent with σ_X. However, now layer one DP needs to perform a big task in a single step: namely, it needs to find the optimal way to arrange vertices between two consecutive cheap jumps. We delegate this task to the layer two DP, described in what follows.

Layer Two: Chains. Here we assume that we are given two of the layer one states (A^1, X^1, σ_X^1), (A^2, X^2, σ_X^2), with no cheap jump sets between X^1 and X^2, and we are to place the vertices of $(A^2 \cup X^2)\setminus A^1$ on positions $[|A^1|+1, |A^2 \cup X^2|]$ respecting σ_X^1 and σ_X^2.

We now derive a different 'horizontal' way of partitioning G and G^σ, based on the following definition. For any $q \in [1, n]$, consider the following sequence: $z_q(0) = q$ and $z_q(i + 1) = \mathtt{jump}(z_q(i))$ (taking $\mathtt{jump}(\infty) = \infty$). Observe that:

Lemma 11. *For any $q > |A^1|$, it holds that $z_q(\tau) \geq |A^2|$.*

Proof. Observe that for each $i > 0$ with $z_q(i) < |A^2|$, the jump set $X_{z_q(i)}$ is expensive and, moreover, these jump sets are pairwise disjoint for different choices of i. Hence, there are less than τ such jump sets. $\square$

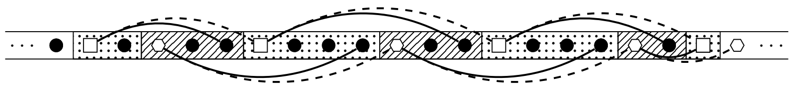


Fig. 4. The separation property provided by Lemma 12. The sequences $z_c(i)$ and $z_d(i)$ are denoted with rectangular and hexagonal shapes, respectively. The sets C_i and D_i are denoted boxes with dots and lines, respectively.

Moreover, observe that if we pick two positions c, d with $c \le d \le \mathtt{jump}(c)$ we have $z_c(i) \le z_d(i) \le z_c(i+1)$ for any $i \ge 0$.

The next immediate corollary of the definition of the umbrella property and the jump gives us the crucial separation property for the layer two DP (see Fig. 4).

Lemma 12. *For any positions c, d with $c \le d \le \mathtt{jump}(c)$, let $C_i = \sigma^{-1}([z_c(i), z_d(i) - 1])$ and $D_i = \sigma^{-1}([z_d(i), z_c(i+1) - 1])$. Then*

1. *sets C_i, D_i form a partition of $V(G) \setminus A_c$;*
2. *for every $i \ge 0$, both $C_i \cup D_i$ and $D_i \cup C_{i+1}$ are cliques in G^σ;*
3. *for every $j > i \ge 0$, there is no edge in G^σ between C_i and D_j;*
4. *for every $i > j + 1 > 0$, there is no edge in G^σ between C_i and D_j.*

Lemma 12 allows us to define a layer two DP state consisting of sequences $z_c(i)$ and $z_d(i)$, up to minimum index i that satisfies $z_c(i) > |A^2|$, together with sections $A_{z_c(i)}$ and $A_{z_d(i)}$, for some choice of starting positions $|A^1| < c \le d \le \min(\mathtt{jump}(c), |A^2 \cup X^2|)$. In such a state, we ask for an optimal ordering of the sets $X^1 \cup X^2 \cup \bigcup_i C_i$ that respects the orderings σ_X^1 and σ_X^2, places the vertices of each C_i into positions $[z_c(i), z_d(i) - 1]$ and turns each C_i into a clique. Note that Theorem 6, together with Lemma 11, gives us a bound $k^{\mathcal{O}(\tau^2)} = k^{\mathcal{O}(k^{2/3})}$ on the number of such states.

To compute the value of a layer two DP state, we guess the sequence $z_q(i)$ 'sandwiched' between $z_c(i)$ and $z_d(i)$ for some $c < q < d$, with the corresponding sections $A_{z_q(i)}$, and we glue the optimal values for states (c, q) and (q, d). If no such q exists, there are two special cases. If $c = d$, then the DP state in fact asks for $\sigma_X^1 \cup \sigma_X^2$. Finally, if $c + 1 = d$, then observe that all vertices of C_1 are adjacent to $\sigma^{-1}(d)$ and nonadjacent to $\sigma^{-1}(c)$. Hence, the vertices at positions c and d do not impose any constraints on the ordering of C_1, and, as a value for the state (c, d), we may use the value of the state $(\mathtt{jump}(c), \mathtt{jump}(d)) = (z_c(1), z_d(1))$, extended with the placement of the unique vertex of $A_d \setminus A_c$ at position c.

Overall, the described layer two DP allows us to perform a single step of the layer one DP in time $k^{\mathcal{O}(\tau^2)} = k^{\mathcal{O}(k^{2/3})}$. This concludes the proof of Theorem 1.

6 Conclusions and Open Problems

We have presented the first subexponential algorithm for Proper Interval Completion, running in time $k^{\mathcal{O}(k^{2/3})} + \mathcal{O}(nm(kn+m))$. As many algorithms for completion problems in similar graph classes [3,7,9,10] run in time $\mathcal{O}^\star(k^{\mathcal{O}(\sqrt{k})})$, it is tempting to ask for such a running time also in our case. The bottleneck in our approach is the trade-offs between the two layers of dynamic programming.

Also, observe that every $\mathcal{O}^\star(2^{o(\sqrt{k})})$-time algorithm for PIC would be in fact also a $2^{o(n)}$-time algorithm. Since existence of such an algorithm seems unlikely, we would like to ask for a $2^{\Omega(\sqrt{k})}$ lower bound, under the assumption of the Exponential Time Hypothesis. Note that no such lower bound is known for any other completion problem for related graph classes.

References

1. Alon, N., Lokshtanov, D., Saurabh, S.: Fast FAST. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009, Part I. LNCS, vol. 5555, pp. 49–58. Springer, Heidelberg (2009)
2. Bessy, S., Perez, A.: Polynomial kernels for Proper Interval Completion and related problems. Information and Computation 231, 89 (2013)
3. Bliznets, I., Fomin, F.V., Pilipczuk, M., Pilipczuk, M.: A subexponential parameterized algorithm for interval completion. CoRR abs/1402.3473 (2014)
4. Bliznets, I., Fomin, F.V., Pilipczuk, M., Pilipczuk, M.: A subexponential parameterized algorithm for proper interval completion. CoRR abs/1402.3472 (2014)
5. Cai, L.: Fixed-parameter tractability of graph modification problems for hereditary properties. Inf. Process. Lett. 58(4), 171–176 (1996)
6. Demaine, E.D., Fomin, F.V., Hajiaghayi, M., Thilikos, D.M.: Subexponential parameterized algorithms on graphs of bounded genus and H-minor-free graphs. J. ACM 52(6), 866–893 (2005)
7. Drange, P.G., Fomin, F.V., Pilipczuk, M., Villanger, Y.: Exploring subexponential parameterized complexity of completion problems. In: STACS 2014 (2014)
8. Feige, U.: Coping with the NP-hardness of the graph bandwidth problem. In: Halldórsson, M.M. (ed.) SWAT 2000. LNCS, vol. 1851, pp. 10–19. Springer, Heidelberg (2000)
9. Fomin, F.V., Villanger, Y.: Subexponential parameterized algorithm for minimum fill-in. SIAM J. Comput. 42(6), 2197–2216 (2013)
10. Ghosh, E., Kolay, S., Kumar, M., Misra, P., Panolan, F., Rai, A., Ramanujan, M.S.: Faster parameterized algorithms for deletion to split graphs. In: Fomin, F.V., Kaski, P. (eds.) SWAT 2012. LNCS, vol. 7357, pp. 107–118. Springer, Heidelberg (2012)
11. Impagliazzo, R., Paturi, R., Zane, F.: Which problems have strongly exponential complexity? J. Comput. Syst. Sci. 63(4), 512–530 (2001)
12. Kaplan, H., Shamir, R., Tarjan, R.E.: Tractability of parameterized completion problems on chordal, strongly chordal, and proper interval graphs. SIAM J. Comput. 28(5), 1906–1922 (1999)
13. Kratsch, S., Wahlström, M.: Two edge modification problems without polynomial kernels. Discrete Optimization 10(3), 193–199 (2013)
14. Liu, Y., Wang, J., Xu, C., Guo, J., Chen, J.: An effective branching strategy for some parameterized edge modification problems with multiple forbidden induced subgraphs. In: Du, D.-Z., Zhang, G. (eds.) COCOON 2013. LNCS, vol. 7936, pp. 555–566. Springer, Heidelberg (2013)
15. Looges, P.J., Olariu, S.: Optimal greedy algorithms for indifference graphs. Computers and Mathematics with Applications 25(7), 15–25 (1993)
16. Villanger, Y., Heggernes, P., Paul, C., Telle, J.A.: Interval completion is fixed parameter tractable. SIAM J. Comput. 38(5), 2007–2020 (2009)
17. Yannakakis, M.: Computing the minimum fill-in is NP-complete. SIAM J. Alg. Disc. Meth. 2, 77–79 (1981)

Computing Persistent Homology with Various Coefficient Fields in a Single Pass[⋆]

Jean-Daniel Boissonnat and Clément Maria

INRIA Sophia Antipolis-Méditerranée, France
{jean-daniel.boissonnat,clement.maria}@inria.fr

Abstract. This article introduces an algorithm to compute the persistent homology of a filtered complex with various coefficient fields in a single matrix reduction. The algorithm is output-sensitive in the total number of *distinct* persistent homological features in the diagrams for the different coefficient fields. This computation allows us to infer the prime divisors of the torsion coefficients of the integral homology groups of the topological space at any scale, hence furnishing a more informative description of topology than persistence in a single coefficient field. We provide theoretical complexity analysis as well as detailed experimental results. The code is part of the `Gudhi` library, and is available at [8].

1 Introduction

Persistent homology [5,12] is an algebraic method for measuring the topological features of the sublevel sets of a function defined on a topological space. Its generality and stability [4] with regard to noise have made it a widely used tool for the study of data. At the algebraic level [12], it admits a decomposition – represented by mean of a persistence diagram – only when considered with field coefficients (by opposition to integer coefficients). The persistence diagram contains a rich information about the topology of the studied space and very efficient methods exist to compute it. However, the integral homology groups of a topological space are strictly more informative than the homology groups with field coefficients, in particular because they convey information about "torsion". Torsion can be pictured geometrically as a "twisting" of the shape and happens frequently as global topological feature in topological data analysis where, for example, Klein bottles appear naturally [3,9]. Algebraically, torsion is characterized by cyclic subgroups of the integral homology groups. When computed with field coefficients, these subgroups may either vanish or appear as "infinite", and consequently obfuscate the study of the topology of data. A simple solution is to compute persistent homology with different coefficient fields and track the differences in the persistence diagrams.

[⋆] This research has been partially supported by the European Research Council under Advanced Grant 339025 GUDHI (Algorithmic Foundations of Geometric Understanding in Higher Dimensions).

We build on this idea and describe an algorithm to compute persistent homology with various coefficient fields $\mathbb{Z}_{q_1}, \cdots, \mathbb{Z}_{q_r}$ in a single pass of the matrix reduction algorithm, where $\mathbb{Z}_q$ denotes the finite field $\mathbb{Z}/q\mathbb{Z}$ for a prime q. To do so, we introduce a method we call *modular reconstruction* consisting in using the *Chinese Remainder Isomorphism* to encode an element of $\mathbb{Z}_{q_1} \times \cdots \times \mathbb{Z}_{q_r}$ with an element of $\mathbb{Z}_{q_1 \cdots q_r}$. We describe algorithms to perform elementary row/column operations in a matrix with $\mathbb{Z}_{q_1 \cdots q_r}$ coefficients, corresponding to simultaneous elementary row/column operations in matrices with coefficients in the fields $\mathbb{Z}_{q_1}, \cdots, \mathbb{Z}_{q_r}$. The method results in an algorithm with an output-sensitive complexity in the total number of *distinct* pairs in the echelon forms of the matrices with $\mathbb{Z}_{q_1}, \cdots, \mathbb{Z}_{q_r}$ coefficients, plus an overhead due to arithmetic operations on big numbers in $\mathbb{Z}_{q_1 \cdots q_r}$. The method is generic and applies to every algorithm for persistent homology. Finally, we describe how to infer the torsion coefficients of the integral homology using the *Universal Coefficient Theorem for Homology.*

We provide detailed experimental analysis of the algorithm and show, in particular, that on practical examples our method is substancially faster than the brute-force approach consisting in reducing separately r matrices with coefficients in $\mathbb{Z}_{q_1}, \cdots, \mathbb{Z}_{q_r}$. It is important to note that the method does not pretend to scale to very large r, as the arithmetic complexity of operations in $\mathbb{Z}_{q_1 \cdots q_r}$ becomes problematic. Experiments show, however, that for very large r (up to 100000) our approach is still substancially faster than brute-force.

Computing persistent homology with different coefficients has been mentioned in the literature [12] in order to verify if a persisting feature was due to an actual "hole" (or high-dimensional equivalent) or to torsion (and consequently existed only for a certain coefficient field). However, to the best of our knowledge, this is the first work formalizing the inference of torsion coefficients in the framework of persistent homology and describing an efficient algorithm to compute persistence with various coefficient fields.

2 Multi-field Persistent Homology

For simplicity, we focus in the following on simplicial homology. However, the approach applies to any type of boundary matrix (defined below).

Background on Simplicial Homology and Persistence: A *simplicial complex* $\mathbf{K}$ on a set of *vertices* $V = \{1, \cdots, n\}$ is a collection of simplices $\{\sigma\}$, $\sigma \subseteq V$, such that $\tau \subseteq \sigma \in \mathbf{K} \Rightarrow \tau \in \mathbf{K}$. The dimension $d = |\sigma| - 1$ of σ is its number of elements minus 1. For a ring $\mathcal{R}$, the group of d-chains, denoted $\mathbf{C}_d(\mathbf{K}, \mathcal{R})$, of $\mathbf{K}$ is the group of formal sums of d-simplices with $\mathcal{R}$ coefficients. The *boundary operator* is a linear operator $\partial_d : \mathbf{C}_d(\mathbf{K}, \mathcal{R}) \to \mathbf{C}_{d-1}(\mathbf{K}, \mathcal{R})$ such that $\partial_d \sigma = \partial_d [v_0, \cdots, v_d] = \sum_{i=0}^{d} (-1)^i [v_0, \cdots, \widehat{v_i}, \cdots, v_d]$, where $\widehat{v_i}$ means v_i is deleted from the list. It will be convenient to consider later the endomorphism $\partial_* : \bigoplus_d \mathbf{C}_d(\mathbf{K}, \mathcal{R}) \to \bigoplus_d \mathbf{C}_d(\mathbf{K}, \mathcal{R})$ extended by linearity to the external sum of chain groups. Denote by $\mathbf{Z}_d(\mathbf{K}, \mathcal{R})$ and $\mathbf{B}_{d-1}(\mathbf{K}, \mathcal{R})$ the kernel and the image of ∂_d respectively. Observing $\partial_d \circ \partial_{d+1} = 0$, we define the d^{th} homology group $\mathbf{H}_d(\mathbf{K}, \mathcal{R})$ of $\mathbf{K}$ by the quotient $\mathbf{H}_d(\mathbf{K}, \mathcal{R}) = \mathbf{Z}_d(\mathbf{K}, \mathcal{R}) / \mathbf{B}_d(\mathbf{K}, \mathcal{R})$.

If $\mathcal{R}$ is the *ring of integers* $\mathbb{Z}$, $\mathbf{H}_d(\mathbf{K},\mathbb{Z})$ is an abelian group and, according to the *fundamental theorem of finitely generated abelian groups* [10], admits a *primary decomposition*: $\mathbf{H}_d(\mathbf{K},\mathbb{Z}) \cong \mathbb{Z}^{\beta_d(\mathbb{Z})} \bigoplus_{q \text{ prime}} \left(\mathbb{Z}_{q^{k_1}} \oplus \cdots \oplus \mathbb{Z}_{q^{k_{t(d,q)}}}\right)$ for uniquely defined integer $\beta_d(\mathbb{Z})$, called the d^{th} *integral Betti number*, and integers $t(d,q) \geq 0$ and $k_i > 0$ for every prime number q. If $t(d,q) > 0$, the integers $q^{k_1}, \cdots, q^{k_{t(d,q)}}$ are called *torsion coefficients*, and they admit q as unique *prime divisor*. Intuitively, in dimension 0, 1 and 2, the integral Betti numbers count the number of connected components, the number of holes and the number of voids respectively. The torsion coefficients represent non-orientable twisting of different order of the shape. If $\mathcal{R}$ is a *field* $\mathbb{F}$, $\mathbf{H}_d(\mathbf{K},\mathbb{F})$ is a vector-space and decomposes into $\mathbf{H}_d(\mathbf{K},\mathbb{F}) \cong \mathbb{F}^{\beta_d(\mathbb{F})}$, where $\beta_d(\mathbb{F})$ is the d^{th} *field Betti number*. The field Betti numbers $(\beta_d(\mathbb{F}))_d$ are entirely determined by the characteristic of $\mathbb{F}$ and the inegral homology (see Section 5); hence, the integral homology is more informative than homology in $\mathbb{F}$.

A *filtration* of a complex is a function $f : \mathbf{K} \to \mathbb{R}$ satisfying $f(\tau) \leq f(\sigma)$ whenever $\tau \subseteq \sigma$. The sequence $[\sigma_i]_{i=1,\cdots,m}$ sorted according to increasing f values induces the sequence of inclusions $\emptyset = \mathbf{K}_0 \subsetneq \mathbf{K}_1 \subsetneq \cdots \subsetneq \mathbf{K}_{m-1} \subsetneq \mathbf{K}_m = \mathbf{K}$, $\mathbf{K}_i = \mathbf{K}_{i-1} \cup \{\sigma_i\}$, and the sequence of d-homology groups $0 = \mathbf{H}_d(\mathbf{K}_0,\mathcal{R}) \to \mathbf{H}_d(\mathbf{K}_1,\mathcal{R}) \to \cdots \to \mathbf{H}_d(\mathbf{K}_{m-1},\mathcal{R}) \to \mathbf{H}_d(\mathbf{K}_m,\mathcal{R}) = \mathbf{H}_d(\mathbf{K},\mathcal{R})$ connected by homomorphisms. When $\mathcal{R}$ is a field, the later sequence admits a decomposition in terms of intervals $\{(i,j)\}$, called an *indexed persistence diagram*, where a pair (i,j) is interpreted as a homology feature that *is born* at index i and *dies* at index j. Computing persistent homology consists in computing the interval decomposition and hence the persistence diagram. We refer to [10] for an introduction to homology and to [5] for an introduction to persistent homology.

We call the algorithmic problem of computing persistent homology with various coefficient fields *multi-field persistent homology*. Computing multi-field persistence allows us to infer a more informative description of the topology of a space, compared to persistence in a single field (see Section 5 for details). For a complex of size m, we know that the persistence diagram for any coefficient field contains at most m pairs. When computing multi-field persistent homology for r coefficient fields, denote by m' the total number of *distinct* pairs in all persistence diagrams. In practice, the fields are $\mathbb{Z}_{q_1}, \cdots, \mathbb{Z}_{q_r}$ for the r first prime numbers $q_1, \cdots, q_r$, where q_r is an upper bound on the prime divisors of the torsion coefficients of the integral homology of the space, which are usually small (see Section 5). The quantity m' satisfies $m' \leq r \times m$ but in practice we observe that $m' \approx m$. We design in the following an algorithm for the multi-field persistence problem whose complexity depends mostly on m'. It is however an interesting open problem to exhibit a "natural" example where m' is must larger than m and/or the prime divisors of the torsion coefficients are big (for a fix m).

3 Algorithm for Multi-field Persistent Homology

For clarity, we focus in this section on the persistent homology algorithm as presented in [5], which consists in a reduction to column echelon form (defined later) of a matrix. All other persistent homology algorithms are based on similar reductions, and our approach adapts directly to them. In the following, $\mathbb{Z}_n$ denotes the ring $(\mathbb{Z}_n, +, \times)$ for any integer $n \geq 1$. and $\mathbb{Z}_n^\times$ the subset of invertible elements for $\times$. If exists, we denote the inverse of $x \in \mathbb{Z}_n$ by x^{-1}.

In computer algebra, working modulo small prime numbers is usually desireable in order to reduce coefficient growth. Our work goes the otherway around: we introduce tools to reduce a family of r matrices with coefficients in the fields $\mathbb{Z}_{q_1}, \cdots, \mathbb{Z}_{q_r}$ respectively, by means of a single reduction of a matrix with coefficients in $\mathbb{Z}_{q_1 \cdots q_r}$. We give theoretical and experimental evidence that, for reasonable values of r, our algorithm is significantly more efficient than the brute-force approach consisting in reducing the r matrices separately.

Persistent Homology Algorithm: For an $m \times m$ matrix $\mathbf{M}$, denote by C_j the j^{th} column of $\mathbf{M}$, $1 \leq j \leq m$, and $C_j[k]$ its k^{th} coefficient. Let $\text{low}(j)$ denote the row index of the lowest non-zero coefficient of C_j. If the column j is entirely zero, $\text{low}(j)$ is undefined. We say that $\mathbf{M}$ is in *reduced column echelon form* if $\text{low}(j) \neq \text{low}(j')$ for every non-zero columns C_j and $C_{j'}$ with $j \neq j'$.

Let $\mathbf{K} = [\sigma_i]_{i=1\cdots m}$ be a filtered complex. Its boundary matrix $\mathbf{M}_\partial$ is the $m \times m$ matrix, with $\mathbb{F}$ coefficients, of the endomorphism ∂_* in the basis $\{\sigma_1, \cdots, \sigma_m\}$ of $\bigoplus_d \mathbf{C}_d(\mathbf{K}, \mathbb{F})$. The basis is ordered according to the filtration. It is a matrix with $\{-1, 0, 1\}$ coefficients, where 0 and 1 are the identities for $+$ and $\times$ in $\mathbb{F}$ respectively, and -1 is the inverse of 1 in $\mathbb{F}$. The persistent homology algorithm consists in a left-to-right reduction to column echelon form of $\mathbf{M}_\partial$: we denote by $\mathbf{R}$ the matrix we reduce, with columns C_j, which is initially equal to $\mathbf{M}_\partial$. The algorithm returns the *(indexed) persistence diagram*, which is the set of pairs $\{(\text{low}(j), j)\}$ in the reduced column echelon form of the matrix.

```
   Data: Boundary matrix $\mathbf{R} \leftarrow \mathbf{M}_\partial$, persistence diagram $\mathcal{P} \leftarrow \emptyset$
   Output: Persistence diagram $\mathcal{P} = \{(i, j)\}$
 1 for $j = 1, \cdots, m$ do
 2     while there exists $j' < j$ with $low(j') = low(j)$ do
 3         $k \leftarrow \text{low}(j)$;
 4         $C_j \leftarrow C_j - \big(C_j[k] \times C_{j'}[k]^{-1}\big) \cdot C_{j'}$;
 5     end
 6     if $C_j \neq 0$ then $\mathcal{P} \leftarrow \mathcal{P} \cup \{(\text{low}(j), j)\}$
 7 end
```

The reduced form of the matrix is not unique, but the pairs (i, j) such that $i = \text{low}(j)$ in the column echelon form are [5]. The algorithm requires $O(m^3)$ arithmetic operations in $\mathbb{F}$.

3.1 Modular Reconstruction for Elementary Matrix Operations:

We present a particular case of the *Chinese Remainder Theorem* [7]: *For a family $\{q_1, \cdots, q_r\}$ of r distinct prime numbers, there exists a ring isomorphism $\psi : \mathbb{Z}_{q_1} \times \cdots \times \mathbb{Z}_{q_r} \to \mathbb{Z}_{q_1 \cdots q_r}$. The isomorphisms ψ and ψ^{-1} can be computed in $O(r)$ arithmetic operations in $\mathbb{Z}_{q_1 \cdots q_r}$.*

Let $[r]$ refer to the set $\{1, \cdots, r\}$. For a family of r distinct prime numbers $\{q_1, \cdots, q_r\}$, and a subset of indices $S \subseteq [r]$, Q_S refers to $\prod_{s \in S} q_s$, and we write simply $Q = Q_{[r]}$. We define the function $\psi_S : \prod_{s \in S} \mathbb{Z}_{q_s} \to \mathbb{Z}_{Q_S}$ realizing the isomorphism of the Chinese Remainder Theorem for the subset $\{q_s\}_{s \in S}$ of primes, and we write simply ψ for $\psi_{[r]}$. For a family of elements $u_s \in \mathbb{Z}_{q_s}, s \in S$, we denote the corresponding $|S|$-uplet $(u_s)_{s \in S} \in \prod_{s \in S} \mathbb{Z}_{q_s}$.

Finally, we recall *Bezout's lemma* [7]: *For two integers a and b, not both* 0*, there exist integers v and w such that $va + wb = \gcd(a, b)$, the greatest common divisor of a and b, with $|v| < |b/\gcd(a, b)|$ and $|w| < |a/\gcd(a, b)|$. The Bezout's coefficients (v, w) can be computed with the extended Euclidean algorithm [7].*

Elementary Column Operations: We are given a family of distinct prime numbers $\{q_1, \cdots, q_r\}$, and their product $Q = q_1 \cdots q_r$. Let $\mathbf{M}_Q$ be a matrix with coefficients in the ring $\mathbb{Z}_Q$. Denoting $\psi^{-1} : \mathbb{Z}_Q \to \mathbb{Z}_{q_1} \times \cdots \times \mathbb{Z}_{q_r}$ the isomorphism of the Chinese Remainder Theorem, and $\pi_s : \mathbb{Z}_{q_1} \times \cdots \times \mathbb{Z}_{q_r} \to \mathbb{Z}_{q_s}$ the projection on the s^{th} coordinate, we call *projection of* $\mathbf{M}_Q$ *onto* $\mathbb{Z}_{q_s}$, denoted $\mathbf{M}_Q(\mathbb{Z}_{q_s})$, the matrix $\mathbf{M}_{q_s}$ with $\mathbb{Z}_{q_s}$ coefficients, obtained by applying $\pi_s \circ \psi^{-1}$ to each coefficient of $\mathbf{M}_Q$. Conversely, given r $(m \times m)$-matrices $\mathbf{M}_{q_1}, \cdots, \mathbf{M}_{q_r}$ with coefficients in $\mathbb{Z}_{q_1}, \cdots, \mathbb{Z}_{q_r}$ respectively, there exists a unique matrix $\mathbf{M}_Q$ with $\mathbb{Z}_Q$ coefficients such that for every s the projection of $\mathbf{M}_Q$ onto $\mathbb{Z}_{q_s}$ is $\mathbf{M}_{q_s}$. This is simply a matrix version of the Chinese remainder theorem. *Elementary column operations* on $\mathbf{M}_q$ with $\mathbb{Z}_q$ coefficients are of three kinds:

(i) exchange Col k and Col ℓ
(ii) multiply Col k by $-1 \in \mathbb{Z}_q$
(iii) replace Col k by (Col k)+ $x\times$(Col ℓ), for $x \in \mathbb{Z}_q$.

For an elementary column operation $(*)$ (i.e. an operation of type (i), (ii) or (iii) applied to columns k (and ℓ)), we denote by $(*) \circ \mathbf{M}_q$ the result of applying $(*)$ to $\mathbf{M}_q$. In this section, we introduce algorithms to run elementary column operations simultaneously on the matrices $(\mathbf{M}_{q_s})_{s=1,\cdots,r}$ by performing "partial column operations" on $\mathbf{M}_Q$. Specifically, for an elementary column operation $(*)$ and a subset of indices $S \subseteq [r]$, we call *partial column operation* on $\mathbf{M}_Q$ the operation transforming $\mathbf{M}_Q$ into $\mathbf{M}'_Q$ such that: for every $s \notin S$, the projection onto $\mathbb{Z}_{q_s}$ satisfies $\mathbf{M}_Q(\mathbb{Z}_{q_s}) = \mathbf{M}'_Q(\mathbb{Z}_{q_s}) = \mathbf{M}_{q_s}$ and for every $s \in S$, the projection onto $\mathbb{Z}_{q_s}$ satisfies $\mathbf{M}'_Q(\mathbb{Z}_{q_s}) = (*) \circ \mathbf{M}_{q_s}$.

As the correspondence $\psi : \mathbb{Z}_{q_1} \times \cdots \times \mathbb{Z}_{q_r} \to \mathbb{Z}_Q$ is a ring homomorphism, it satisfies the properties: $\psi(u_1, \cdots, u_r) + \psi(v_1, \cdots, v_r) \times \psi(w_1, \cdots, w_r) = \psi(u_1 + v_1 \times w_1, \cdots, u_r + v_r \times w_r)$ and we can compute addition and multiplication componentwise in $\mathbb{Z}_{q_1} \times \cdots \times \mathbb{Z}_{q_r}$ using addition and multiplication in $\mathbb{Z}_Q$. In order to compute partial column operations, we first introduce the set of *partial identities*, which are coefficients that allow us to proceed to the partial column operations of type (i) and (ii). Secondly, as the rings $\mathbb{Z}_{q_s}$ are fields, we need to compute the multiplicative inverse of an element, that is used as multiplicative coefficient x in elementary column operation (iii). As $\mathbb{Z}_Q$ is not a field, inversion is not possible, and we introduce the concept of *partial inverse* to overcome this difficulty. In the following, the term "arithmetic operation" refers to any operation $\{+, -, \times, \gcd(\cdot,\cdot), \cdot \bmod Q_S, \text{Extended Euclidean algorithm}\}$ on integer smaller than Q. Note they do not have constant time complexity for large Q.

Partial Identity and Partial Inverse: Given a subset of indices $S \subseteq [r]$, we define the *partial identities w.r.t. S*, denoted L_S, by $L_S = \psi(\delta_{1,S}, \cdots, \delta_{r,S})$ where the symbol $\delta_{t,S} \in \mathbb{Z}_{q_t}$ is equal to 1 if $t \in S$ and to 0 otherwise. For any $S \subseteq [r]$, the partial identity L_S can be constructed in $O(r)$ arithmetic operations in $\mathbb{Z}_Q$ by evaluating ψ on $(\delta_{1,S}, \cdots, \delta_{r,S})$. However, it is important to notice that if $S = [r]$, $L_{[r]} = \psi(1, \cdots, 1) = 1$, because ψ is a ring isomorphism, and L_r is computed in time $O(1)$.

Knowing the partial identities, we can implement the partial column operations (i) and (ii) for a set of indices S. Partial column operation (i) is implemented by replacing column k by (Col $k \times L_{[r]\setminus S} +$ Col $\ell \times L_S$) and column ℓ by (Col $\ell \times L_{[r]\setminus S} +$ Col $k \times L_S$). Partial column operation (ii) is implemented by multiplying column k by $L_{[r]} - 2 \times L_S$.

We define now the *partial inverse* of an element in the ring $\mathbb{Z}_Q$:

Definition 1 (Partial Inverse). *Given a set $S \subseteq [r]$ of indices, the* partial inverse *of $x = \psi(u_1, \cdots, u_r)$ with regard to S is the element $\overline{x}^S \in \mathbb{Z}_Q$:*

$$\overline{x}^S = \psi(\overline{u_1}^S, \cdots, \overline{u_r}^S), \quad \textit{with} \quad \overline{u_s}^S = \begin{cases} u_s^{-1} & \textit{if} \quad s \in S \textit{ and } u_s \in \mathbb{Z}_{q_s}^\times \\ 0 & \textit{otherwise} \end{cases}$$

Using elementary algebra (see [2] for details) we prove:

Proposition 2 (Partial Inverse Construction). *For $x = \psi(u_1, \cdots, u_r) \in \mathbb{Z}_Q$ and $S \subseteq [r]$,*

(1) $\gcd(x, Q_S) = Q_R$ *for some $R \subseteq S$ and for all $s \in S$, u_s is invertible in $\mathbb{Z}_{q_s}$ iff $s \notin R$; we denote $T = S \setminus R$.*
(2) The Bezout's identity for x and Q_T gives $vx + wQ_T = 1$, where v satisfies $v \bmod Q_T = \psi_T((u_s^{-1})_{s\in T})$
(3) $\overline{x}^S = \left[\psi_T((u_s^{-1})_{s\in T}) \times L_T \bmod Q\right] \in \mathbb{Z}_Q$, where L_T is the partial identity w.r.t T.

We deduce directly an algorithm to compute the partial inverse of x w.r.t S if Q_S is given: compute $Q_R = \gcd(x, Q_S)$ and $Q_T = Q_S/Q_R$, then v using the extended Euclidean algorithm and finally $\overline{x}^S = (v \bmod Q_T) \times L_T \bmod Q$. Computing the partial identity L_T requires $O(r)$ arithmetic operations in $\mathbb{Z}_Q$, but is constant if $T = [r]$, which happens iff $S = [r]$ and x is invertible in $\mathbb{Z}_Q$. Consequently, computing $\overline{x}^S$ requires $O(r)$ arithmetic operations in general, but only $O(1)$ arithmetic operations in the later case.

3.2 Modular Reconstruction for Multi-field Persistent Homology

Let $\mathbf{K}$ be a filtered complex with m simplices. Define $\mathbf{M}_\partial(\mathbb{Z}_{q_s})$ to be the $(m \times m)$ boundary matrix of $\mathbf{K}$ with $\mathbb{Z}_{q_s}$ coefficients. Define $\mathbf{M}$ to be the $(m \times m)$ matrix with $\mathbb{Z}_Q$ coefficients such that the projection of $\mathbf{M}$ onto $\mathbb{Z}_{q_s}$ is equal to $\mathbf{M}_\partial(\mathbb{Z}_{q_s})$, for all $s \in [r]$. Note that the matrices $\mathbf{M}$ and $\mathbf{M}_\partial(\mathbb{Z}_{q_s})$, for any s, are "identical" matrices in the sense that they contain 0, 1 and -1 coefficients at the same positions, where 0, 1 and -1 refer respectively to elements of $\mathbb{Z}_Q$ and $\mathbb{Z}_{q_s}$.

We reduce a matrix $\mathbf{R}$ which is initially equal to $\mathbf{M}$. Denote by C_j the j^{th} column of $\mathbf{R}$. Define $\text{low}(j, Q_S)$ to be the index of the lowest element of C_j such that $C_j[\text{low}(j, Q_S)] \mod Q_S \neq 0$. In particular, $\text{low}(j, q_s)$ is equal to the index of the lowest non-zero element of column j in the projection $\mathbf{R}(\mathbb{Z}_{q_s})$. After iteration j, we say that the columns $C_1, \cdots, C_j$ are *reduced.* We maintain, for every reduced column C_j, the collection of "lowest indices" i as a set $\mathcal{L}(j) = \{(i, Q_S)\}$ satisfying:

- For every $(i, Q_S) \in \mathcal{L}(j)$, $i = \text{low}(j, Q_S)$
- For every $(i, Q_S), (i', Q_{S'}) \in \mathcal{L}(j)$, either $i = i'$ and $S = S'$, or $i \neq i'$ and $S \cap S' = \emptyset$
- $\cup_{(i,Q_S)\in\mathcal{L}(j)} S = [r]$

The algorithm returns the set of triplets $\mathcal{P} = \{(i, j, Q_S)\}$ such that $i = \text{low}(j)$ in the column echelon form of the matrix $\mathbf{M}_\partial(\mathbb{Z}_{q_s})$ iff $s \in S$, or, equivalently, $(i, Q_S) \in \mathcal{L}(j)$ once C_j has been reduced. This is a compact encoding of the persistence diagrams of the filtered complex in persistent homology with all coefficient fields. We call it *multi-field persistence diagram.*

```
   Data: Matrix $\mathbf{R} = \mathbf{M}$
   Output: Multi-field persistence diagram $\mathcal{P} = \{(i, j, Q_S)\}$
1  for $j = 1, \cdots, m$ do
2      $Q_S \leftarrow Q_{[r]}$;
3      while $low(j, Q_S)$ is defined do
4          $k \leftarrow \text{low}(j, Q_S)$;      $Q_T \leftarrow Q_S / \gcd(C_j[k], Q_S)$ ;
5          while there exists $j' < j$ with $(i, Q_{T'}) \in \mathcal{L}(j')$ satisfying
6                  $[i = low(j, Q_S)$ and $\gcd(Q_{T'}, Q_T) > 1]$ do
7              $C_j \leftarrow C_j - \left(C_j[k] \times \overline{C_{j'}[k]}^T\right) \cdot C_{j'}$;
8              $Q_T \leftarrow Q_T / \gcd(Q_{T'}, Q_T)$;
9          end
10         if $Q_T \neq 1$ then $\mathcal{P} \leftarrow \mathcal{P} \cup \{(k, j, Q_T)\}$;      $Q_S \leftarrow Q_S / Q_T$;
11     end
12 end
```

The $\{\mathcal{L}(j)\}_j$ form an index table that we maintain implicitely. At iteration j of the `for` loop, we use Q_S for the product of all prime numbers $\prod_{s\in S} q_s$ for which the column j in $\mathbf{R}(\mathbb{Z}_{q_s})$ has not yet been reduced.

Correctness: First, note that all operations processed on $\mathbf{R}$ correspond to left-to-right elementary column operations in the matrices $\mathbf{R}(\mathbb{Z}_{q_s})$ for all $s \in [r]$. By definition of the partial inverse, the column operation in line 7 can only reduce the value of $\text{low}(j, Q_S)$. Moreover, one iteration of the `while` loop in line 3 either strictly reduces Q_S by dividing it by Q_T (in line 10) or set $(C_j[k] \mod Q_S)$ to zero, hence reducing strictly $\text{low}(j, Q_S)$. The later case happens when Q_T is set to 1 in line 8. Consequently, the algorithm terminates.

We prove recursively, on the numbers of columns, that each of the matrix $\mathbf{R}(\mathbb{Z}_{q_s})$ gets reduced to column echelon form. We fix an arbitrary field $\mathbb{Z}_{q_s}$: suppose that the $j-1$ first columns of $\mathbf{R}(\mathbb{Z}_{q_s})$ have been reduced at the end of iteration $j-1$ of the `for` loop in line 1. We prove that at the end of the j^{th} iteration of the `for` loop in line 1, the j first columns of the matrix $\mathbf{R}(\mathbb{Z}_{q_s})$ are reduced. Consider two cases. First suppose there is a triplet $(i, j, Q_T) \in \mathcal{P}$ for some $i < j$ and Q_T satisfying $q_s \mid Q_T$. This implies that the algorithm exits the `while` loop in line 5 with $q_s \mid Q_S$ (because by definition of Q_T, in line 4, $Q_T \mid Q_S$) and there is no $j' < j$ such that $[\text{low}(j', Q_{T'}) = \text{low}(j, Q_S)$ and $\gcd(Q_{T'}, Q_T) > 1]$. This in particular implies that there is no $j' < j$ such that $\text{low}(j', q_s) = \text{low}(j, q_s)$ and column j is reduced in $\mathbf{R}(\mathbb{Z}_{q_s})$.

Secondly, suppose that there is no such pair (i, j, Q_T) in $\mathcal{P}$, with q_s dividing Q_T. Consequently, during all the computation of the `while` loop in line 3, $q_s \mid Q_S$. When exiting this `while` loop, $\text{low}(j, Q_S)$ is undefined, implying in particular that $\text{low}(j, q_s)$ is undefined and column j of $\mathbf{R}(\mathbb{Z}_{q_s})$ is zero, and hence reduced.

4 Output-Sensitive Complexity Analysis

Arithmetic Complexity Model for Large Integers: During the reduction algorithm we perform arithmetic operations on big integers, for which we describe a complexity model [7]. Suppose that on our architecture, a memory word is encoded on w bits (on modern architectures, w is usually 64). Computer chips contains Arithmetic Logic Units that allow arithmetic operations on a 1-memory word integer in $O(1)$ machine cycles. Let the *length* of an integer z be defined by: $\lambda(z) = \lfloor \log_2 z/\mathsf{w} \rfloor + 1$, i.e. by the number of memory words necessary to encode z. We express the arithmetic complexity as a function of the length. For any positive integer z of length $\lambda(z) = \mathsf{B}$, operations in $\mathbb{Z}_z$ cost $\mathsf{A}_+(z) = O(\mathsf{B})$ for addition, $\mathsf{A}_\times(z) = O(M(\mathsf{B}))$ for multiplication and $\mathsf{A}_\div(z) = O(M(\mathsf{B}) \log \mathsf{B})$ for (extended) Euclidean algorithm, inversion and division, where $M(n)$ is a monotonic upper bound on the number of word operations necessary to multiply two integers of length B. By a result of [6], $M(\mathsf{B}) = O(\mathsf{B} \log \mathsf{B}\ 2^{O(\log^* \mathsf{B})})$, where $\log^* n$ is the iterated logarithm of n. In the following, we write $\mathsf{A}(z)$ for a bound on the complexity of arithmetic operations on integers smaller than z.

In the case of multi-field persistent homology, we are interested in the value of λ for an element in $\mathbb{Z}_Q$, $Q = q_1 \cdots q_r$, in the case where $\{q_1, \cdots, q_r\}$ are the first r prime numbers. We know [11] that $\ln Q < 1.01624 q_r$ and $q_r < r \ln(r \ln r)$ for $r \geq 6$. Consequently, $\lambda(Q) < \lfloor 1.46613 r \ln(r \ln r)/\mathsf{w} \rfloor + 1$. Note that $\lambda(Q) \ll r$ for $r \ln r \ll e^{\mathsf{w}}$, which is a reasonable assumption.

Complexity of the Modular Reconstruction Algorithm: Let $\mathbf{K}$ be a filtered complex of size m. The persistent homology algorithm described in Section 3, applied on $\mathbf{K}$ with coefficients in a field $\mathbb{F}$, requires $O(m^3)$ operations in $\mathbb{F}$. For a field $\mathbb{Z}_q$ these operations take constant time and the algorithm has complexity $O(m^3)$. The output of the algorithm is the persistence diagram, which has size $O(m)$ for any field.

For a set of prime numbers $\{q_1, \cdots, q_r\}$, let m' be the total number of distinct pairs in all persistence diagrams for the persistent homology of $\mathbf{K}$ with coefficient fields $\mathbb{Z}_{q_1}, \cdots, \mathbb{Z}_{q_r}$. We express the complexity of the modular reconstruction algorithm in terms of the size of its output (i.e. the multi-field persistence diagram of size m'), the number of fields r and the arithmetic complexity $\mathsf{A}(Q)$. First, note that, for a column j' in the reduced form of $\mathbf{R}$, the size of $\mathcal{L}(j')$ is equal to the number of triplets of the multi-field persistence diagram with death index j'; denote this quantity by $|\mathcal{L}(j')|$. Hence, when reducing column $j > j'$, the column $C_{j'}$ is involved in a column operation $C_j \leftarrow C_j + \alpha \cdot C_{j'}$ at most $|\mathcal{L}(j')|$ times. Consequently, reducing C_j requires $O(\sum_{j'<j} |\mathcal{L}(j')|) = O(m')$ column operations. There is a total number of $O(m \times m')$ column operations to reduce the matrix, each of them being computed in time $O(m \times \mathsf{A}(Q))$.

Computing the partial inverse of an element $x \in \mathbb{Z}_Q$ takes time $O(r \times \mathsf{A}(Q))$ in the general case, and only $O(\mathsf{A}(Q))$ if x is invertible in $\mathbb{Z}_Q$. The partial inverse of an element $x = C_j[k]$ is computed only if there is a pair $(k, Q_T) \in \mathcal{L}(j)$. This element is not invertible in $\mathbb{Z}_Q$ iff $|\mathcal{L}(j)| > 1$. There are consequently $O(|m' - m|)$ non-invertible elements x that are at index $\mathrm{low}(j, Q_T)$ in some column j, for some Q_T. If we store the partial inverses when we compute them, the total complexity for computing all partial inverses in the modular reconstruction algorithm is $O((m + r \times (m' - m) \times \mathsf{A}(Q))$. We conclude that the total cost of the modular reconstruction algorithm for multi-field persistent homology is $O(\left[r \times (m' - m) + m^2 m'\right] \times \mathsf{A}(Q)) = O(\left[r \times (m' - m) + (m')^3\right] \times \mathsf{A}(Q))$, while the brute-force algorithm, consisting in computing persistence separately for every field $\mathbb{Z}_{q_1}, \cdots, \mathbb{Z}_{q_r}$ requires $O(r \times m^3)$ operations.

Note that asymptotically in r, one arithmetic operation in $\mathbb{Z}_{Q_{[r]}}$ becomes more costly than r distinct arithmetic operations in $\mathbb{Z}_{q_1}, \cdots, \mathbb{Z}_{q_r}$, in which case the modular reconstruction approach developed in this article becomes worse than brute-force (even when m' and m are close). This however happens for extremely big values of r (see Section 5) and has no incidence on practical cases.

Remark: On all datasets considered in our experiments, we have found no example where m' was significantly bigger than m. However, it is unclear whether many "short-lived torsion" might appear in general. We prove in a long version of the paper [2] that this is not an issue, by giving a finer complexity analysis of the algorithm in terms of index persistence $|j - i|$ of the pairs (i, j) in the persistence diagram.

5 Experiments

In this section, we report on the performance of the modular reconstruction algorithm for multi-field persistent homology. Our implementation is in `C++`, and we use the `GMP` library for storing large integers. All timings are measured on a 64 bits Linux machine with 3.00 GHz processor and 32 GB RAM. All timings are averaged over 10 independent runs. We compute the persistent homology of Rips complexes [5] built on a variety of both real and synthetic datasets. We use the *compressed annotation matrix* implementation of persistence [1] for its

Data	$\lvert\mathcal{P}\rvert$	D	d	r	$\lvert\mathcal{K}\rvert$	T_1	R_1	T_{50}	R_{50}	T_{100}	R_{100}	T_{200}	R_{200}
Bud	49,990	3	2	0.09	$127 \cdot 10^6$	96.3	0.51	110.3	22.2	115.9	42.3	130.7	75.0
Bro	15,000	25	?	0.04	$142 \cdot 10^6$	123.8	0.41	143.5	17.8	150.2	34.0	174.5	58.5
Cy8	6,040	24	2	0.8	$193 \cdot 10^6$	121.2	0.63	134.6	28.2	139.2	54.6	148.8	102.2
Kl	90,000	5	2	0.25	$114 \cdot 10^6$	78.6	0.52	89.3	23.0	93.0	44.1	105.2	78.0
S3	50,000	4	3	0.65	$134 \cdot 10^6$	125.9	0.40	145.7	17.2	152.6	32.8	177.6	50.3

Fig. 1. Timings of the modular reconstruction algorithm vs brute-force

efficiency and stability. **Bud** is a set of points sampled from the surface of the *Stanford Buddha* in $\mathbb{R}^3$. **Bro** is a set of 5×5 *high-contrast patches* derived from natural images, interpreted as vectors in $\mathbb{R}^{25}$, from the Brown database (with parameter $k = 300$ and cut 30%) [3]. **Cy8** is a set of points in $\mathbb{R}^{24}$, sampled from the space of conformations of the cyclo-octane molecule [9], which is the union of two intersecting surfaces. **Kl** is a set of points sampled from the surface of the figure eight Klein Bottle embedded in $\mathbb{R}^5$. Finally **S3** is a set of points distributed on the unit 3-sphere in $\mathbb{R}^4$. Datasets are listed in Figure 1 with the size of points sets $|\mathcal{P}|$, the ambient dimension D and intrinsinc dimension d of the sample points (if known), the parameter r for the Rips complex and the size of the complex $|\mathcal{K}|$. The values T_r for $r \in \{1, 50, 100, 200\}$ refers to the running time of the modular reconstruction algorithm for the r first prime numbers, and R_r refers to the ratio between the brute-force approach and the modular reconstruction algorithm.

Interpretation of the Results: Surprisingly, we have observed that, on all experiments, the number of differences between persistence diagrams with various coefficient fields was extremely small. As a consequence, $m' - m$ can be considered as a very small constant in our experiments (≤ 10). We have also observed that these differences appeared for small prime numbers q_s.

Figure 1 presents the timings of the modular reconstruction approach for a variety of simplicial complexes ranging between 114 and 193 million simplices. We note that from $r = 1$ to $r = 200$ prime numbers, the time for computing multi-field persistence using the modular reconstruction approach only increases by 23 to 41%, when the brute-force approach requires about 200 times more time. This difference appears in the speedup expressed by the ratio R_r. For $r = 1$, the modular reconstruction approach is about twice slower than the standard persistent homology algorithm in one field, because modular reconstruction is a more complex procedure and deals, in our implementation, with `GMP` integers that are slower than the classic `int` used in the standard persistent homology algorithm. However, this difference fades away as soon as $r > 1$ and the modular reconstruction is significantly more efficient than brute-force: it is, in particular, between 50.3 and 102.2 times faster for $r = 200$.

Figures 2 and 3 present the evolution of the running time of the modular reconstruction approach and the brute-force approach for an increasing number of fields r (using the first r prime numbers). Persistence is computed for a Rips complex built on a set of 10000 points sampling a Klein bottle, which

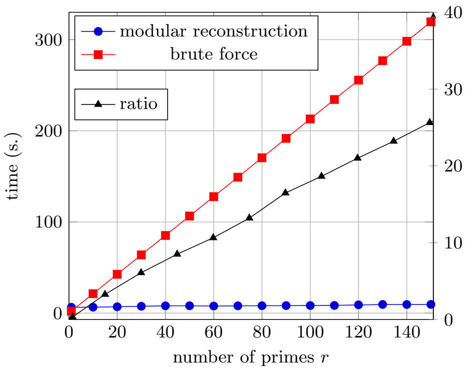


Fig. 2. Timings for the modular reconstruction algorithm and brute force

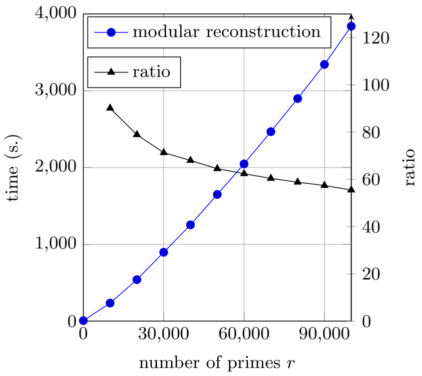


Fig. 3. Asymptotic behavior of modular reconstruction and brute force

contains torsion in its integral homology, resulting in a simplicial complex of 6.14 million simplices. We analyze the result in terms of the complexity analysis of Section 4. The quantity m is fixed and m' is fixed for $r \geq 2$. The complexity of the brute-force algorithm is $O(r \times m^3)$ and we indeed observe a linear behavior when r increases. The complexity of the modular reconstruction approach is $O(\left[r \times (m' - m) + m'^3\right] \mathsf{A}(Q_{[r]}))$. The part $r \times (m' - m)$ of the complexity is negligeable because $m' - m$ is extremely small. For medium values of r (≤ 150), like in Figure 2, the arithmetic complexity $O(\mathsf{A}(Q_{[r]}))$ increases very slowly because $\lambda(Q_{[r]}) = \lfloor \log_2 Q_{[r]}/\mathsf{w} \rfloor + 1$ increases slowly. We consequently observe a very slow increasing of the time complexity compare to the one of brute-force.

Figure 3 describes the asymptotic behavior of the modular approach, where the arithmetic operations become costly. We observe that the timings for the modular reconstruction approach follow a convex curve. The convexity comes from the growth of $\lambda(Q_{[r]})$, which is asymptotically $\Theta(r \log r)$) [11]. However, the increasing of the slope is very slow: all along this experiment, we have been unable to reach a value of r for which the modular approach is worse than the brute-force approach. For readability, the timings for the brute-force approach are implicetly represented through their ratio with the modular approach: all along the experiment, for $10000 \leq r \leq 100000$, the modular approach is between 55 and 90 times faster. Based on a linear interpolation of the timings for the brute-force approach, and a polynomial interpolation of the modular reconstruction timings, we expect the modular reconstruction to become worse than brute-force for a number of primes r bigger than 4.9 million. In the case of multi-field persistent homology however, there is no need to take r bigger than 200, because r is related to torsion coefficients (see Section 5), which are small in practice.

Back to Topology: Inference of Torsion. For a topological space $\mathbb{X}$, the *Universal Coefficient Theorem for Homology* [10] establishes the relationship between the homology groups $\mathbf{H}_d(\mathbb{X}, \mathbb{Z})$ with $\mathbb{Z}$ coefficients and the homology groups $\mathbf{H}_d(\mathbb{X}, \mathbb{Z}_q)$ with coefficients in the field $\mathbb{Z}_q$ (of characteristic q), for q prime. We use the following corollary:

Corollary 3 (Universal Coefficient Theorem [10].) *For $\beta_d(\mathbb{Z})$ and $\beta_d(\mathbb{Z}_q)$ the Betti numbers of $\mathbf{H}_d(\mathbb{X}, \mathbb{Z})$ and $\mathbf{H}_d(\mathbb{X}, \mathbb{Z}_q)$ respectively, and $t(j,q)$ the number of $\mathbb{Z}_{q^{k_i}}$ summands in the primary decomposition of $\mathbf{H}_j(\mathbb{X}, \mathbb{Z})$, we have:*

$$\beta_d(\mathbb{Z}_q) = \beta_d(\mathbb{Z}) + t(d,q) + t(d-1,q)$$

Suppose $\{q_1, \cdots, q_r\}$ are the first r prime numbers and q_r is a strict upper bound on the prime divisors of the torsion coefficients of $\mathbb{X}$. Consequently, according to Corollary 3, $\beta_d(\mathbb{Z}_{q_r}) = \beta_d(\mathbb{Z})$ for all dimensions d. Moreover, we know [10] that there is no torsion in 0-homology (i.e. $t(0,q) = 0$ for all primes q). Given the Betti numbers of $\mathbb{X}$ in all fields $\mathbb{Z}_{q_s}, 1 \leq s \leq r$, we deduce from Corollary 3 the recurrence formula $t(d, q_s) = \beta_d(\mathbb{Z}_{q_s}) - \beta_d(\mathbb{Z}_{q_r}) - t(d-1, q_s)$, from which we compute the value of $t(d,q)$ for every dimension d and prime q. For any dimension d, we consequently infer the integral Betti numbers and the number $t(d,q)$ of $\mathbb{Z}_{q^{k_i}}$ summands in the primary decomposition of $\mathbf{H}_d(\mathbb{X}, \mathbb{Z})$. We note however that the powers k_i from the decomposition remain unknown.

We describe in a long version of this article [2] a representation of multi-field persistence diagrams with torsion coefficients. We also describe an efficient algorithm to compute distances between them.

References

1. Boissonnat, J.-D., Dey, T.K., Maria, C.: The compressed annotation matrix: An efficient data structure for computing persistent cohomology. In: Bodlaender, H.L., Italiano, G.F. (eds.) ESA 2013. LNCS, vol. 8125, pp. 695–706. Springer, Heidelberg (2013)
2. Boissonnat, J.-D., Maria, C.: Computing persistent homology with various coefficient fields in a single pass. RR-8436, INRIA (December 2013)
3. Carlsson, G., Ishkhanov, T., Silva, V., Zomorodian, A.: On the local behavior of spaces of natural images. Int. J. Comput. Vision, 1–12 (2008)
4. Cohen-Steiner, D., Edelsbrunner, H., Harer, J.: Stability of persistence diagrams. Discrete & Computational Geometry 37(1), 103–120 (2007)
5. Edelsbrunner, H., Harer, J.: Computational Topology - an Introduction. American Mathematical Society (2010)
6. Fürer, M.: Faster integer multiplication. SIAM J. Comput. (2009)
7. Von Zur Gathen, J., Gerhard, J.: Modern Computer Algebra, 2nd edn. Cambridge University Press, New York (2003)
8. Maria, C.: Gudhi, simplex tree and persistent cohomology packages, `https://project.inria.fr/gudhi/software/`
9. Martin, S., Thompson, A., Coutsias, E.A., Watson, J.: Topology of cyclo-octane energy landscape. J. Chem. Phys. 132(23), 234115 (2010)
10. Munkres, J.R.: Elements of algebraic topology. Addison-Wesley (1984)
11. Rosser, J.B., Schoenfeld, L.: Approximate formulas for some functions of prime numbers 6, 64–94 (1962)
12. Zomorodian, A., Carlsson, G.E.: Computing persistent homology. Discrete & Computational Geometry 33(2), 249–274 (2005)

De-anonymization of Heterogeneous Random Graphs in Quasilinear Time

Karl Bringmann[1,⋆], Tobias Friedrich[2], and Anton Krohmer[2]

[1] Max Planck Institute for Informatics, Saarbrücken, Germany
[2] Friedrich-Schiller-Universität Jena, Germany

Abstract. There are hundreds of online social networks with billions of users in total. Many such networks publicly release structural information, with all personal information removed. Empirical studies have shown, however, that this provides a false sense of privacy — it is possible to identify almost all users that appear in two such anonymized network as long as a few initial mappings are known.

We analyze this problem theoretically by reconciling two versions of an artificial power-law network arising from independent subsampling of vertices and edges. We present a new algorithm that identifies most vertices and makes no wrong identifications with high probability. The number of vertices matched is shown to be asymptotically optimal. For an n-vertex graph, our algorithm uses n^{ε} seed nodes (for an arbitrarily small ε) and runs in quasilinear time. This improves previous theoretical results which need $\Theta(n)$ seed nodes and have runtimes of order $n^{1+\Omega(1)}$. Additionally, the applicability of our algorithm is studied experimentally on different networks.

1 Introduction

Imagine owning a large social network G_1 (like Facebook or Google+), and a competitor publishes an anonymized version of its own social network G_2, i.e. the graph structure without any additional labeling. This can happen on purpose or indirectly by APIs which are permitted to access the competitor's network or special access granted to advertising partners. If we identify vertices that are the same in both networks, we effectively deanonymize G_2 and gain new information, as there are connections in G_2 that do not exist in our social network G_1. This is valuable information for e.g. suggesting friends who are not yet connected in one of the networks. In this paper, we approach this social network reconciliation problem from an algorithm theory point of view.

Model. We model the above situation similar to [7] by assuming the existence of an underlying "real" social network $G = (V, E)$, which encodes whether two people know each other in the real world. Empirical studies showed that most social

⋆ Karl Bringmann is a recipient of the *Google Europe Fellowship in Randomized Algorithms*, and this research is supported in part by this Google Fellowship.

networks have a power-law degree sequence [10], which we model by an n-vertex Chung-Lu random graph [1, 4, 5]. Then we assume the online social networks G_1 and G_2 to be subsets of G: Every node of G exists in G_i independently with probability q_i, and every edge of G exists in G_i independently with probability p_i. Additionally, we randomly permute the graphs G_1 and G_2. We assume that there is a set of seed nodes $V_I \subseteq V$ which are known to match between G_1 and G_2 because they e.g. are persons of public interest. The algorithmic problem now is to identify as many vertices as possible from the given graphs G_1 and G_2 without making any wrong identifications (with high probability). We call a vertex *identifiable* if it survives in both graphs G_1 and G_2.

Theoretical results. We present an algorithm with the following guarantees. Here we let δ be the (expected) average degree of the graph G. See Section 2 for the technical assumptions about the parameters of the Chung-Lu random graph G and the parameters of the subsampling process.

Theorem 1. *Assume we are given the n^{ε} largest identifiable vertices as seed nodes for an arbitrary constant $\varepsilon > 0$. There is an algorithm that with high probability*[1] *makes no wrong identifications and successfully matches a fraction of $1 - \exp(-\Omega(p_1 p_2 q_1 q_2 \delta))$ of the identifiable vertices.*[2] *The algorithm runs in expected quasilinear runtime $\mathcal{O}(\delta n \log(n) / \min\{p_1, p_2\}^{\mathcal{O}(1/\varepsilon)})$.*

In the full version, we also show that this fraction of identified vertices is asymptotically optimal, since intuitively an $\exp(-\mathcal{O}(p_1 p_2 q_1 q_2 \delta))$ fraction of the vertices does not have any common neighbors in the two social networks. For constant p_1, p_2, q_1, q_2, the runtime is $\mathcal{O}(\delta n \log(n))$, which is within a factor $\log(n)$ of the expected number of edges $\Theta(\delta n)$ of G. Thus, our algorithm is the first with *quasilinear runtime*. This is crucial for handling large graphs. The best previous algorithms have a runtime of order $\mathcal{O}(\delta n \Delta^2)$ [9] or $\mathcal{O}(\delta n \Delta \log(\Delta))$ [7], where Δ is the maximum degree, which is typically of size $n^{\Omega(1)}$. Our approach also only needs n^{ε} seed nodes — previous algorithms with proven runtime and quality use at least $\Theta(n)$ seeds [7], although some heuristic approaches are also known to work with few seeds [9]. We remark that our algorithm is also successful with only $\mathcal{O}(\log(n)/p_1 p_2)$ seed nodes, but runs in quadratic time in this case.

Empirical results. We implemented a variant of our algorithm and applied it to different sets of networks. We match $\geqslant 89\%$ of the vertices in Chung-Lu graphs, preferential attachment graphs (PA), affiliation networks, and also subsampled real-world networks (Facebook, Orkut). All runs took less than 60 minutes on a single core, where previous results used compute clusters for an unreported amount of time [7]. In all cases, we need $\leqslant 0.03\%$ seed nodes to bootstrap our algorithm. This indicates that our approach translates to a wide variety of scale-free networks, even though we formally prove it on the Chung-Lu model.

[1] Throughout the paper, we say that a bound holds *with high probability* (w.h.p.) if it holds with probability at least $1 - n^{-c}$ for some $c > 0$.

[2] In the whole paper $\mathcal{O}(\cdot)$ and $\Omega(\cdot)$ hide any dependency on the power law exponent β of G. We always assume $2 < \beta < 3$.

Algorithm description. Starting with the seed nodes, we identify the remaining vertices by their *signatures*, an idea used in many algorithms for graph isomorphism. However, we have to cope with the additional complexity of the neighborhoods of identical vertices not being equal. We identify vertices when their signatures are strongly overlapping, and show an easy criterion for deciding whether two signatures stem from identical vertices. Using this criterion we make no errors with high probability and identify a large constant fraction of the vertices. We achieve a quasilinear runtime by locality sensitive hashing [6], which reduces the number of comparisons.

Applications. Anonymous copies of some social networks are available online. Several experimental papers describe how to find mappings between two online social networks [2, 9, 14, 15]. While some use the network structure alone [9], most of them exploit metadata like browser history [14], group memberships [16], writing style [12], semantic features of user aliases [11], or artificially added subgraphs [2]. The only theoretical result on this subject is by Korula and Lattanzi [7]. They identify 97% of the nodes on subsampled ($p_1, p_2 \geqslant \sqrt{22/\delta}$, $q_1 = q_2 = 1$) preferential attachment graphs [3], but need a linear amount of seed nodes and substantially more computing resources.

2 Preliminaries

Graph model. The model has two adjustable parameters: the exponent of the scale-free network β and the average degree δ. Depending on these two parameters, each node i has a weight w_i. For $n \in \mathbb{N}$ and weight distribution $\mathbf{w} = (w_1, \ldots, w_n) \in \mathbb{R}^n_{\geqslant 0}$ the Chung-Lu graph Chung-Lu$(n, \mathbf{w})$ is a graph on vertex set $V = [n]$ that contains each edge $\{u, v\}$, $u \neq v \in V$, with probability $p_{u,v} := \min\{w_u w_v / W, 1\}$, where $W := \sum_{v \in V} w_v$.

In order to simplify the presentation, we use a simple explicit weight distribution $w_i = \delta(n/i)^{1/(\beta-1)}$. Then $W = (1 + o(1))\frac{\beta-1}{\beta-2}\delta n = \Theta(\delta n)$, the expected average degree is $(1+o(1))\frac{2(\beta-1)}{\beta-2}\delta = \Theta(\delta)$, and we get a power law with exponent β [13]. We note that most of our results generalize to other weight distributions and even to weights drawn at random from "nice" distributions. We assume constant $2 < \beta < 3$, as real-world social networks have been observed to fulfill this. Moreover, we require $\delta \leqslant n^{o(1)}$, $p_1, p_2 \geqslant n^{-o(1)}$, and $q_1, q_2 = \Theta(1)$. We also assume that we know a lower bound for q_1, q_2, so that we know a constant factor approximation of n. For the sake of readability we even assume that we know n exactly. Finally, we require that $p_1 p_2 q_1 q_2 \delta$ is at least a sufficiently large constant (depending only on β).

De-anonymization. In our problem we have an underlying graph $G =$ Chung-Lu$(n, \mathbf{w})$ as defined above. This graph gets subsampled twice to generate two subgraphs: We put each node $v \in V := V(G)$ into V_1 independently with probability q_1. Then we put each edge $e \in E \cap \binom{V_1}{2}$ into E_1 independently with probability p_1 to form a graph $G_1 = (V_1, E_1)$. Now we randomly permute the nodes of G_1 to obtain a graph $\widetilde{G}_1$. We repeat this process with independent choices (and probabilities q_2, p_2) to form G_2 and $\widetilde{G}_2$.

We call two nodes $\widetilde{v}_i$ in $\widetilde{G}_i$, $i \in \{1,2\}$ *identical*, if they stem from the same node $v \in V$. The *identifiable* nodes are $V_\cap := V_1 \cap V_2$.

The input for the de-anonymization problem is $(\widetilde{G}_1, \widetilde{G}_2)$ and the task is to report pairs of vertices ("identified vertices") such that with high probability every identified pair is identical. We want to maximize the number of identified pairs. Note that the algorithm gets the randomly permuted graphs $\widetilde{G}_i$, but in the analysis we usually talk about the graphs G_i for the sake of readability. We write $\deg_i(v)$ for the degree of vertex $v \in V_i$ in G_i and $N_i(v)$ for its neighborhood in graph G_i, $i \in \{1,2\}$.

3 Estimating Weights and Edge Probabilities

In this section we show how to compute upper and lower bounds for the weight w_v of any vertex v based on the degree $\deg_i(v)$, $i \in \{1,2\}$. This also yields bounds for the edge probabilities $p_{u,v}$ for any vertices u and v. These bounds hold with high probability. Then we argue that our subsequent de-anonymization algorithms can use these computed bounds and still assume that all edges of G_i and G were sampled independently as if these graphs were not looked at before (where we used G_i as a short term for both graphs G_i, $i \in \{1,2\}$).

For the sake of readability we assume that the parameters n, β, p_1, and p_2 are known to the algorithm. However, it would be easy to also estimate these parameters with small error, and run our subsequent algorithms with these approximations. For a sketch of this, we note that we can estimate β from the degree distributions in G_i, similar to what we do for individual weights in this section. Moreover, we can estimate p_2 (and p_1, respectively) by dividing the number of edges that appear in $G_1[V_I] \cap G_2[V_I]$ by the number of edges in $G_2[V_I]$ $(G_1[V_I])$.

Afterwards we can run the method presented in this section to estimate the individual weights w_v and W. Additionally, one could estimate δ (e.g. from W and β) and q_1/q_2 (e.g. from $|V_1|/|V_2|$), but our algorithms do not need them. Note that in our model it is hard to estimate the parameters q_1, q_2.

The degree of each vertex v in G_i is composed of a random decision for each other node u, namely whether it is connected to v in the original graph G and a random decision whether this edge is present in the subsampled graph. In total,

$$\deg_i(v) \sim \sum_{u \in V \setminus \{v\}} \mathrm{Ber}\left(p_i q_i \cdot \min\left\{\frac{w_v w_u}{W}, 1\right\}\right),$$

if node v survives in G_i. By a Chernoff bound, we see that this degree is concentrated. This allows to compute intervals for the weights w_v for all v in G_i.

Lemma 1. *Let $i \in \{1,2\}$. Given $\deg_i(v)$ (and p_i and an approximation of n) we can compute $0 \leqslant \underline{w}_v \leqslant \overline{w}_v$ such that w.h.p. for all $v \in V$ we have*

1. $\underline{w}_v \leqslant q_i w_v \leqslant \overline{w}_v$,

2. $\overline{w}_v \leqslant \mathcal{O}\big(q_i w_v + \frac{1}{p_i}\log n\big)$, *and*
3. $\underline{w}_v \geqslant \Omega(q_i w_v) - \mathcal{O}\big(\frac{1}{p_i}\log n\big)$.

In a similar fashion, we can compute a bound on $q_i^2 \cdot W$.

Lemma 2. *Let $i \in \{1, 2\}$. Given G_i, we can compute $\underline{W}$ such that $\underline{W} \leqslant q_i^2 W \leqslant (1 + o(1))\underline{W}$ holds with high probability.*

The proof of Lemma 1 and most other proofs in the remainder can be found in the full version. Plugging the estimated weights into the edge probability formula allows us to compute bounds on the edge probabilities. It is worth noting that although the estimations on $\overline{w}_v$ and $\underline{W}$ give a result depending on q_i, the computed upper bound $\overline{p}_{uv}$ on the edge probabilities is oblivious to q_i.

Corollary 1. *For any $u, v \in V_i$ we can compute bounds $\overline{p}_{u,v} := \min\{\overline{w}_u\overline{w}_v/\underline{W}, 1\}$ such that w.h.p. we have*

$$p_{u,v} \leqslant \overline{p}_{u,v} \leqslant \mathcal{O}\Big(p_{u,v}\Big(1 + \frac{\log n}{p_i q_i w_u}\Big)\Big(1 + \frac{\log n}{p_i q_i w_v}\Big)\Big).$$

In particular, for $w_u, w_v = \Omega(\frac{1}{p_i q_i}\log n)$ we have $\overline{p}_{u,v} = \mathcal{O}(p_{u,v})$.

Corollary 1 allows to compute estimations for all edge probabilities with certain guarantees that hold with high probability. We want to use these estimations in the subsequent algorithms without losing the independence of the edges, i.e., in the subsequent algorithms we want to assume that the (edges of the) graphs G_i were not revealed yet, although we already computed bounds on the weights based on the degrees in G_i. In order to see that this might be a problem, assume that throughout a proof we reveal edges of a node v. However, once we have seen $\deg_i(v)$ edges, we know that there can be no other edge anymore, which violates our intuition of having independent edges.

To solve this technical problem, we model our weight estimation method as an *adaptive adversary*, which knows the parameters $p_1, p_2, q_1, q_2, w_1, \ldots, w_n, G_1$ and G_2, and reports estimations $\overline{p}_{u,v}$ for all $u, v \in V$ that fulfill the guarantees in Corollary 1 w.h.p. (over the randomness of the instance generation). The subsequent de-anonymization algorithms are then designed such that they assume to get edge probability estimations by our above method (or an adversary) that fulfill the said guarantees but are otherwise arbitrary. Then they may still assume that the random graphs G_i are not revealed. The details of this can be found in the full version of this paper.

4 Matching Phase

In the matching phase we assume that we know the identity of some vertices $V_I \subseteq V$ containing the $h = \Omega(\log(n)/p_1p_2)$ highest weight nodes (that survive in both G_1, G_2), and show how to identify most of the remaining vertices based on these initial nodes. Observe that the adaptive adversary model allows us to assume that all edges are independently present with their respective probability $p_{u,v}$.

4.1 The Y-Test

Denote by V_I the thus far identified vertices. Then for every unidentified vertex v in G_i we consider its *signature* $S_i^v := N_i(v) \cap V_I$. Unlike in the Graph Isomorphism problem, in our case signatures of identical vertices are not equal. However, for identical vertices the signatures S_1^v, S_2^v should be similar sets, while for non-identical vertices $u \neq v$ the signatures S_1^u, S_2^v should have small intersection. One contribution of our work is the test presented in this section, which allows to check whether two nodes are identical based on their signatures. This test never identifies two non-identical vertices (w.h.p.) and it identifies most vertices once sufficiently many of their neighbors are identified.

Let $v_1, v_2 \in V \setminus V_I$ and $u \in V_I$. Consider all possibilities of the edges $\{v_1, u\} \in E_1$ and $\{v_2, u\} \in E_2$ being present or not. We denote by A_u the event that both of these edges are present, by B_u^i the events that exactly one edge is present in G_i, and by C_u the remaining case. Based on these cases we now define

$$Y_u = Y_u^{v_1,v_2} := \begin{cases} \frac{1}{2\overline{p}_{v_1,u}}, & \text{if } u \in S_1^{v_1} \cap S_2^{v_2} \ (A_u) \\ 1 - \frac{p_2}{2}, & \text{if } u \in S_1^{v_1} \text{ and } u \notin S_2^{v_2} \ (B_u^1) \\ 1 - \frac{p_1}{2}, & \text{if } u \notin S_1^{v_1} \text{ and } u \in S_2^{v_2} \ (B_u^2) \\ 1, & \text{otherwise } (C_u). \end{cases}$$

and $Y := \prod_{u \in V_I} Y_u$. Intuitively, Y_u encodes the evidence of $v_1 = v_2$ given the connections to the identified node u; a common neighbor (A_u) has a large positive evidence, $Y_u > 1$, while a node u connected to only one of the two (B_u^1, B_u^2) has a small negative evidence, $Y_u < 1$. Note that when $v_1 = v_2$ we have $\overline{p}_{v_1,u} \approx \overline{p}_{v_2,u}$, so in the case (A_u) having one of the estimates turns out to be sufficient. The technical factor $1/2$ is needed later for some tail bounds.

We claim that Y is typically small for non-identical $v_1 \neq v_2$ and can be large (if V_I contains sufficiently many neighbors of v_1) if $v_1 = v_2$. In particular, we can test whether $v_1 = v_2$ by testing $Y > n^c$ for some appropriate constant $c > 0$. We call this the Y-test. This intuition is proven by the following lemmas. First we show that Y is not too large if $v_1 \neq v_2$ (w.h.p.). To this end, we verify $\mathbb{E}[Y_u] \leqslant 1$, then the statement follows from independence of the edges and Markov's inequality.

Lemma 3. *For any $v_1 \neq v_2 \in V \setminus V_I$ and $t > 0$ we have* $\Pr[Y > t] \leqslant 1/t$.

The next lemma can be used to show that our test allows to identify the two copies of v if we have already identified enough low-degree neighbors of v. We call the high-degree neighbors u "bad nodes" as they result in an estimated connection probability of $\overline{p}_{v,u} = \Omega(1)$.

Lemma 4. *Let V_I be any set of identified vertices, and consider an unidentified $v \in V_\cap \setminus V_I$. Let $B \subseteq V_I$ ("bad nodes") be the vertices u with $p_{u,v} \geqslant b > 0$, b being a sufficiently small constant. Assume that for $c > 0$ we have*

$$\sum_{u \in V_I} p_{u,v} \geqslant \Omega\big(\tfrac{1}{p_1 p_2} c \log n + |B|\big)$$

with a sufficiently large hidden constant. Then we have $\Pr[Y^{v,v} > n^c] \geqslant 1 - n^{-c}$.

For a vertex v with small weight $w_v = n^{o(1)}$ the above lemma does not apply and we have to take a closer look at $Y^{v,v}$.

Lemma 5. *Let $c > 0$ and consider an unidentified vertex $v \in V_\cap \setminus V_I$ with $w_v \leqslant n^{o(1)}$. Let $T \subseteq V_I$ be a set of identified vertices with $p_{u,v} = \Theta(\varepsilon)$ for all $u \in T$ and some $\varepsilon > 0$. Assume that $\mu := p_1 p_2 \varepsilon |T|$ is at least a sufficiently large constant (depending only on c and β). Then we have*

$$\Pr[Y^{v,v} > n^c] \geqslant 1 - n^{-c} - \exp(-\Omega(\mu)).$$

4.2 The Algorithm

We use the test developed in the last section as follows. As we build an algorithm that w.h.p. never identifies non-identical vertices, we can again write this algorithms in terms of the graphs G_1, G_2, but it can easily be translated to the randomly permuted graphs $\widetilde{G}_1, \widetilde{G}_2$.

Our algorithm gets as input the graphs G_1, G_2 and an initial set V_I of identified vertices containing the h highest weight vertices. Then in every round the algorithm compares all pairs v_1, v_2 of unidentified vertices. One comparison consists of a Y-test, i.e., we compute Y^{v_1,v_2} and test whether it is at least n^c, where $c > 0$ a constant. If this is the case, then we identify v_1 and v_2. The algorithm terminates after the first round in which no new vertex is identified.

Note that this algorithm is oblivious to the q_i's, as it only considers edges to nodes that are already identified, and thus survive in both subsampled graphs.

We will see that it suffices to run this algorithm for $\mathcal{O}(\log n)$ rounds to identify most of the vertices. As Y^{v_1,v_2} can be computed in time $\mathcal{O}(\deg_1(v_1) + \deg_2(v_2))$, the immediate runtime of this algorithm is $\mathcal{O}(nm \log n)$, where m is the total number of edges in G_1 and G_2, which is $\mathcal{O}(\delta n)$ with high probability. We will see in Section 5 how to decrease this to quasilinear runtime.

Algorithm 1. De-anonymization using Y-tests

```
Input: graphs $G_1, G_2$, identified vertices $V_I \supseteq \{1, \dots, h\}$
  for $r = 1, 2, \dots, \mathcal{O}(\log n)$ do
      for all $v_1, v_2 \in V \setminus V_I$ do
          if $Y^{v_1,v_2} > n^c$ then
              $V_I := V_I \cup \{v_1, v_2\}$.                ▷ We identified $v_1 = v_2$
```

Using Lemma 3 it is easy to see that Algorithm 1 never identifies any non-identical vertices. Note that choosing $c > 2$ yields error probability $o(1)$.

Lemma 6. *Algorithm 1 does not identify any two non-identical vertices with probability at least $1 - \mathcal{O}(n^{2-c} \log n)$.*

4.3 Quality Analysis

It remains to show that the algorithm identifies most vertices. We do this by examining the propagation of identified vertices in the graphs. For this, we define $L_j := \{2^{j-1}, \ldots, 2^j - 1\}$, $j = 1, \ldots, \log n$ as the j-th layer of vertices, and let $\tilde{L}_j \subseteq L_j$ be the set of vertices from layer j that survive in both graphs. If $|L_j| = \Omega(\log n)$, then w.h.p. by a Chernoff bound we have $|\tilde{L}_j| \geqslant \Omega(q_1 q_2 |L_j|)$.

We proceed in three steps. First, we show that given the seed nodes, there will be some layer k that gets identified with high probability. We choose k such that the estimated edge probability $\overline{p}_{v,h}$ of every vertex $v \in L_k$ with vertex h (the h-th highest weight vertex) is at most a sufficiently small constant, and k is minimal with this property. We can compute that $|L_k| = \Omega(n^{3-\beta}/\log n)$, meaning that $|\tilde{L}_k| = \Theta(q_1 q_2 |L_k|)$. In the first step of the analysis of our algorithm we show that after round 1 layer $\tilde{L}_k$ is identified with high probability.

In the second step, we show that from there on we identify one more layer each round, i.e., after round r we have identified layer $\tilde{L}_{k+r-1}$. This, however, cannot hold w.h.p. once the weights drop below $\mathcal{O}(\text{polylog}\, n)$. Instead, each vertex $v \in \tilde{L}_j$, $j > k$ is identified after round $j - k + 1$ with probability at least $1 - \alpha_j \geqslant 1 - \exp(-\Omega(p_1 p_2 q_1 q_2 \delta))$. This holds independently of the other vertices in L_j and of the edges from vertices above layer L_j to vertices below layer L_j or layer L_j itself. This way we identify most of the vertices in the layers above k in at most $\log(n) - k + 1$ rounds. We remark that these vertices could be identified already earlier, but we claim that they are identified at the latest after round $j - k + 1$ (with the mentioned probability).

In the third step, we show that after round $\log(n) - k + 2$ all high-degree vertices in layers below $\tilde{L}_k$ are identified with high probability. As the number of such vertices is small, this third step is not necessary for the conclusion that the algorithm identifies a large fraction of all identifiable vertices — it proves, however, the intuition that this algorithm identifies all vertices with sufficiently high weight ($\log^{\Omega(1)}(n)$) with high probability. We omit this third step in this extended abstract.

First step. Initially we know the identity of a set V_I of vertices containing the h highest weight nodes that survive in both graphs. We let $h = \gamma^2 \frac{1}{p_1 p_2} \log n$ where γ is a sufficiently large constant. Let $\ell := \gamma \frac{1}{p_1 p_2} \log n$. Choose a layer L_k such that for any $v \in L_k$ we have $p_{v,\ell} \approx b$, where $b = \Theta(1)$ is the constant from Lemma 4, so that we have $|B| = \mathcal{O}(\ell) = \mathcal{O}(\gamma \frac{1}{p_1 p_2} \log n)$ *bad nodes.* For our weight distributions one can show that $p_{v,h} = p_{v,\ell} \cdot (\ell/h)^{1/(\beta-1)} = \Theta(\gamma^{-1/(\beta-1)})$. Hence, any node $1 \leqslant u \leqslant h$ has $p_{v,u} \geqslant p_{v,h} = \Theta(\gamma^{-1/(\beta-1)})$. Summing up over $1 \leqslant u \leqslant h$, we have $\sum_{u \in V_I} p_{v,u} \geqslant \Omega(\gamma^{2-1/(\beta-1)} \frac{1}{p_1 p_2} \log n)$. Since $\gamma^{2-1/(\beta-1)} = \gamma^{1+\Omega(1)}$ and γ is sufficiently large, for any arbitrarily large hidden constant we have

$$\sum_{u \in V_I} p_{v,u} \geqslant \Omega((\gamma + c) \tfrac{1}{p_1 p_2} \log n) = \Omega(\tfrac{1}{p_1 p_2} c \log n + |B|),$$

which proves that the assumption of Lemma 4 is fulfilled and we identify v in the first round with high probability.

Second step. Consider any following level $k < j \leqslant \log n - \Omega(\log\log n)$ (with sufficiently large hidden constant) and let $v \in \tilde{L}_j$. We prove by induction that v is identified in round $j-k+1$ with high probability. By induction hypothesis, every vertex $u \in \tilde{L}_{j-1}$ is identified after round $j-k$ with high probability. The probability of v to connect to a vertex $u \in L_{j-1}$ is $p_{u,v} = \min\{w_v w_u/W, 1\}$. Plugging in $w_v, w_u = \Theta(\delta(n/2^j)^{1/(\beta-1)})$ yields $p_{u,v} = \Theta(\varepsilon)$ for $\varepsilon := \delta n^{(3-\beta)/(\beta-1)}\, 2^{-2j/(\beta-1)}$ and $\overline{p}_{u,v} = \mathcal{O}\big(p_{u,v} + \frac{1}{\delta n}\log^2 n\big)$. Note that since $j \leqslant \log n - \Omega(\log\log n)$ we have $w_v \geqslant \log^{\Omega(1)} n$ so that $\overline{p}_{u,v} = \mathcal{O}(p_{u,v})$. We can apply Lemma 4 with $|B| \leqslant \mathcal{O}(\frac{1}{p_1 p_2}\log n)$ (since the number of bad vertices is at most the number of bad vertices for layer L_k). Considering only the edges to $V_I \cap \tilde{L}_{j-1}$ and using $|\tilde{L}_{j-1}| = \Omega(q_1 q_2 2^j)$ we obtain

$$\sum_{u \in V_I} p_{u,v} \geqslant \Omega\big(q_1 q_2 2^j \delta n^{(3-\beta)/(\beta-1)}\, 2^{-2j/(\beta-1)}\big) = \Omega\big(q_1 q_2 \delta \log^{\Omega(1)} n\big),$$

which is larger than $\Omega(\frac{1}{p_1 p_2}\log n)$ since $p_1 p_2 q_1 q_2 \delta$ is at least a sufficiently large constant. Hence, Lemma 4 implies that we identify all vertices in $\tilde{L}_j$ with high probability.

For $\log n - o(\log n) \leqslant j \leqslant \log n$, so that $w_v = n^{o(1)}$, we instead use Lemma 5 to show that any vertex $v \in \tilde{L}_j$ is identified after round $j-k+1$ with probability at least $1-\alpha_j \geqslant 1 - \exp(-\Omega(p_1 p_2 q_1 q_2 \delta))$. We again consider the edges of v into $T := V_I \cap \tilde{L}_{j-1}$ and obtain

$$\mu := p_1 p_2 \varepsilon |T| = \Omega\big(p_1 p_2 q_1 q_2 2^j \delta n^{(3-\beta)/(\beta-1)}\, 2^{-2j/(\beta-1)}\big) = \Omega(p_1 p_2 q_1 q_2 \delta).$$

Hence, the assumption of Lemma 5 amounts to $p_1 p_2 q_1 q_2 \delta$ being at least a sufficiently large constant, and we identify each vertex in $\tilde{L}_j$ with probability at least $1 - \alpha_j := 1 - \exp(-\Omega(\mu)) - n^{-c} \geqslant 1 - \exp(-\Omega(p_1 p_2 q_1 q_2 \delta))$.

5 Quasilinear Runtime

Algorithm 1 in its pure form takes quadratic time, as we have seen in the last section. In this section we show how to decrease its runtime to quasilinear using locality sensitive hashing [6]. We assume to have identified the $h = n^{2\varepsilon}$ highest weight vertices for any constant $\varepsilon > 0$.

The basic idea for speeding up the algorithm is to reduce the number of tested pairs v_1, v_2. To this end, in every round we choose a random permutation π of V_I. For a vertex $v \in V \setminus V_I$ in G_i consider the vertices in V_I that have a small estimated probability to connect to v, $T_v := \{u \in V_I \mid \overline{p}_{v,u} \leqslant n^{-\varepsilon}\}$. We compute the first C/ε vertices $(u_1, \ldots, u_{C/\varepsilon}) =: M_v^i$ in $N_i(v) \cap T_v$ with respect to the order π (for some constant $C \geqslant 2$ to be fixed later). Note that M_v^i can be computed in constant time, if we permute the graphs $G_1[V_I]$ and $G_2[V_I]$ with respect to π (and store a version containing only the edges in $\bigcup_v T_v$), so that the first neighbor of v is simply the first entry of its adjacency list. In the analysis we show that for a so-called *good* vertex $v \in V_\cap$ the sets M_v^1, M_v^2 are equal

Algorithm 2. Fast de-anonymization using Y-tests

```
Input: graphs $G_1, G_2$, identified vertices $V_I \supseteq \{1, \dots, n^{2\varepsilon}\}$
  for $r = 1, 2, \dots, \Theta(p)^{-\Theta(1/\varepsilon)} \log n$ do
    choose a random permutation $\pi$ of $V_I$
    for all $v \in V \setminus V_I$ and $i \in \{1, 2\}$ do
      if $|N_i(v) \cap T_v| \geqslant C/\varepsilon$ then
        compute the set $M_v^i$ of the first $C/\varepsilon$ vertices in $N_i(v) \cap T_v$ w.r.t. $\pi$
        hash $(v, i)$ at $M_v^i$
    for all hash collisions of the form $(v_1, 1)$ and $(v_2, 2)$ do
      if $Y^{v_1, v_2} > n^c$ then
        $V_I := V_I \cup \{v_1, v_2\}$.                    ▷ We identified $v_1 = v_2$
```

with probability $\Theta(p_1 + p_2)^{\Theta(1/\varepsilon)}$, while for non-identical vertices $v_1 \neq v_2$ these sets are equal with probability at most $1/n$. Thus, we may hash v at M_v^i (with a perfect hash function that produces collisions only if the corresponding hash values are equal) and test vertices v_1, v_2 only if they form a hash collision. Then in expectation we test $\mathcal{O}(n)$ pairs of vertices per round. More precisely, one can show that these tests take expected time $\mathcal{O}(m)$ per round, where m is the total number of edges in G_1 and G_2. Everything else we do in one round also runs in time $\mathcal{O}(m)$. Note that in expectation we have $m = \mathcal{O}((p_1 + p_2)\delta n) \leqslant \mathcal{O}(\delta n)$.

As we will see in the full version, roughly the same quality analysis as in the last section goes through, with the necessary number of rounds growing to $\Theta(\min\{p_1, p_2\})^{-\Theta(1/\varepsilon)} \log n$. As $\varepsilon > 0$ is a constant, this is $\mathcal{O}(\min\{p_1, p_2\}^{-\mathcal{O}(1)} \log n)$. Furthermore, by the same arguments as in the last section, Algorithm 2 makes no wrong identifications w.h.p. (intuitively, it only does a subset of the Y-tests of Algorithm 1, but since we let it run for a few more rounds the error probability grows to $\mathcal{O}(n^{2-c} \min\{p_1, p_2\}^{-\mathcal{O}(1)} \log n)$). In total we get an expected runtime of $\mathcal{O}(\min\{p_1, p_2\}^{-\mathcal{O}(1)} \delta n \log n)$.

6 Experiments

The focus of this paper lies on the theoretical algorithm analysis. To check our theory for robustness, however, we conducted a preliminary empirical study. We implemented a variant of the fast algorithm single threaded in C++. The source code is available upon request. The experiments were run on a single computer with Dual Xeon CPU E5-2670 and 128 GB RAM.

We evaluated the algorithm on Chung-Lu graphs as shown in Table 1. The results indicate that our quality bounds hold up well in practice, as we identify 95% of the nodes with as little as 0.008% seeds (80 nodes). Similarly, the runtime follows our asymptotic bounds which allows for deanonymizing large graphs (2 million nodes) on a single core, whereas previous approaches would typically require a computing cluster due to their polynomial runtime.

Finally, we investigated the robustness of our algorithm with respect to changes to the underlying graph model. We ran it on different random graph

Table 1. Performance of our de-anonymization algorithm on various graphs

Graph Model								
Name	n	m	$p_1 \cdot p_2$	$q_1 \cdot q_2$	Seeds	Recall	Prec.	Runtime
Chung-Lu	2M	80M	0.25	0.72	80	0.95	1	29 min.
Pref. Attachment	1M	20M	0.25	1	200	0.95	1	9 min.
Affiliation Network	60K	8M	–	1	50	0.83	0.99	56 sec.
Facebook	63K	1.5M	0.58	1	50	0.50	0.95	6 sec.
Orkut	3M	117M	0.56	0.81	1000	0.89	0.88	54 min.

models (Preferential Attachment [3], Affiliation Networks[3] [8]) and even subsampled real graphs (Facebook, Orkut)[4]. We point out that [7] also performed experiments on Facebook and Affiliation Networks, achieving slightly better recall (0.6 and 0.9, respectively). However, they typically use 10% of the networks as seeds; and they do not report on their runtimes and machines.

In all cases, our algorithm was able to extend the small set of identified seed nodes to a linear fraction of the entire graph; while making comparatively few errors. This indicates that even though our proofs rely on the topology of the Chung-Lu model (e.g. independent edge probabilities), the algorithm performs reasonably well in practice.

7 Conclusion

We presented a new method for de-anonymizing scale-free networks with two crucial improvements compared to previous work: (i) faster runtime and (ii) less required a-priori knowledge.

While all previous algorithms have a runtime of $\Omega(n\Delta)$, our new algorithm runs in quasilinear time. This improvement is not only asymptotical: Recent experiments of Korula and Lattanzi [7] required large compute clusters, whereas our algorithm can handle graphs with millions of vertices in less than an hour on off-the-shelf hardware. The quasilinear runtime is achieved by a variant of locality sensitive hashing. We believe that this technique can be used in future work to speed up other matching and graph isomorphism algorithms.

Our second contribution is a rigorous proof that much fewer seed nodes suffice for de-anonymizing subsamples of a common model of scale-free networks. Our approach needs only n^ε seed nodes, while all previous algorithms with proven runtime and quality use $\Theta(n)$ seed nodes. The analysis is based on a new weight estimation scheme relative to an adaptive adversary, which appears to be useful also for analyzing other algorithms on Chung-Lu graphs. Our result shows that de-anonymization is possible with few seed nodes, which is important for practical attacks on anonymized networks.

[3] In this model, a bipartite graph of users and interests is constructed; and two users are connected if they share an interest. To create two subsampled graphs, each interest is deleted independently with probability 0.25 in both graphs.

[4] `http://snap.stanford.edu/data/`

References

[1] Aiello, W., Chung, F., Lu, L.: A random graph model for massive graphs. In: 32nd Annual ACM Symposium on Theory of Computing (STOC), pp. 171–180 (2000)

[2] Backstrom, L., Dwork, C., Kleinberg, J.: Wherefore art thou r3579x?: Anonymized social networks, hidden patterns, and structural steganography. In: 16th International Conference on World Wide Web (WWW), pp. 181–190 (2007)

[3] Barabási, A.-L., Albert, R.: Emergence of scaling in random networks. Science 286, 509–512 (1999)

[4] Chung, F., Lu, L.: Connected components in random graphs with given expected degree sequences. Annals of Combinatorics 6(2), 125–145 (2002)

[5] Chung, F., Lu, L.: The average distances in random graphs with given expected degrees. Proceedings of the National Academy of Sciences (PNAS) 99(25), 15879–15882 (2002)

[6] Indyk, P., Motwani, R.: Approximate nearest neighbors: towards removing the curse of dimensionality. In: 30th Annual ACM Symposium on Theory of Computing (STOC), pp. 604–613 (1998)

[7] Korula, N., Lattanzi, S.: An efficient reconciliation algorithm for social networks. In: 40th International Conference on Very Large Data Bases (VLDB), pp. 377–388 (2014)

[8] Lattanzi, S., Sivakumar, D.: Affiliation networks. In: 41st Annual ACM Symposium on Theory of Computing (STOC), pp. 427–434 (2009)

[9] Narayanan, A., Shmatikov, V.: De-anonymizing social networks. In: 30th IEEE Symposium on Security and Privacy (SP), pp. 173–187 (2009)

[10] Newman, M.E.J.: The structure and function of complex networks. SIAM Review 45(2), 167–256 (2003)

[11] Novak, J., Raghavan, P., Tomkins, A.: Anti-aliasing on the web. In: 13th International Conference on World Wide Web (WWW), pp. 30–39 (2004)

[12] Rao, J.R., Rohatgi, P.: Can pseudonymity really guarantee privacy? In: 9th USENIX Security Symposium (USENIX), pp. 85–96 (2000)

[13] van der Hofstad, R.: Random graphs and complex networks (2009), `www.win.tue.nl/~rhofstad/NotesRGCN.pdf`

[14] Wondracek, G., Holz, T., Kirda, E., Kruegel, C.: A practical attack to de-anonymize social network users. In: IEEE Symposium on Security and Privacy (SP), pp. 223–238 (2010)

[15] Zafarani, R., Liu, H.: Connecting corresponding identities across communities. In: 3rd International Conference on Weblogs and Social Media (ICWSM), pp. 354–357 (2009)

[16] Zheleva, E., Getoor, L.: To join or not to join: The illusion of privacy in social networks with mixed public and private user profiles. In: 18th International Conference on World Wide Web (WWW), pp. 531–540 (2009)

Competitive Algorithms for Restricted Caching and Matroid Caching

Niv Buchbinder[1,⋆], Shahar Chen[2], and Joseph (Seffi) Naor[2,⋆]

[1] Statistics and Operations Research Dept., Tel Aviv University, Israel
niv.buchbinder@gmail.com,
[2] Computer Science Dept., Technion, Haifa, Israel
{shaharch,naor}@cs.technion.ac.il

Abstract. We study the online restricted caching problem, where each memory item can be placed in only a restricted subset of cache locations. We solve this problem through a more general online caching problem in which the cache is subject to matroid constraints. Our main result is an $O(\min\{d, \log r\} \cdot \log c)$-competitive algorithm for the matroid caching problem, where r and c are the rank and circumference of the matroid, and d is the diameter of an auxiliary graph defined over it. In general, this result guarantees an $O(\log^2 k)$-competitiveness for any restricted cache of size k, independently of its structure. In addition, we study the special case of the (n, ℓ)-companion caching problem [8]. For companion caching we prove that our algorithm achieves an optimal competitive factor of $O(\log n + \log \ell)$, improving on previous results of [18].

1 Introduction

Caches are key components in modern computer and networking architectures. Designing efficient caching (or paging) policies is a fundamental online optimization problem with multiple applications. Take, for example, the classical two-level memory system, consisting of a slow memory of infinite size and a small fast memory (the cache). The input is a sequence of page requests which are satisfied one by one. If a page p being requested is already in the cache, then no action is required and no cost is incurred. Otherwise, page p must be brought from the slow memory to the cache (a page fault), incurring some fetching cost, and possibly requiring the eviction of another page in the cache. The objective is to minimize the total fetching cost by wisely choosing which pages to evict.

In recent years significant progress has taken place in the areas of parallel and distributed computing, as well as in local and web storage, giving rise to new, more complex, cache architectures. One example is a multi-core processor, which has become the dominant processor architecture today. In a multi-core processor, every core has a private (and fast) cache, and in addition all the cores share a fully associative cache. Despite extensive research on paging problems, algorithms and performance results for common real-life models such as multi-core processors are still not yet fully understood.

⋆ Supported by ISF grant 954/11 and BSF grant 2010426.

Web storage via content distribution networks provides another example of a non-traditional cache architecture. A content distribution network (CDN) is a large distributed system of servers deployed in multiple data centers over the web. The goal is to provide end-users fast and easy access to content from various devices and locations. There are quite a few public companies that offer such services nowadays, e.g., Akamai, Microsoft's Azure and Amazon's CloudFront. The cache in this context can be a set of servers (e.g., in a data center) that maintains content and provides fast service to a set of end-users. Typically, placing content on this set of servers needs to adhere to various restrictions and constraints, e.g., not any content can be placed on every server. Suppose an online company wishes to hold private, possibly encrypted, content via a CDN. It may be the case that due to digital rights management, or encryption schemes, content belonging to a particular company can only be located on certain dedicated servers which, naturally, have limited capacity. Content which is not placed on these servers will be fetched (upon request) from a distance, incurring a cache miss. Given that massive amounts of information are involved, management of content on the cache is of utmost importance for end-user experience.

We model this kind of constraints on cache architectures via the *restricted caching* model, defined by Brehob *et al* [8]. The idea is that pages, i.e., content, can only be placed in a *restricted* set of cache locations (i.e., servers). Thus, the sets of legal cache locations for two distinct pages may not be identical, though they may have a non-empty intersection. This is in contrast to traditional fully associative caches, where all cache locations are identical, and pages can be located anywhere in the cache. Restricted caches are sometimes referred to as having *arbitrary associativity.* As shown by [22], various algorithms for fully associative caching can result in very poor performance for some settings with arbitrary associativity. One can hope to design *general algorithms* that can cope with this extended family of cache architectures.

A hybrid cache architecture model that interests us in particular is *companion cache*, a simple restricted caching model which includes *victim caches* and *assist caches* as special cases. A companion cache architecture has two components: a fully-associative shared cache of size n and m private caches of size ℓ. The private caches can store items corresponding to different types, e.g., users, locations, file types, etc, while the fully associative cache can store items of any type. Companion caching was first considered by [8] and further studied by [18,16]. A schematic description of a companion cache is presented in Fig. 1.

1.1 Our Contributions and Techniques

The starting point of our work is the key observation that the restricted caching problem is captured by the more general problem of maintaining over time an independent set of a matroid, with respect to an online sequence of element requests. That is, at any point of time the online algorithm has to maintain an independent set of elements which includes the currently requested element. We call this problem the *matroid caching* problem (a formal definition of the problem is given in Section 2). Surprisingly, although the generalization itself is

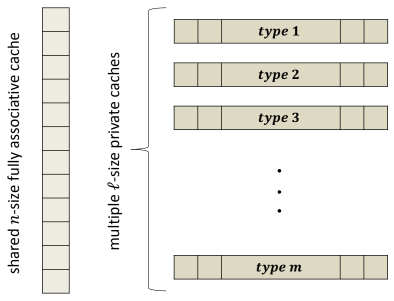

shared n-size fully associative cache
multiple ℓ-size private caches
type 1
type 2
type 3
type m

Fig. 1. (n, ℓ)-companion cache

fairly simple, exploiting matroid properties turns out to be more convenient and powerful, and enables us to both improve and generalize on previous results.

We introduce a general randomized algorithm for the online matroid caching problem on a matroid $\mathcal{M}$, consisting of two components. The first component is a fractional online $O(\log c(\mathcal{M}))$-competitive algorithm, where $c(\mathcal{M})$ is the circumference (largest circuit) of $\mathcal{M}$. The online algorithm and its analysis exploit the matroid properties, and obtain this improved upper bound based on primal-dual linear programming techniques developed in competitive analysis (see the survey of [12]). The second component is a *randomized rounding* scheme which integrates two online rounding algorithms. The first algorithm maintains for every fractional solution a distribution on integral matroid bases (see e.g. [5]). The main difficulty is to wisely update this distribution after every change in the fractional solution. These updates are done by reducing the problem to finding shortest paths in an auxiliary graph defined on the matroid. This auxiliary graph was first introduced by Cunningham [13] for the purpose of determining in strongly polynomial time whether a point is inside a matroid polyhedron. By using the fact that we maintain a feasible fractional solution, and by proving additional properties of the auxiliary graph, we obtain a rounding algorithm which loses a factor of $d_G(\mathcal{M})$, the diameter of the auxiliary graph. The second algorithm is an $O(\log r(\mathcal{M}))$-approximate rounding, recently obtained by [19]. The idea behind this algorithm is to maintain spanning sets of $\mathcal{M}$ in every iteration, and then transform them into bases without incurring any additional loss.

Combining the two components we get our main theorem. To our knowledge, this is the first randomized competitive algorithm for general restricted caching.

Theorem 1. *There is a an $O(\min\{d_G(\mathcal{M}), \log r(\mathcal{M})\} \cdot \log c(\mathcal{M}))$-competitive algorithm for matroid caching, where $r(\mathcal{M})$ and $c(\mathcal{M})$ are the rank and circumference of matroid $\mathcal{M}$, and $d_G(\mathcal{M})$ is the diameter of an auxiliary graph of $\mathcal{M}$.*

We remark that $c(\mathcal{M}) \leq r(\mathcal{M}) + 1$, and thus Theorem 1 guarantees an $O(\log^2 k)$-approximation for any restricted cache of size k, independently of its structure. Nevertheless, in many cases $c(\mathcal{M})$ can be much smaller. For example, in graphic matroids the longest cycle in a graph can be much smaller than the total number of vertices. As for the diameter of the auxiliary graph, we show an example for which $d_G(\mathcal{M}) = \Omega(r(\mathcal{M}))$. This is essentially a worst case example since we prove that $d_G(\mathcal{M}) \leq r(\mathcal{M}) + 1$. However, in some interesting cases of restricted caching the diameter can be much smaller, even a constant. Specifically, for companion caching we show:

Theorem 2. *For the companion caching matroid, $c(\mathcal{M}) = \min\{m, n+1\} \cdot \ell + n + 1$ and $d_G(\mathcal{M}) = 3$. Thus, our algorithm is $O(\log n + \log \ell)$-competitive.*

This result improves on the randomized upper bound of $O(\log n \log \ell)$ of [18], and is optimal as it matches their lower bound of $\Omega(\log n + \log \ell)$.

1.2 Related Work

Caching over a fully associative cache, known as the *paging problem*, was introduced by [7]. Sleator and Tarjan [24] showed that any deterministic algorithm is at least k-competitive, and also proved that LRU is exactly k-competitive. When randomization is allowed, [17] showed a tight $O(\log k)$-competitive algorithm to this well studied problem (later improved by [21,1]). Restricted caching, defined by [8], is a generalization of the paging problem to cache architectures which cannot be modeled as set associative caches. There are very few results on this generalized setting (see [22] for example), however, some specific restricted cache architectures were studied. In particular, [8] introduced the companion caching problem, further studied by [18,22,9,16].

Primal-dual analysis is a fundamental approach in tackling online optimization problems ([3,12,4]), specifically used for the design of caching problems ([5,6,2]). Under a unified framework of online learning and competitive analysis, for a matroid $\mathcal{M} = (\mathcal{E}, \mathcal{I})$, [11] considered the online problem of maintaining matroid bases over time, incurring both movement and service cost, and obtained an $O\left(\log(|\mathcal{E}| - r(\mathcal{M}))\right)$-approximation via a primal-dual approach. The work of [11] differs from ours in two crucial respects. First, their algorithm does not handle constraints such as forcing elements to be included in the matroid base (as in caching);[1] second, they only obtain a fractional solution to their problem. We present a modified primal-dual algorithm which can handle element requests and also simplify and improve on the analysis of [11]. Recently, [10] used a regularization approach to generalize the matroid caching problem. Still, they could only obtain fractional solutions to the problem (similarly to [11]).

Recently, Gupta *et al* [19] also studied the problem of online maintaining matroid bases, giving an $O(\log |\mathcal{E}|)$-competitive fractional algorithm using a primal-dual approach. Similarly to [11], the algorithm cannot handle constraints such as forcing elements to be included in the matroid base. Interestingly, [19] gives an $O(\log r(\mathcal{M}))$-approximation for rounding online a fractional solution. They further generalize the rounding to the weighted version and show that

[1] Note that forcing inclusion via, e.g., a series of steps that incur small costs on other pages until the element is completely in the base, changes the objective function and therefore cannot be performed in that manner.

their result is tight. We note that their hardness result does not hold for the unweighted version, and specifically in some interesting special cases, e.g., uniform matroids, partition matroids, and some restricted cache models.

2 Definitions and Problem Formulation

In the **restricted caching** problem we are given a set $\mathcal{E}$ of n pages and a set $\mathcal{S}$ of available memory slots. Every page p can be located on some subset of $\mathcal{S}$. In every time step t we are given a request for a page p_t. If the page is already in the cache, no cost is incurred. Otherwise, the page must be fetched (from a slower memory) to one of its feasible slots incurring a cost of one unit. If the slot is not empty the algorithm can reorganize the cache for free by either moving pages around between feasible slots or by evicting them. We assume that reorganization is for free, since the time required for fetching a page from slower memory dominates by several orders of magnitude the time for moving pages inside a cache ([8,18]). Thus, the goal is to minimize the total cost, i.e. the number of pages that are fetched.

Companion caching [18] is a special case of restricted caching which is a hybrid of two classic cache architectures – an ℓ-way set-associative cache, and a fully associative cache. There are m caches of size ℓ and a single cache of size n. There are m types of pages, where a page of type i can be stored either in the ith associative cache (of size ℓ), or in the fully associative cache (of size n).

We solve the restricted caching problem through a more general problem on matroids. Matroids are extremely useful combinatorial objects that play an important role in combinatorial optimization since the pioneering work of Edmonds in the 1970s [14,15]. We assume basic familiarity with matroids and only review briefly the important properties that we need for our algorithms and analysis.

A matroid $\mathcal{M} = (\mathcal{E}, \mathcal{I})$ is defined over a finite set $\mathcal{E}$ of element, and a non-empty collection of subsets $\mathcal{I}$ of $\mathcal{E}$, called *independent sets*. For $\mathcal{S} \subseteq \mathcal{E}$, a subset B of $\mathcal{S}$ is called a *base* of $\mathcal{S}$ if B is a maximal independent subset of $\mathcal{S}$. A well known fact is that for any subset $\mathcal{S}$ of $\mathcal{E}$, any two bases of $\mathcal{S}$ have the same size, called the *rank* of $\mathcal{S}$, denoted by $r(\mathcal{S})$. A *circuit* C in a matroid $\mathcal{M}$, is defined as an inclusion-wise minimal dependent set, that is $C \setminus \{e\} \in \mathcal{I}$, for every $e \in C$. The *circumference* of a matroid $\mathcal{M}$, $c(\mathcal{M})$, is the cardinality of the largest circuit in $\mathcal{M}$. For example, the circumference of a graphic matroid in a graph $G = (V, E)$ is the length of the longest simple cycle in it. A subset F of $\mathcal{E}$ is called *nonseparable* if every pair of elements in F lie in a common circuit; otherwise there is a partition of F into non-empty sets F_1 and F_2 with $r(F) = r(F_1) + r(F_2)$ (see [20,26] for more details). Three polytopes associated with a matroid $\mathcal{M}$ are the *matroid polytope* $\mathcal{P}(\mathcal{M})$, the matroid *base polytope* $\mathcal{B}(\mathcal{M})$, and the *spanning set polytope* $\mathcal{P}_{ss}(\mathcal{M})$ (see [14,23]). $\mathcal{P}(\mathcal{M})$ is the convex hull of incidence vectors of the independent sets of $\mathcal{M}$. Similarly, $\mathcal{B}(\mathcal{M})$ is the convex hull of the incidence vectors of the bases of $\mathcal{M}$, and $\mathcal{P}_{ss}(\mathcal{M})$ is the convex hull of the incidence vectors of the spanning sets of $\mathcal{M}$.

$$
\begin{array}{lll|lll}
\text{(P)} & \min \sum_{t=1}^{T} \infty \cdot y_{p_t,t} + \sum_{t=1}^{T} \sum_{p\in\mathcal{E}} z_{p,t} & & \text{(D)} & \max \sum_{t=0}^{T} \sum_{\mathcal{S}\subseteq\mathcal{E}} (|\mathcal{S}| - r(\mathcal{S}))\, a_{\mathcal{S},t} & \\
 & & & \forall p \in \mathcal{E} & b_{p,1} \geq \sum_{\mathcal{S}|p\in\mathcal{S}} a_{\mathcal{S},0} & \\
\forall t \geq 0 \text{ and } \mathcal{S} \subseteq \mathcal{E} & \sum_{p\in\mathcal{S}} y_{p,t} \geq |\mathcal{S}| - r(\mathcal{S}) & & \forall t \geq 1 \text{ and } p \neq p_t & b_{p,t+1} \geq b_{p,t} + \sum_{\mathcal{S}|p\in\mathcal{S}} a_{\mathcal{S},t} & \\
\forall t \geq 1 \text{ and } p \in \mathcal{E} & z_{p,t} \geq y_{p,t-1} - y_{p,t} & & \forall t \geq 1 \text{ and } p \in \mathcal{E} & b_{p,t} \leq 1 & \\
\forall t \text{ and } p \in \mathcal{E} & z_{p,t}, y_{p,t} \geq 0 & & \forall t, p \in \mathcal{E}, \mathcal{S} \subseteq \mathcal{E} & b_{p,t}, a_{\mathcal{S},t} \geq 0 &
\end{array}
$$

Fig. 2. The primal and dual LP formulations for the matroid caching problem

Transversal matroids are one interesting example of matroids. Let $\mathcal{A} = (A_1, A_2, \ldots, A_n)$ be a family of subsets of a finite set $\mathcal{E}$. A set T is called a *transversal* of $\mathcal{A}$ if there exist distinct elements $a_1 \in A_1, a_2 \in A_2, \ldots, a_n \in A_n$ such that $T = \{a_1, a_2, \ldots, a_n\}$. A *partial transversal* is a transversal of some subfamily $(A_{i_1}, A_{i_2}, \ldots, A_{i_k})$ of $\mathcal{A}$. Let $\mathcal{I}$ be the collection of all partial transversals of $\mathcal{A}$. Then $M = (\mathcal{E}, \mathcal{I})$ is a matroid. The bases of this matroid are the inclusion-wise maximal partial transversals, and it follows from König's matching theorem that the rank function r of the transversal matroid induced by $\mathcal{A}$ is given by $r(S) = \min_{T\subseteq S} (|S \setminus T| + |\{i : A_i \cap T \neq \emptyset\}|)$ for $S \subseteq \mathcal{E}$. It is not hard to see that we can model restricted caching using a transversal matroid. For every page $p \in \mathcal{E}$ there is a subset of cache slots to which it can be assigned. Equivalently, for every cache slot s_i, $P(s_i)$ denotes the subset of pages in $\mathcal{E}$ that can be assigned to it. Let $\mathcal{P} = (P(s_1), P(s_2), \ldots, P(S_k))$ be the page subsets for all cache slots. Then, every subset of pages T inducing a valid assignment of pages to the cache is a partial transversal of $\mathcal{P}$. Let $\mathcal{I}$ be the collection of all valid cache assignments. Then, $M = (\mathcal{E}, \mathcal{I})$ is a matroid.

We thus define a general caching problem on any matroid $\mathcal{M} = (\mathcal{E}, \mathcal{I})$. In the *matroid caching* problem the online algorithm must maintain at any time t an independent set $\mathcal{S}_t \in \mathcal{I}$ (the cache) of the matroid. At any time t, upon receiving a request for an element of the matroid (page) p_t: if $p_t \in \mathcal{S}_{t-1}$ no cost is incurred; otherwise, p_t must be added to $\mathcal{S}_{t-1}$ paying one unit. If $\mathcal{S}_{t-1} \cup \{p_t\}$ is dependent, elements must be removed (evicted), so that S_t becomes independent.

We can formulate the matroid caching problem as follows. Let the variables $y_{p,t}$ denote the fraction of page $p \in \mathcal{E}$ **missing** from the cache at time t. This means that at any time t, $(1 - y_{p_t,t}) = 1$ and $(1 - y_t) \in \mathcal{P}(\mathcal{M})$. The fetching cost of page p is $\max\{0, y_{p,t-1} - y_{p,t}\}$. Figure 2 contains the linear relaxation of this formulation (P), as well as the corresponding dual program (D). Both programs play a central role in our analysis. We define D as the value of the dual program, which is a lower bound on the value of any primal solution.

3 Main Algorithm

In this section we solve the fractional version of the matroid caching problem, proving the following theorem.

Algorithm 1. Matroid Caching Algorithm

```
Initiate $\eta = \log 2$.
During execution, maintain the following relation between primal and dual variables:
    $y_{p,t} = f(b_{p,t+1}) = \frac{\mathrm{e}^{\eta \cdot b_{p,t+1}} - 1}{\mathrm{e}^{\eta} - 1}$.
Start with an empty cache $y_{p,0} = 1$, and set $b_{p,1} = 1$ accordingly.
for $t = 1, 2, \ldots$ do
    Let $p_t$ be the current requested page.
    (Update step): Set $y_{p_t,t} = b_{p_t,t+1} = 0$.
    (Normalization step): As long as $\sum_{p \in \mathcal{E}} y_{p,t} < |\mathcal{E}| - r(\mathcal{E})$:
    1.  Let $\mathcal{S}$ be the set of evictable pages.
    2.  Update
            $\eta \leftarrow \max\left\{\eta, \log\left(\frac{|\mathcal{S}|}{|\mathcal{S}| - r(\mathcal{S})} + 1\right)\right\}$,
        update $b_{p,t+1}$ to maintain $y_{p,t}$ unchanged.
    3.  For each $p \in \mathcal{S}$, aside from $p_t$, update $b_{p,t+1} \leftarrow b_{p,t+1} + a_{\mathcal{S},t}$ (and $y_{p,t}$ accordingly), where $a_{\mathcal{S},t}$ is the smallest value such that there exists $p \in \mathcal{S}$ that becomes unevictable.
end for
```

Theorem 3. *There is an online algorithm with competitive ratio* $2 \log(1 + c(\mathcal{M}))$ *for the fractional matroid caching problem on a matroid* $\mathcal{M}$*, where* $c(\mathcal{M})$ *is the circumference of* $\mathcal{M}$*.*

At a high level the description of the algorithm is as follows. Without loss of generality we assume that the cache is initially full (this can be achieved by requesting a sequence of $r(\mathcal{M})$ independent dummy pages, before the actual input sequence). The algorithm maintains a solution y_t such that $(1 - y_t)$ is in the *base polytope* of $\mathcal{M}$. Whenever a page p_t is fetched, the algorithm updates $y_{p_t,t} = 0$. This generates a solution y_t whose complement is in the *spanning polytope* of $\mathcal{M}$, i.e., $(1 - y_t) \in \mathcal{P}_{ss}(\mathcal{M})$. Next, the algorithm performs a sequence of steps which gradually evict other pages from the cache making it feasible again. We refer to each such step as a *normalization step.* In each normalization step we consider the set $\mathcal{S}$ of all *evictable* pages. A page is evictable if we can increase $y_{p,t}$ by an infinitely small value ε, and remain in $\mathcal{P}_{ss}(\mathcal{M})$. It is well known that $\mathbf{x} \in \mathcal{P}(\mathcal{M}^*)$ iff $(1 - \mathbf{x}) \in \mathcal{P}_{ss}(\mathcal{M})$, where $\mathcal{M}^*$ is the dual matroid for $\mathcal{M}$. Thus, we can use the latter condition to compute $\mathcal{S}$ efficiently. Let us consider the maximal *dual tight set* with respect to our current solution. A set $\mathcal{T} \subseteq \mathcal{E}$ is tight if $\sum_{p \in \mathcal{T}} y_{p,t} = r^*(\mathcal{T})$, and using submodularity of r^*, if $\mathcal{T}_1$ and $\mathcal{T}_2$ are tight, then so are $\mathcal{T}_1 \cap \mathcal{T}_2$ and $\mathcal{T}_1 \cup \mathcal{T}_2$. In particular, there is a maximal tight set $\mathcal{T}_{\max}$ containing all pages whose value $y_{p,t}$ cannot be increased without violating the dual matroid constraints. Therefore, in each normalization step we define the evictable set $\mathcal{S}$ as all pages which are *not* in a dual tight set, $\mathcal{S} = \mathcal{E} \setminus \mathcal{T}_{\max}$, and increase their value until an additional page joins a tight set. In general, the above condition can be checked in polynomial time by a reduction to submodular function minimization [23, Ch. 40]. For transversal matroids, $\mathcal{S}$ can be computed using flow techniques. The sequence of normalization steps ends when all pages become tight. Algorithm 1 formally describes the procedure.

We remark that the algorithm can be easily generalized to the weighted version, where fetching page p incurs a cost of w_p, by maintaining the following primal-dual relationship between the variables: $y_{p,t} = f(b_{p,t+1}) = \frac{\mathbf{e}^{\eta(b_{p,t+1}/w_p)}-1}{\mathbf{e}^{\eta}-1}$.

We prove Theorem 3 using a technical lemma on matroids whose proof is deferred to the full version.

Lemma 1.

$$c(\mathcal{M}) = \max_{\text{nonseparable } A \subseteq \mathcal{E}} \left\{ \frac{|A|}{|A| - r(A)} \right\}.$$

Proof (of Theorem 3).
The analysis of the algorithm's performance is done using the primal-dual method.

Primal (P) is feasible: Clearly y_0 is feasible. By induction on the steps, we prove that the algorithm produces a feasible solution (i.e., $(1 - y_t) \in \mathcal{B}(\mathcal{M})$). The update step sets $y_{p_t,t} = 0$. Then, in the normalization step the value of each $y_{p,t}$ only grows. Note that as long as the primal solution is not feasible, $y_t \in \mathcal{P}(M^*)$, but $y_t \notin \mathcal{B}(M^*)$, hence not all pages are in a dual tight set and $\mathcal{S}$ is non-empty. After at most n iterations all pages become tight, and y_t is feasible.

Dual (D) is feasible: Since initially the cache is empty, and for each p, $y_{p,0} = b_{p,1} = 1$, then by setting $a_{\mathcal{S},0} = 0$ for all $\mathcal{S} \subseteq \mathcal{E}$, we have that the first set of dual constraints is feasible. The primal solution is feasible, thus $0 \leq y_{p,t} \leq 1$, so since we preserve the primal-dual relation we get: $0 \leq \frac{\mathbf{e}^{\eta \cdot b_{p,t+1}}-1}{\mathbf{e}^{\eta}-1} \leq 1$. Simplifying we get $0 \leq b_{p,t+1} \leq 1$. Finally, by construction we keep the dual constraints with equality: $b_{p,t+1} = b_{p,t} + \sum_{\mathcal{S}|p\in\mathcal{S}} a_{\mathcal{S},t}$. The only exception is due to line 2 in the normalization step. Since f is monotonically decreasing in η and in $b_{p,t+1}$, every increase in η in line 2 also induces an increase in $b_{p,t+1}$, implying $b_{p,t+1} \geq b_{p,t} + \sum_{\mathcal{S}|p\in\mathcal{S}} a_{\mathcal{S},t}$.

Primal-dual relation: We bound the primal cost in each iteration by the change in the dual cost. Let $\frac{dD}{a_{\mathcal{S},t}}$ be the increase rate of the dual solution at time t, when $a_{\mathcal{S},t}$ gradually increases. Let η_t denote the current value of η, and let η_{final} denote its value at the end of the execution. The dual increase rate is $|S| - r(S)$. For the primal cost, let us consider eviction costs instead of fetching costs (clearly, this adds at most $r(\mathcal{E})$ to the overall cost). That is, we bound the change in the primal variables during the normalization step instead of the update step.

$$\sum_{p\in\mathcal{S}\setminus\{p_t\}} \frac{dy_{p,t}}{da_{S,t}} = \eta_t \sum_{p\in S\setminus\{p_t\}} \left(y_{p,t} + \frac{1}{\mathbf{e}^{\eta_t}-1}\right) \tag{1}$$

$$< \eta_t \cdot \left(r^*(\mathcal{E}) - r^*(\mathcal{E}\setminus\mathcal{S}) + \frac{|\mathcal{S}|-1}{\mathbf{e}^{\eta_t}-1}\right) \tag{2}$$

$$\leq 2\eta_t \cdot (|\mathcal{S}| - r(\mathcal{S})) = 2\eta_t \cdot \frac{dD}{da_{S,t}} \leq 2\eta_{\text{final}} \cdot \frac{dD}{da_{S,t}} \tag{3}$$

$$\leq 2\log(1+c(\mathcal{M}))\frac{dD}{da_{S,t}}, \tag{4}$$

where (1) follows from the fact that $\frac{dy_{p,t}}{da_{S,t}} = \frac{dy_{p,t}}{db_{p,t+1}} = \eta_t \cdot (y_{p,t} + \frac{1}{\mathbf{e}^{\eta_t}-1})$, (2) follows as $\sum_{p\in\mathcal{E}\setminus\mathcal{S}} y_{p,t} = r^*(\mathcal{E}\setminus\mathcal{S})$, (3) follows by the dual rank function definition and as $\eta_t \geq \log\left(1+\frac{|\mathcal{S}|}{|\mathcal{S}|-r(\mathcal{S})}\right)$ in the algorithm, and finally (4) follows by Lemma 1, since $\eta_{\text{final}} \leq \max_{\mathcal{S}}\{\log\left(\frac{|\mathcal{S}|}{|\mathcal{S}|-r(\mathcal{S})}+1\right)\}$. It is not hard to see that $\mathcal{S}$ is nonseparable. Assume by contradiction that $\mathcal{S}$ can be partitioned into $\mathcal{S}_1, \mathcal{S}_2$ such that $r(\mathcal{S}_1) + r(\mathcal{S}_2) = r(\mathcal{S})$, and assume without loss of generality that $p_t \in \mathcal{S}_1$. Then, clearly, since every page $p \in \mathcal{S}_2$ is unevictable before p_t is fetched, it remains unevictable when $y_{p_t,t}$ is set to 0.

4 Rounding the Fractional Solution Online

In this section we describe our rounding procedure for matroid caching. Our goal is to map the fractional solution produced by the algorithm of Section 3 into a distribution on the bases of the matroid, and show how to maintain the distribution while paying a small cost. Let $y_{t-1} \in \mathcal{B}(\mathcal{M})$ be the fractional solution at time $t-1$. Moving from $y_{t-1} \in \mathcal{B}(\mathcal{M})$ to $y_t \in \mathcal{B}(\mathcal{M})$ can be divided without loss of generality into a sequence of changes, where in each one $y_{u,t-1}$ is increased by ε, and $y_{v,t-1}$ is decreased by ε, for some choice of u and v. Thus, the change in the cost of the fractional solution is ε. Our algorithm holds at any time a decomposition $\mathcal{D}$ of the current fractional solution y, such that $y = \sum_{B\in\mathcal{D}} \lambda_B \cdot B$, and $\sum_{B\in\mathcal{D}} \lambda_B = 1$. We want to update the current distribution $\mathcal{D}$ so it is consistent with the new fractional solution y_t, while making as few changes as possible. This immediately gives us an online mapping of our fractional algorithm to a randomized integral one (see, e.g., [5] for the explicit mapping).

To update the distribution $\mathcal{D}$ we use an auxiliary graph which was initially introduced by [13] to determine whether a given point is inside $\mathcal{P}(\mathcal{M})$, and if so find a decomposition of it into independent sets. Given a decomposition $\mathcal{D}$ of a fractional solution $y \in \mathcal{B}(\mathcal{M})$, let $G(\mathcal{D}) = (V, E)$ be a directed graph, with $V = \mathcal{E}$. The edges of the graph are defined as follows:

$$E = \{(u,v) : u, v \in V \text{ and } \exists B \in \mathcal{D}, \lambda(B) > 0 \text{ such that } B + u - v \text{ is a base}\}.$$

Proposition 1. *Let $y \in \mathcal{B}(\mathcal{M})$ be a fractional solution and let $\mathcal{D}$ be a decomposition of y. Let $y' \in \mathcal{B}(\mathcal{M})$ be a fractional solution such that $y'_u = y_u + \varepsilon, y'_v = y_v - \varepsilon$ and $y'_w = y_w$ for any $w \neq u, v$. Then, there exists a directed path in $G(\mathcal{D})$ from u to v. Furthermore, if $P_G(u,v)$ is the shortest path from u to v, $\mathcal{D}$ can be converted to a decomposition of y', $\mathcal{D}'$, while paying $\varepsilon \cdot |P_G(u,v)|$.*

Proof. Let us look at the problem considered by [13]. In this work, we are given $x_0 \in \mathcal{P}(\mathcal{M})$, a decomposition of x_0 into independent sets, and $x_1 \in \mathbb{R}^{|S|}$, where $x_0 \leq x_1$. The goal is to iteratively bring x_0 closer to x_1, i.e. find $x' \in \mathcal{P}(\mathcal{M})$ such that $x_0 \leq x' \leq x_1$, until we either reach x_1, or no such solution exists. To do so, an auxiliary graph similar to ours is constructed and an *augmenting path* in it is computed. In particular, it is shown that if $x_1 \in \mathcal{P}(\mathcal{M})$, then such a path always exists [13, Theorem 2.2]; we follow this path, and for every edge (e, f) on it we add e and remove f, in some base $B \in \mathcal{D}$ in which $B + e - f$ is also a base[2], and obtain a feasible decomposition of x' [13, Lemma 4.3]. In fact, our setting is a special case of the latter setting. If we set $x_0 = y - \varepsilon \cdot 1_v$, $x_1 = x_0 + \varepsilon \cdot 1_u = y'$, and a decomposition of x_0 is defined as $\mathcal{D}$ after removing v from ε-measure of its bases, then our problem satisfies the conditions of [13]. As a result, there exists a path from u to v, and to obtain $\mathcal{D}'$ all we need to do is follow the path $P_G(u,v)$ and perform $|P_G(u,v)|$ swaps. □

Proposition 1 suggests a way of maintaining a decomposition of the fractional solution online. The payment of the rounding scheme depends on the length of the path in $G(\mathcal{D})$. The property that determines the worst case rounding quality is therefore the diameter of $G(\mathcal{D})$, that is, the maximum shortest-path among all pairs $u, v \in V$ for which there exists a path from u to v. For a matroid $\mathcal{M}$, let $d_G(\mathcal{M})$ be the maximum diameter over any two fractional solutions $y, y' \in \mathcal{B}(\mathcal{M})$ such that y' differs from y in two coordinates, and any valid decomposition of y into bases. We obtain the following bound on $d_G(\mathcal{M})$.

Lemma 2. *For any matroid $\mathcal{M}$, $d_G(\mathcal{M}) \leq r(\mathcal{M}) + 1$.*

The above lemma immediately provides an upper bound on the performance of our rounding algorithm. As a matter of fact, this bound is tight up to an additive constant. We defer the proof of the lemma, and the lower bound analysis, to the full version of the paper. In general, this bound may be quite large. However, in Section 5 we explore several interesting special cases of the restricted caching problems and show that the diameter in these cases is much smaller. Moreover, for the general matroid case, we are able to guarantee a logarithmic competitive ratio using an online rounding algorithm recently proposed by [19]. The idea behind the algorithm is to maintain spanning sets in every iteration, and then transform them into bases without incurring any additional loss. Rounding spanning sets is based on recent results on contention resolution schemes [25] (See [19], Section 4 for more details). Therefore, for rounding, one can always apply the auxiliary graph approach, and if $d_G(\mathcal{M})$ is greater than $\log r(\mathcal{M})$ switch to

[2] Only an ε-measure of B is updated. That is $\lambda_B \leftarrow \lambda_B - \varepsilon$, and the probability of $B + e - f$ increases by ε.

the rounding approach of [19]. Combining Theorem 3 with Proposition 1 as well as with the above insight, yields our main Theorem 1.

5 Special Cases of Restricted Caching

As already mentioned, although $d_G(\mathcal{M})$ has a tight upper bound of $r(\mathcal{M})+1$ in general, there are several special cases in which the diameter becomes significantly smaller. We demonstrate this on two previously studied cache architectures, obtaining tight performance guarantees.

Classical Paging: In the classical paging problem there is a cache of size k and n pages. Each page may be located anywhere in the cache. It is not hard to see that in this case $c(\mathcal{M}) = k+1$, and $d_G(\mathcal{M}) = 2$ (as for every u, v ε-update, the out-degree of u is at least $n-k$, and the in-degree of v is at least k). Thus, our algorithm is $O(\log k)$-competitive which is optimal up to constants.

Companion Caching: In the companion caching problem there are m types of pages, where page of type i can be stored either in the ith associative cache (of size ℓ), or in the fully associative cache (of size n). As companion cache is a restricted cache, we can represent it via a transversal matroid, and prove the bound in Theorem 2 (the proof is deferred to the full version). This result matches the lower bound shown by [18]. As argued by [18], any algorithm with free reorganization can be implemented online in the no-reorganization model while losing at most a factor of 3. Thus, we also get tight competitiveness for the companion caching problem without free reorganization.

6 Conclusions

We studied the restricted caching problem in which each page in memory can only be placed in a restricted subset of cache locations. We solved this problem through a more general problem of maintaining over time an independent set of a matroid $\mathcal{M}$, obtaining an $O(\min\{d_G(\mathcal{M}), \log r(\mathcal{M})\} \cdot \log c(\mathcal{M}))$-competitive algorithm, where $r(\mathcal{M})$ and $c(\mathcal{M})$ are the rank and circumference of $\mathcal{M}$, and $d_G(\mathcal{M})$ is the diameter of an auxiliary graph of $\mathcal{M}$. This guarantees an $O(\log^2 k)$-competitiveness for any restricted cache of size k, independently of its structure.

Our work suggests several future research directions and open questions. First, we showed that $d_G(\mathcal{M})$ can be in some cases as large as $r(\mathcal{M})$. However, we could not come up with an example where the rounding algorithm can be forced to use long paths repeatedly for many steps. Thus, a reasonable conjecture is that the amortized cost of our rounding algorithm (via the auxiliary graph) might only be a constant. Proving this conjecture will give an optimal $O(\log k)$-competitive algorithm for any restricted caching problem. Another open problem is finding (and characterizing) the circumference of a transversal matroid. This parameter is interesting since it serves as the performance bound of our fractional algorithm. The problem of finding the circumference in a general matroid has very poor approximation factors, as in graphic matroids it reduces to finding the longest simple cycle. However, we do not know anything about the hardness of the problem for transversal matroids.

Acknowledgements. We thank Roy Friedman for helpful discussions on restricted caching in content distribution networks.

References

1. Achlioptas, D., Chrobak, M., Noga, J.: Competitive analysis of randomized paging algorithms. Theoretical Computer Science 234, 203–218 (2000)
2. Adamaszek, A., Czumaj, A., Englert, M., Räcke, H.: An o(log k)-competitive algorithm for generalized caching. In: Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 1681–1689 (2012)
3. Alon, N., Awerbuch, B., Azar, Y., Buchbinder, N., Naor, J.: The online set cover problem. SIAM Journal on Computing 39(2), 361–370 (2009)
4. Bansal, N., Buchbinder, N., Naor, J.: Towards the randomized k-server conjecture: A primal-dual approach. In: Proceedings of the Twenty-first Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 40–55 (2010)
5. Bansal, N., Buchbinder, N., Naor, J.: A primal-dual randomized algorithm for weighted paging. J. ACM 59(4) (2012)
6. Bansal, N., Buchbinder, N., Naor, J.: Randomized competitive algorithms for generalized caching. SIAM J. Comput. 41(2), 391–414 (2012)
7. Belady, L.: A study of replacement algorithms for a virtual-storage computer. IBM Systems Journal 5(2), 78–101 (1966)
8. Brehob, M., Enbody, R., Torng, E., Wagner, S.: On-line restricted caching. In: Proc. 12th Annual ACM-SIAM Symp. on Discrete Algorithms, pp. 374–383 (2001)
9. Brehob, M., Enbody, R., Wagner, S., Torng, E.: Optimal replacement is np-hard for nonstandard caches. IEEE Transactions on Computers 53(1), 73–76 (2004)
10. Buchbinder, N., Chen, S., Naor, J.: Competitive analysis via regularization. In: SODA, pp. 436–444 (2014)
11. Buchbinder, N., Chen, S., Naor, J., Shamir, O.: Unified algorithms for online learning and competitive analysis. In: COLT, pp. 5.1–5.18 (2012)
12. Buchbinder, N., Naor, J.: The design of competitive online algorithms via a primal-dual approach. Foundations and Trends in Theoretical Computer Science 3(2-3), 93–263 (2009)
13. Cunningham, W.H.: Testing membership in matroid polyhedra. Journal of Combinatorial Theory, Series B 36(2), 161–188 (1984)
14. Edmonds, J.: Submodular functions, matroids, and certain polyhedra. In: Combinatorial Structures and Their Applications, pp. 69–87 (1970)
15. Edmonds, J.: Matroids and the greedy algorithm. Mathematical Programming 1(1), 127–136 (1971)
16. Epstein, L., van Stee, R.: Calculating lower bounds for caching problems. Computing 80(3), 275–285 (2007)
17. Fiat, A., Karp, R.M., Luby, M., McGeoch, L.A., Sleator, D.D., Young, N.E.: Competitive paging algorithms. Journal of Algorithms 12(4), 685–699 (1991)
18. Fiat, A., Mendel, M., Seiden, S.S.: Online companion caching. In: Möhring, R.H., Raman, R. (eds.) ESA 2002. LNCS, vol. 2461, pp. 499–511. Springer, Heidelberg (2002)

19. Gupta, A., Talwar, K., Wieder, U.: Changing bases: Multistage optimization for matroids and matchings. In: Esparza, J., Fraigniaud, P., Husfeldt, T., Koutsoupias, E. (eds.) ICALP 2014. LNCS, vol. 8572, pp. 563–575. Springer, Heidelberg (2014)
20. Harary, F., Welsh, D.: Matroids versus graphs. In: The Many Facets of Graph Theory. Lecture Notes in Mathematics, vol. 110, pp. 155–170 (1969)
21. McGeoch, L.A., Sleator, D.D.: A strongly competitive randomized paging algorithm. Algorithmica 6(1-6), 816–825 (1991)
22. Peserico, E.: Online paging with arbitrary associativity. In: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 555–564 (2003)
23. Schrijver, A.: Combinatorial Optimization: polyhedra and efficiency. Springer (2003)
24. Sleator, D.D., Tarjan, R.E.: Amortized efficiency of list update and paging rules. Communication of the ACM 28(2), 202–208 (1985)
25. Vondrák, J., Chekuri, C., Zenklusen, R.: Submodular function maximization via the multilinear relaxation and contention resolution schemes. In: Proc. of the 43rd Annual ACM Symposium on Theory of Computing, pp. 783–792 (2011)
26. Whitney, H.: On the abstract properties of linear dependence. American Journal of Mathematics 57(3), 509–533 (1935)

Improved Algorithms for Resource Allocation under Varying Capacity[⋆]

Venkatesan T. Chakaravarthy[1], Anamitra R. Choudhury[1], Shalmoli Gupta[2,⋆⋆], Sambuddha Roy[3,⋆⋆], and Yogish Sabharwal[1]

[1] IBM Research - India
{vechakra,anamchou,ysabharwal}@in.ibm.com
[2] University of Illinois at Urbana-Champaign, USA
shalmoli@gmail.com
[3] Amazon, Bangalore, India
shombuddha@gmail.com

Abstract. We consider the problem of scheduling a set of jobs on a system that offers certain resource, wherein the amount of resource offered varies over time. For each job, the input specifies a set of possible scheduling instances, where each instance is given by starting time, ending time, profit and resource requirement. A feasible solution selects a subset of job instances such that at any timeslot, the total requirement by the chosen instances does not exceed the resource available at that timeslot, and at most one instance is chosen for each job. The above problem falls under the well-studied framework of unsplittable flow problem (UFP) on line. The generalized notion of scheduling possibilities captures the standard setting concerned with release times and deadlines. We present improved algorithms based on the primal-dual paradigm, where the improvements are in terms of approximation ratio, running time and simplicity.

1 Introduction

We study the classical scheduling setting of *unsplittable flow problem on line* (UFP). Consider a system offering a certain resource as a service for executing jobs. The total amount of the resource offered by the system may be different at different points of time. Each job is specified as an interval consisting of a starting time and an ending time, and requires a particular amount of the resource for its execution. A feasible solution selects a subset of jobs for execution such that at any point of time, the total amount of resource requirement does not exceed the total amount of the resource available at that time point. Each job is associated with a profit and the objective is to maximize the aggregate profit of the scheduled jobs.

The problem is applicable in a variety of settings based on the resource under consideration, examples of which include computational nodes, storage, electricity and network bandwidth. We refer to prior work for real-life applications of

[⋆] Full version of the paper is available as Arxiv preprint.
[⋆⋆] Work was done while the author was at IBM Research - India

the above job selection problem in parallel/distributed computing and network management [22]. The terminology "unsplittable flow" arises from a more generic graph theoretic framework and the above scheduling problem corresponds to the case, wherein the graph is simply a path. We refer to the survey by Kolliopoulos [24] for a discussion on the general graph theoretic framework.

In the setting considered so far, each job is specified by a single interval where it must be scheduled. Consider a more general scenario where each job specifies a set of possible time intervals and the job may be scheduled in any one of those intervals. In other words, each job can be a viewed as a set (or *bag*) of *job instances* of which at most one can be selected for execution. We allow different instances of the same job to have different resource requirements, processing times (interval length) and profits. The above generalization captures a variety of scenarios. For example, consider the standard setting where time is divided into discrete timeslots and each job is specified by a processing time and a window consisting of release time and deadline. A job with release time r and deadline d, and processing time p can be modelled as a bag containing $(d - r - p + 1)$ instances corresponding to the integer intervals of length p lying between r and d. Motivated by such applications, the `UFP` *problem with bag constraints* (`BagUFP`) has been well-studied. The problem is formally defined next.

`BagUFP` **- Problem Definition:** We assume that time is divided into discrete timeslots $1, 2, \ldots, T$ and let $\mathcal{T} = \{1, 2, \ldots, T\}$ denote the set of all timeslots. For each timeslot t, the input specifies an integer $c(t)$, which is the *capacity* available at the timeslot t. The input consists of a set of jobs $\mathcal{J}$. Each job $a \in \mathcal{J}$ consists of a set of *job instances* of which at most one can be selected for execution. Each job instance u is associated with a starting timeslot $s(u)$, an ending timeslot $e(u)$, a *demand* $h(u)$ and a profit $p(u)$. The interval $[s(u), e(u)]$ is called the *span* of u.

Let $\mathcal{U}$ denote the set of all job instances (over all jobs) and let n be the total number of job instances. Given a set of job instances $X \subseteq \mathcal{U}$, let $p(X)$ denote the cumulative profit $\sum_{u \in X} p(u)$. A job instance $u \in \mathcal{U}$ is said to be *active* at a timeslot t, if t belongs to the range $[s(u), e(u)]$; this is denoted using the notation $u \sim t$. A feasible solution is a set of job instances $S \subseteq \mathcal{U}$ such that the following two constraints are satisfied. The first constraint (called the *capacity constraint*) enforces that for any timeslot t, the cumulative demand of job instances in S active at the timeslot t is at most the capacity $c(t)$ available at t: $\sum_{u \in S\,:\,u \sim t} h(u) \leq c(t)$. The second constraint (called the *bag constraint*) requires that at most one instance is picked from each job. The problem is to find a feasible solution S having the maximum profit $p(S)$. □

Remark. Using preprocessing, we can modify the input capacities suitably so that only the timeslots wherein some job instance starts or finishes is of relevance and the other timeslots can be ignored (see [6]). In the rest of the paper, without loss of generality, we assume that the number of timeslots is at most $2n$.

Special Cases of `BagUFP`: Prior work has addressed two important restrictions of the `BagUFP` problem.

- *Single job instance:* In this setting, each job has exactly one job instance; namely, the setting does not involve the bag constraint.
- *No-bottleneck Assumption (NBA):* In this setting, we assume that the maximum demand of any job instance is at most the minimum capacity available, i.e., $h_{\max} \leq c_{\min}$, where $h_{\max} = \max_{u \in \mathcal{U}} h(u)$ and $c_{\min} = \min_{t \in \mathcal{T}} c(t)$.

The NBA setting is well-studied and arises in scenarios wherein the system capacity is larger than the demand of any individual job instance.

By considering the combinations of the two restrictions, we get four different special cases of the problem: (1) `BagUFP` - the most general case, where neither restriction applies; (2) `UFP` - assumes that every job has only one job instance, but does not impose NBA; (3) `BagNbaUFP` - requires NBA, but allows each job to have arbitrary number of job instances; (4) `NbaUFP` - the most specialized case that imposes both the restrictions.

Prior Work: Consider the simplest special case, where each job has only one instance and furthermore, all the job instances have unit demand and all the timeslots offer unit capacity. This is the same as the classical maximum weight independent set problem on interval graphs, which can easily be solved optimally via dynamic programming.

Spieksma [26] considered the above problem along with bag constraints, under the name *weighted job interval selection problem* (`WJISP`). He showed that the problem is NP-hard and APX-hard. Bar-Noy et al. [8] and independently, Berman and Dasgupta [9] presented 2-approximation algorithms. Both these algorithms are based on the local ratio technique. For the unweighted version (wherein all the job instances have unit profit), Chuzhoy et al. [20] presented an algorithm with an approximation ratio of 1.582.

Calinescu et al. [11] studied the case where each job has only one instance and the capacity offered is uniform across all timeslots (however, the instances can have arbitrary demands). They presented a 3-approximation algorithm via the LP-rounding technique. For the above setting with bag constraints, Bar-Noy et al. [7] designed a 5-approximation algorithm using the local ratio technique.

Let us now look at non-uniform capacity setting, viz `NbaUFP`, `UFP`, `BagNbaUFP` and `BagUFP`. For the simplest case of `NbaUFP`, Chakrabarti et al. [16] provided the first constant factor approximation algorithm. Subsequently, Chekuri et al. [19] improved the ratio to $2 + \epsilon$ (here and in the rest of the paper, ϵ would refer to a constant $\epsilon > 0$). Relaxing the NBA assumption, Chakrabarti et al. [16] also presented an algorithm for `UFP`, with an approximation ratio of $O(\log \frac{h_{\max}}{h_{\min}})$, where $h_{\max}$ and $h_{\min}$ are the maximum and minimum demands. For the same problem, Bansal et al. [6] presented an $O(\log n)$-approximation algorithm. In a different paper, Bansal et al. [5] obtained a QPTAS. Recently Bonsma et al. [10] gave the first constant factor polynomial time algorithm; their algorithm achieves an approximation factor of $7 + \epsilon$. Subsequently, the ratio was improved to $2 + \epsilon$

by Anagnostopoulos et al. [4]. The above algorithms are based on sophisticated LP-rounding and dynamic programming strategies.

Chakaravarthy et al. [14] studied the notion of bag constraints and devised an algorithm for BagUFP with an approximation ratio of $O(\log \frac{c_{\max}}{c_{\min}})$, where $c_{\max}$ and $c_{\min}$ are the maximum and minimum capacities, respectively. It remains an interesting open question to design a constant factor approximation algorithm for the BagUFP problem. However, this has been achieved under the NBA assumption. Chakaravarthy et al. [13] presented a 120-approximation algorithm for the BagNbaUFP problem, via a reduction from the non-uniform capacity setting to the uniform capacity setting. Subsequently, Elbassioni et al. [21] improved the factor to 65 using LP-rounding techniques.

The BagUFP problem has also been studied under distributed models and constant factor approximation algorithms are known in the uniform capacity setting, and logarithmic factor approximations in the non-uniform capacity setting [25,15,12]. These algorithms are based on the primal-dual paradigm and they also apply to the parallel setting.

Our Results: In this paper, we present improved algorithms for the BagUFP and its special cases. The main tool used in our work is the primal-dual paradigm (or equivalently the local ratio method), leading to simpler algorithms which provide improvements in terms of approximation ratio and running time. In contrast, prior work on the non-uniform setting predominantly use sophisticated LP rounding and dynamic programming approaches. Furthermore, prior work [12,15,25] has shown that the primal-dual method is more suitable for the parallel/distributed models, and so, the procedures developed in this paper may be adaptable for these environments. We next state the main results of the paper.

– A 17-approximation algorithm for BagNbaUFP problem.
– An $O(\log n)$-approximation algorithm for the BagUFP problem.

Both the algorithms are based on the primal-dual method and run in time $O(n^2)$. The previously best known approximation ratios for the above two problems are 65 [21] and $O(\log \frac{c_{max}}{c_{min}})$ [14], respectively. The second ratio can be as high as $O(n)$ in the worst case. Furthermore, our algorithms are also more efficient in terms of running time; both the previous algorithms go via LP-rounding and need to solve linear programs.

The above two main results deal with the versions having the bag constraint. The procedures developed as part of these results also provide an interesting improvement for the versions devoid of the constraint. Recall that for the UFP problem, Bonsma et al. [10] presented a $(7+\epsilon)$-approximation algorithm, which was subsequently improved to $(2+\epsilon)$ by Anagnostopoulos [4]. Both these algorithms run in polynomial time, but the exponent of the polynomial is very high. Bonsma et al. addressed the issue by presenting another algorithm having a faster running time of $O(n^4)$, but with an increased approximation ratio of $(25+\epsilon)$. The above algorithm has two components based on LP-rounding and dynamic programming, respectively. Of these, the first component can be replaced by one of our procedures yielding a simpler algorithm with the same running time, but with a better approximation ratio of 13. We present a 13-approximation algorithm for the UFP problem with a running time of $O(n^4)$.

Finally, consider NbaUFP, the most restricted special case. Chakrabarti et al. [16] designed the first constant factor approximation algorithm for this problem via rounding a natural LP. In that context, they raised the question of devising such an algorithm based on the primal-dual method. Our algorithm for BagNbaUFP answers this question affirmatively.

2 Overview

In this section, we provide an overview of our algorithms, highlighting the main components and place them in the context of prior work. Most prior work on BagUFP and its variants go via classifying the job instances into two categories based on their demands. Consider any job instance $u \in \mathcal{U}$. Among all timeslots in the span of u, let t be any timeslot having the minimum capacity (breaking ties arbitrarily). The timeslot t is called the *bottleneck timeslot* for u (denoted by $\mathtt{bt}(u)$) and its capacity *bottleneck capacity* for u (denoted by $\mathtt{bc}(u)$). Fix any constant $0 < \gamma \leqslant 1$. We say that the job instance u is *γ-small*, if $h(u) \leqslant \gamma \mathtt{bc}(u)$; otherwise, u is said to be *γ-large*. For the case where $\gamma = 1/2$, we shall drop the prefix and simply write "small" and "large" to mean 1/2-small and 1/2-large job instances, respectively.

Let Opt denote the optimal solution. Let $\mathcal{U}_s$ and $\mathcal{U}_l$ denote the set of all small and large job instances, respectively. Let Opt_s and Opt_l denote the optimal solution considering only the small and large job instances, respectively. We shall design two procedures that would produce solutions S_s and S_l such that S_s is an f_1-approximation to Opt_s and S_l is an f_2-approximation to Opt_l, for some $f_1, f_2 \geqslant 1$. The best of the two solutions is taken to be the final solution S. It is easy to see that S is an $(f_1 + f_2)$-approximation to Opt.

Given the above aggregation result, we consider the small and the large job instances separately. The core technical component of the paper is a simple and fast primal-dual procedure for handling the small job instances, while guaranteeing a good approximation ratio.

Lemma 1 (PD-Small)**.** *Consider the* BagUFP *problem. There exists a procedure that considers only the small job instances and outputs a solution $S \subseteq \mathcal{U}_s$ such that $p(\mathrm{Opt}_s) \leqslant 9 \cdot p(S)$. The running time is $O(n^2)$.*

The PD-Small lemma is proved in Section 3. Here we highlight certain key aspects of the procedure given by the lemma. For the sake of clarity, we have stated the lemma for the case of $\gamma = 1/2$. However, it can be extended for any $\gamma > 0$, to derive an algorithm for handling γ-small job instances having an approximation ratio of $1 + \frac{4}{1-\gamma}$. Prior work on BagNbaUFP [21] and NbaUFP [19,18] also provide procedures for handling γ-small job instances. However, these procedures yield a good approximation ratio only when γ is set to a small value. In contrast, the PD-Small lemma achieves good approximation factors even for large values of γ (such as $\gamma = 1/2$). The advantage is that the complementary problem of handling γ-large job instances can be solved more efficiently and with better approximation ratios, leading to improved algorithms for `BagUFP`.

We note that the PD-Small lemma applies to the general `BagUFP` problem and does not require the NBA assumption. Our next goal is to handle the large job instances. For this purpose, we shall employ two different procedures, one for the general case and a second one for the special case where NBA applies. The lemma below deals with the general case.

Lemma 2. *Consider the* `BagUFP` *problem. There exists a procedure that considers only the large job instances and outputs a solution $S \subseteq \mathcal{U}_l$ such that $p(\mathrm{Opt}_l) \leqslant 16 \lceil \log 2n \rceil p(S)$. The procedure runs in time $O(n^2)$.*

The above procedure exploits a combinatorial lemma regarding large job instances, due to Bonsma et al. [10], that establishes a connection to the *Maximum Weight Independent Set of Rectangles* (`MWISR`) problem: given a set of rectangles with associated profits, find the maximum profit subset of non-overlapping rectangles [2,23,17,1]. For our purposes, we consider a generalization involving bag constraints and present a $(4\lceil \log 2n \rceil)$-approximation algorithm running in time $O(n^2)$, which may be of independent interest. Our algorithm goes via the notion of sequential k-independent graphs, studied by Akcoglu et al. [3], and Ye and Borodin [27]. Lemma 2 is proved in the full version.

Combining Lemma 2 with PD-Small lemma, we can get an $(9 + 16\lceil \log 2n \rceil)$-approximation to the overall optimal solution, establishing the following result.

Theorem 1. *There exists an $O(\log n)$-approximation algorithm for the* `BagUFP` *problem having running time of $O(n^2)$.*

The above result improves the previously best known approximation ratio of $O(\log \frac{c_{\max}}{c_{\min}})$ [14]. Obtaining a constant factor approximation algorithm for `BagUFP` remains an open question. The main issue arises in the handling of large job instances. However, prior work has shown that in the NBA setting, the large job instances can be handled via a simple reduction to the `WJISP` problem (see [16,19,21]). The `WJISP` problem can be approximated with a factor of 2 via the primal-dual method [7,9].

Lemma 3 ([21,7]). *There exists a procedure for* `BagNbaUFP` *that considers only the γ-large job instances and outputs a solution $S \subseteq \mathcal{U}_l$ such that $p(\mathrm{Opt}_l) \leqslant f \cdot p(S)$, where $f = \frac{4}{\gamma}(\frac{1}{\gamma} - 1)$. The procedure runs in time $O(n \log n)$.*

For the setting of $\gamma = 1/2$, the above lemma yields an 8-approximation procedure. Combining this with PD-Small lemma, we get an overall approximation ratio of 17 for the `BagNbaUFP` problem, improving upon the previously best known approximation ratio of 65 [21].

Theorem 2. *There exists an algorithm for the* `BagNbaUFP` *problem having an approximation ratio of* 17*. The algorithm runs in time $O(n^2)$.*

The PD-Small lemma provides an interesting corollary for the UFP problem. Bonsma et al. [10] devised a $(7+\epsilon)$-approximation algorithm running in polynomial time, albeit with a prohibitively large exponent in the polynomial. However, they showed that the running time can be improved to $O(n^4)$, at the cost of increasing the approximation ratio to $(25+\epsilon)$. Their algorithm also employs the strategy of classifying the input into small and large job instances, of which the small job instances are handled via a complex procedure based on randomized rounding. For the case of large job instances, they present a procedure that achieves an approximation ratio of $2/\gamma$, where γ is the largeness parameter; the procedure runs in time $O(n^4)$. We can obtain an alternative algorithm for UFP by employing PD-Small lemma in place of the former procedure. Setting $\gamma = 1/2$, we get an approximation ratio of 13.

Theorem 3. *There exists a 13-approximation algorithm for* UFP *running in time* $O(n^4)$.

3 Small Job Instances

Here, we establish Lemma 1 by presenting a 9-approximation algorithm for BagUFP on small job instances. We ignore all the large job instances and assume that the input set $\mathcal{U}$ consists only of small job instances. The algorithm is based on the primal-dual paradigm and builds on prior work on on distributed algorithms for the UFP problem [25,15,12]. However, the prior algorithms either deal with the simpler uniform capacity setting (wherein the capacity across all the timeslots is assumed to be the same) or provide logarithmic approximation factor. All the above primal-dual algorithms consider the job instance in a particular order and the main feature of our approach is to employ a more appropriate ordering. Our analysis exploits the new ordering in a crucial manner leading to constant factor approximations for the generic non-uniform setting.

The LP relaxation and its dual are presented next.

$$\begin{array}{rl|l}
 & \max \sum_{u\in\mathcal{U}} x(u)p(u) & \min \sum_{J\in\mathcal{J}} \alpha(J) + \sum_{t\in\mathcal{T}} c(t)\beta(t) \\
(\forall t\in\mathcal{T}) & \sum_{u\,:\,u\sim t} h(u)x(u) \leqslant c(t) & \alpha(J_u) + h(u) \sum_{t\,:\,u\sim t} \beta(t) \geqslant p(u) \\
(\forall J\in\mathcal{J}) & \sum_{u\in J} x(u) \leqslant 1 & \qquad (\forall u\in\mathcal{U})
\end{array}$$

The primal includes a variable $x(u)$ for each job instance $u \in \mathcal{U}$. The capacity and the bag constraints are enforced next. The dual includes a variable $\alpha(J)$ corresponding to the bag constraint of J, for each job J. Moreover, for each timeslot $t \in \mathcal{T}$, the dual includes variable $\beta(t)$ corresponding to the capacity constraint at t. For each job instance u, we include a constraint corresponding to the primal variable $x(u)$, which we call the *dual constraint of u*. For a job instance u, let J_u denote the job to which the instance u belongs. All the primal and dual variables are non-negative.

3.1 Algorithm

Our primal-dual algorithm uses a two-phase framework consisting of a forward phase and a reverse phase. The forward phase would construct a set of job instances $R \subseteq \mathcal{U}$ and a dual feasible solution $\alpha(\cdot)$ and $\beta(\cdot)$. The set R may not be a feasible solution. The reverse phase would delete certain job instances from R and construct a feasible solution $S \subseteq R$.

Forward Phase: We start by initializing all the dual variables to be zero and taking R to be the empty set. The algorithm would process the job instances in particular order and raise the dual variables in an appropriate manner. The ordering is cardinal to our algorithm in that it dictates the performance guarantee. We order the job instances in the decreasing order of their bottleneck capacities $\mathtt{bc}(u)$ and among the job instances having the same bottleneck capacity, the ordering is determined in the increasing order of ending timeslots (breaking ties arbitrarily). We denote the above ordering as σ.

The forward phase works iteratively, where the ith iteration would process the ith job instance in the ordering σ. Consider an iteration and let u be the job instance under processing. We check if the dual constraint of u is already satisfied and if so, we simply proceed to the next iteration. Otherwise, we shall raise certain dual variables suitably so that the constraint is satisfied, as described below. We first determine the *slackness* of the constraint, which is the difference between the RHS and the LHS of the constraint:

$$\text{slack}(u) = p(u) - \left(\alpha(J_u) + h(u) \sum_{t\,:\,u \sim t} \beta(t) \right). \tag{1}$$

We next select two specific timeslots t_ℓ and t_r from the span of u, as follows. Consider all the timeslots in the span of u having capacity at most $2\mathtt{bc}(u)$ and let t_ℓ be the left-most timeslot among them. Similarly, let t_r be the right-most timeslot among the timeslots satisfying the above property. Intuitively the span in between the timeslots t_ℓ and t_r is the essential span of the job instance u; beyond these timeslots, there is enough capacity. Call t_ℓ and t_r as the *left* and *right critical* timeslots of u, respectively.

We shall suitably raise the dual variables $\alpha(J_u)$, $\beta(t_\ell)$ and $\beta(t_r)$ so that the dual constraint is satisfied. Intuitively, we would like to satisfy two goals: (a) The dual objective value is not raised by much (since the dual is a minimization problem); (b) All the critical timeslots contribute an equal amount of increase in the dual objective value. With the above goals in mind, the dual variables for the critical timeslots are raised inversely proportional to the capacities at those timeslots, conforming to the intuition that timeslots with higher capacities are *less* critical. We choose a suitable value $\delta(u)$ and raise $\alpha(J_u)$ by $\delta(u)$, $\beta(t_\ell)$ by $4\frac{\delta(u)}{c(t_\ell)}$ and $\beta(t_r)$ by $4\frac{\delta(u)}{c(t_r)}$. The amount $\delta(u)$ is calculated so that the slack vanishes and the constraint becomes satisfied tightly i.e., LHS becomes equal to RHS. Namely, compute $\delta(u)$ satisfying the following equation:

$$\delta(u) \cdot \left(1 + 4h(u)\left[\frac{1}{c(t_\ell)} + \frac{1}{c(t_r)}\right]\right) = \text{slack}(u). \tag{2}$$

The job instance u is added to the set R. This completes an iteration of the first phase. We say that all the job instances in R are *raised.*

Reverse Phase: We consider the job instances in reverse order in which they were inserted into R and construct the solution S as follows. In any iteration of this phase, we look at the next job instance u (in the reverse order) and add u to S if doing so does not violate the capacity or the bag constraints. This phase continues until all of the job instances in R have been considered. The algorithm outputs the (feasible) solution S. This completes the description of the algorithm. A pseudocode can be found in the full version.

3.2 Analysis

Let us calculate the objective value of the dual solution constructed by the forward phase, denoted $val(\alpha, \beta)$ in terms of $\delta(\cdot)$. For any job instance $u \in R$, the dual variable $\alpha(J_u)$ is raised by $\delta(u)$ and this increases the dual objective value by $\delta(u)$. Similarly, we raise the dual variables corresponding to the two critical timeslots of u; namely, $\beta(t_\ell)$ is raised by $\frac{4\delta(u)}{c(t_\ell)}$ and $\beta(t_r)$ is raised by $\frac{4\delta(u)}{c(t_r)}$. Therefore, for each job instance $u \in R$, the dual objective value raises by an amount $9\delta(u)$. It follows that $val(\alpha, \beta) = 9 \sum_{u \in R} \delta(u)$ The lemma below provides a comparable lowerbound on the profit of the output solution S.

Lemma 4. *We have $p(S) \geqslant \sum_{u \in R} \delta(u)$.*

The lemma implies that $val(\alpha, \beta) \leq 9 \cdot p(S)$. Coupled with the weak duality theorem, we get that S is a 9-approximation to the optimal solution.

We proceed to Lemma 4. We shall associate a suitable quantity $\pi(u)$ with each job instance $u \in R$ such that $\pi(u) \geq \delta(u)$, and their overall sum satisfies $\sum_{u \in R} \pi(u) = p(S)$. Intuitively, $\pi(u)$ is the contribution made by u towards $p(S)$ (irrespective of whether or not u got picked in the final solution S).

For two job instances $u_1, u_2 \in R$, we say that u_1 is a *predecessor* of u_2, if u_1 appears before u_2 in the ordering σ; in this case, u_2 is said to be a *successor* of u_1. For a job instance $u \in R$, let pred(u) and succ(u) denote the set of all predecessors and successors of u, respectively. We consider a job instance as both predecessor and successor of itself.

Consider a job instance $u \in S$. Let LHS(u) be the variable denoting the LHS of the dual constraint of u:

$$\text{LHS}(u) = \alpha(J_u) + h(u) \sum_{t\,:\,u \sim t} \beta(t).$$

The variable LHS(u) would be zero in the beginning of the forward phase and it would keep increasing as the algorithm proceeds. At the end iteration in which u is raised, LHS(u) would be equal to $p(u)$ (since we ensured that the dual constraint of u is satisfied tightly). Thus, by tracking the variable LHS(u), we can derive a formula for $p(u)$ in terms of $\delta(\cdot)$ values of predecessors of u.

Consider a predecessor $u' \in \text{pred}(u)$. When u' is raised, three dual variables are increased, $\alpha(J_{u'})$, $\beta(t_\ell)$ and $\beta(t_r)$, where $J_{u'}$ is the job to which u' belongs, and t_ℓ and t_r are the left and right critical timeslots of u'. These increments will reflect as an increase in LHS(u), if the dual constraint of u also shares one or more of these variables. The increment in LHS(u) corresponding to the above three types are as follows:

– Type 1: If both u' and u belong to the same job, then the increment is $\delta(u')$.
– Type 2: If u is active at t_ℓ, the increment is $\frac{4h(u)\delta(u')}{c(t_\ell)}$.
– Type 3: If u is active at t_r, the increment is $\frac{4h(u)\delta(u')}{c(t_r)}$.

Notice that LHS(u) may increase via more than one type, in which case the total increment would be given by corresponding sum; if none of the cases occur, then the sum would be zero. We call the above sum as the *contribution of u' towards u* and denote it as $\lambda(u', u)$. The sum of contributions made by the predecessors of u yields the value of LHS(u) (as it stood at the end of the iteration in which u was raised), which is the same as $p(u)$.

The above discussion focuses on a job instance and analyzes the contributions made by the predecessors towards the instance. Conversely, we can fix a job instance u and consider the aggregate contribution that u makes towards its successors found in the solution S. We call the above aggregate quantity as the *total contribution* of u and denote it as $\pi(u)$: $\pi(u) = \sum_{u' \in \text{succ}(u) \cap S} \lambda(u, u')$. Notice that the profit $p(S)$ is given by the sum $\sum_{u \in R} \pi(u)$.

The quantity $\pi(u)$ can be computed by considering the three types of contributions discussed earlier. Let J_u be the job to which u belongs, and let t_ℓ and t_r be its left and right critical timeslots. Let $X = S \cap \text{succ}(u)$. Then,

$$\pi(u) = \sum_{u' \in X \;:\; u' \in J_u} \delta(u) + \sum_{u' \in X \;:\; u' \sim t_\ell} \frac{4h(u')\delta(u)}{c(t_\ell)} + \sum_{u' \in X \;:\; u' \sim t_r} \frac{4h(u')\delta(u)}{c(t_r)}$$

We next establish a lowerbound on total contribution of any raised job instance.

Lemma 5. *For any $u \in R$, $\pi(u) \geq \delta(u)$.*

The lowerbound implies Lemma 4. To prove the lowerbound, we fix a job instance $u \in R$ and analyze three cases. The first case is where u is picked in the solution S. In this case, u would contribute $\delta(u)$ towards itself (Type 1) and hence, $\pi(u) \geq \delta(u)$. So, assume that the reverse phase did not pick u for inclusion in S. This means that u could not be added to X, where $X \subseteq S$ is the set of successors of u found in S. The reason is that a bag constraint or a capacity constraint (or both) gets violated when u is added to X. Consider the first scenario, wherein

X contains some job instance u' that belongs to the same job as u. In this case, u would contribute $\delta(u)$ towards u' (Type 1) and hence, $\pi(u) \geq \delta(u)$.

We next analyze the last and the most interesting scenario, wherein the capacity constraint is violated at some timeslot $\widehat{t}$ found in the span of u, i.e.,

$$h(u) + \sum_{u' \in X \; : \; u' \sim \widehat{t}} h(u') > c(\widehat{t}). \tag{3}$$

If there are multiple such timeslots, choose the one having the minimum capacity (breaking ties arbitrarily). This is denoted by $\widehat{t}$ and is called the *conflict timeslot.*

Let $C \subseteq X$ be the set of job instances from X active at $\widehat{t}$; intuitively, C is the set of job instances that conflict with X at $\widehat{t}$ and prevent it from being included in X. Let t_ℓ and t_r be the left and right critical timeslots of u. We next make an important claim regarding any job instance $u' \in C$.

Lemma 6. *Any job instance $u' \in C$ must be active at t_ℓ or t_r (or both).*

The lemma is proved by exploiting the properties of the ordering σ. Intuitively, the argument is that the timeslots in the span of u outside of the range $[t_\ell, t_r]$ have too high a capacity to cause capacity constraint violation and hence, $\widehat{t}$ must lie within the range. Moreover, the ordering also implies that any job instance in C must start before u or end after u. The above two statements put together would imply the lemma. The lemma is proved in the full version.

Here, we assume the lemma and complete the proof of Lemma 5. Let A and B be the job instances in C that are active at t_ℓ and t_r, respectively. The lemma implies that every job instance in C is included in at least one of the two sets. Let us consider the quantity $\pi(u)$ and focus only on the terms corresponding to the job instances found in the two sets:

$$\pi(u) \geq \left(4\delta(u) \sum_{u' \in A} \frac{h(u')}{c(t_\ell)} \right) + \left(4\delta(u) \sum_{u' \in B} \frac{h(u')}{c(t_r)} \right)$$

The capacity at t_ℓ and t_r is at most twice the bottleneck capacity $\mathtt{bc}(u)$. So,

$$\pi(u) \geq \frac{2\delta(u)}{\mathtt{bc}(u)} \sum_{u' \in C} h(u'). \tag{4}$$

Since we are dealing with small job instances, $h(u) \leq \mathtt{bc}(u)/2$, which implies that $h(u) \leq c(\widehat{t})/2$. From (3), we get that

$$\sum_{u' \in C} h(u') \quad \geq \quad c(\widehat{t})/2 \quad \geq \quad \mathtt{bc}(u)/2.$$

Substituting in (4), we get that $\pi(u) \geq \delta(u)$. The proof of Lemma 5 is completed.

We conclude that the algorithm achieves an approximation guarantee of 9. It is not difficult to see that the algorithm can be implemented in time $O(n^2)$. This completes the proof of PD-Small lemma.

References

1. Adamaszek, A., Wiese, A.: Approximation schemes for maximum weight independent set of rectangles. In: FOCS (2013)
2. Agarwal, P., Kreveld, M., Suri, S.: Label placement by maximum independent set in rectangles. Computational Geometry 11(3-4), 209–218 (1998)
3. Akcoglu, K., Aspnes, J., Dasgupta, B., Kao, M.: Opportunity cost algorithms for combinatorial auctions. In: Kontoghiorghes, E., Rustem, B., Siokos, S. (eds.) Applied Optimization: Computational Methods in Decision-Making (2000)
4. Anagnostopoulos, A., Grandoni, F., Leonardi, S., Wiese, A.: A mazing $2 + \epsilon$ approximation for Unsplittable Flow on a Path. In: Proceedings of the Symposium on Discrete Algorithms, SODA 2014 (2014)
5. Bansal, N., Chakrabarti, A., Epstein, A., Schieber, B.: A quasi-PTAS for unsplittable flow on line graphs. In: STOC (2006)
6. Bansal, N., Friggstad, Z., Khandekar, R., Salavatipour, M.: A logarithmic approximation for unsplittable flow on line graphs. In: SODA (2009)
7. Bar-Noy, A., Bar-Yehuda, R., Freund, A., Naor, J., Schieber, B.: A unified approach to approximating resource allocation and scheduling. Journal of the ACM 48(5), 1069–1090 (2001)
8. Bar-Noy, A., Guha, S., Noar, J., Schieber, B.: Approximating the throughput of multiple machines in real-time scheduling. SICOMP 31(2), 331–352 (2001)
9. Berman, P., Dasgupta, B.: Multi-phase algorithms for throughput maximization for real-time scheduling. J. of Comb. Opt. 4, 307–323 (2000)
10. Bonsma, P., Schulz, J., Wiese, A.: A constant factor approximation algorithm for unsplittable flow on paths. In: FOCS (2011)
11. Calinescu, G., Chakrabarti, A., Karloff, H., Rabani, Y.: Improved approximation algorithms for resource allocation. In: Cook, W.J., Schulz, A.S. (eds.) IPCO 2002. LNCS, vol. 2337, pp. 401–414. Springer, Heidelberg (2002)
12. Chakaravarthy, V., Choudhury, A., Roy, S., Sabharwal, Y.: Distributed algorithms for scheduling on line and tree networks with non-uniform bandwidths. In: IPDPS (2013)
13. Chakaravarthy, V., Choudhury, A.R., Sabharwal, Y.: A near-linear time constant factor algorithm for unsplittable flow problem on line with bag constraints. In: FSTTCS (2010)
14. Chakaravarthy, V., Pandit, V., Sabharwal, Y., Seetharam, D.: Varying bandwidth resource allocation problem with bag constraints. In: IPDPS (2010)
15. Chakaravarthy, V., Roy, S., Sabharwal, Y.: Distributed algorithms for scheduling on line and tree networks. In: PODC (2012)
16. Chakrabarti, A., Chekuri, C., Gupta, A., Kumar, A.: Approximation algorithms for the unsplittable flow problem. Algorithmica 47(1), 53–78 (2007)
17. Chalermsook, P., Chuzhoy, J.: Maximum independent set of rectangles. In: SODA (2009)
18. Chekuri, C., Ene, A., Korula, N.: Unsplittable flow in paths and trees and column-restricted packing integer programs. In: Dinur, I., Jansen, K., Naor, J., Rolim, J. (eds.) Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques. LNCS, vol. 5687, pp. 42–55. Springer, Heidelberg (2009)
19. Chekuri, C., Mydlarz, M., Shepherd, F.: Multicommodity demand flow in a tree and packing integer programs. ACM Trans. on Algorithms 3(3) (2007)

20. Chuzhoy, J., Ostrovsky, R., Rabani, Y.: Approximation algorithms for the job interval selection problem and related scheduling problems. In: FOCS (2001)
21. Elbassioni, K., Garg, N., Gupta, D., Kumar, A., Narula, V., Pal, A.: Approximation Algorithms for the Unsplittable Flow Problem on Paths and Trees. In: FSTTCS (2012)
22. Erlebach, T., Spieksma, F.: Interval selection: Applications, algorithms, and lower bounds. J. Algorithms 46(1), 27–53 (2003)
23. Khanna, S., Muthukrishnan, S., Paterson, M.: On approximating rectangle tiling and packing. In: SODA (1998)
24. Kolliopoulos, S.: Edge-disjoint paths and unsplittable flow. In: Gonzalez, T. (ed.) Handbook of Approximation Algorithms and Metaheuristics, Chapman and Hall/CRC (2007)
25. Panconesi, A., Sozio, M.: Fast primal-dual distributed algorithms for scheduling and matching problems. Distributed Computing 22(4), 269–283 (2010)
26. Spieksma, F.: On the approximability of an interval scheduling problem. J. of Scheduling 2, 215–227 (1999)
27. Ye, Y., Borodin, A.: Elimination graphs. ACM Transactions on Algorithms 8(2), 14 (2012)

Nearly Tight Approximability Results for Minimum Biclique Cover and Partition

Parinya Chalermsook[1], Sandy Heydrich[1,3], Eugenia Holm[2], and Andreas Karrenbauer[1,*]

[1] Max Planck Institute for Informatics, Saarbrücken, Germany
[2] Dept. for Computer and Information Science, University of Konstanz, Germany
[3] Saarbrücken Graduate School of Computer Science
{andreas.karrenbauer,parinya.chalermsook,sandy.heydrich}@mpi-inf.mpg.de, eugenia.holm@uni-konstanz.de

Abstract. In this paper, we consider the *minimum biclique cover* and *minimum biclique partition* problems on bipartite graphs. In the minimum biclique cover problem, we are given an input bipartite graph $G = (V, E)$, and our goal is to compute the minimum number of complete bipartite subgraphs that cover all edges of G. This problem, besides its correspondence to a well-studied notion of *bipartite dimension* in graph theory, has applications in many other research areas such as artificial intelligence, computer security, automata theory, and biology. Since it is NP-hard, past research has focused on approximation algorithms, fixed parameter tractability, and special graph classes that admit polynomial time exact algorithms. For the minimum biclique partition problem, we are interested in a biclique cover that covers each edge exactly once.

We revisit the problems from approximation algorithms' perspectives and give nearly tight lower and upper bound results. We first show that both problems are NP-hard to approximate to within a factor of $n^{1-\varepsilon}$ (where n is the number of vertices in the input graph). Using a stronger complexity assumption, the hardness becomes $\tilde{\Omega}(n)$, where $\tilde{\Omega}(\cdot)$ hides lower order terms. Then we show that approximation factors of the form $n/(\log n)^{\gamma}$ for some $\gamma > 0$ can be obtained.

Our hardness results have many consequences: (i) $\tilde{\Omega}(n)$ hardnesses for computing the Boolean rank and non-negative integer rank of an n-by-n matrix (ii) $\tilde{\Omega}(n)$ hardness for minimizing the number of states in a deterministic finite automaton (DFA), given an n-state DFA as input, and (iii) $\tilde{\Omega}(\sqrt{n})$ hardness for computing minimum NFA from a truth table of size n. These results settle some of the most basic problems in the area of regular language optimization.

1 Introduction

We study the problem of covering the edges of a graph by bipartite complete subgraphs (or bicliques). In this problem, we are given a graph $G = (V, E)$, and our

[*]This work is partially supported by the Zukunftskolleg of the University of Konstanz, the DFG-grant KA 3042/3-1, and the Max Planck Center for Visual Computing and Communication (www.mpc-vcc.org).

objective is to compute a collection of complete bipartite subgraphs of G that together cover all edges of G, while minimizing the number of such subgraphs. The problem is referred to as the minimum biclique cover problem (BicliqueCover) in the optimization literature and has many applications, as well as connections, to other areas of computer science, such as automata and language theory [15], computer security [8], bioinformatics [24], graph drawing [9], and artificial intelligence. Besides these applications, computing a biclique cover of a graph is equivalent to other important notions in mathematics: Given an m-by-n matrix M over a Boolean algebra. The *Boolean rank* of M is the minimum k for which there exist two matrices $(A)_{m\times k}$ and $(B)_{k\times n}$ such that $M = AB$. It has been shown that computing Boolean rank of a matrix is equivalent to computing the bipartite dimension of a bipartite graph (see [14]).

In most applications, one may assume that graph G is bipartite. This problem has received a large amount of attention from a number of research groups. Since the problem is NP-hard, various approaches have been used in studying the problem: approximation algorithms [28,15], heuristics [8], fixed parameter tractability [25], and investigation of special graph classes that admit fast, polynomial-time algorithms [3,4,22,23].

Orlin showed that the problem is NP-hard, even on bipartite graphs [26]. Later Simon showed that the problem is also NP-hard to approximate [28]. Gruber and Holzer used the construction in [28] to show that the problem is $n^{1/3-\varepsilon}$ and $m^{1/5-\varepsilon}$ hard to approximate respectively. On an upper bound side, no non-trivial approximation algorithm has been proposed. The problem can be, however, solved efficiently in many cases. For instance, the fixed-parameter tractability result is known [25], implying that the problem can be solved in time $f(k)poly(n)$ provided that the biclique cover of size k exists. Also, the problem is polynomial time solvable in several graph classes, such as domino-free graphs, C4-free graphs, and bipartite permutation graphs (see [3] and references therein).

A problem closely related to BicliqueCover (but perhaps receives less attention from researchers) is called BicliquePartition where our goal is to find a cover in which each edge is covered by exactly one biclique. In contrast to BicliqueCover, only APX-hardness result has been shown for this problem. This result relies on the equivalence of BicliquePartition and the normal set basis problem shown to be NP-hard by [18], and a reduction from vertex cover.

A further related problem, called maximum edge biclique problem (MaxBiclique), receives a lot of attention from approximation algorithms community. Dawande, Keskinocak and Tayur [21] showed that the weighted version of the MaxBiclique in bipartite graphs is NP-complete, but they were not able to show that the unweighted version is hard also. This was later accomplished by Peeters [27] who proved that MaxBiclique in bipartite graphs is NP-complete. In terms of approximation hardness, Feige [10] shows that the problem is hard to within a factor of n^{ε} assuming average-case complexity hypothesis. Ambühl et al. prove the same result under a more standard assumption [1].

1.1 Our Contributions

Our main result is informally summarized in the following theorem.

Theorem 1 (Informal). BicliqueCover *and* BicliquePartition *on bipartite graphs are (almost) as hard to approximate as graph coloring.*

Combining this theorem with the hardness results for graph coloring [12,20,29] implies that these problems do not admit $n^{1-\varepsilon}$ and $m^{1/2-\varepsilon}$ approximation algorithm unless $\mathsf{P} = \mathsf{NP}$. With a stronger complexity assumption of $\mathsf{NP} \not\subseteq \mathsf{BPTIME}(2^{poly \log n})$, this gives a stronger hardness result of $\frac{n}{2^{\log^{7/8+\varepsilon} n}}$ and $\frac{\sqrt{m}}{2^{\log^{7/8+\varepsilon} m}}$ for any $\varepsilon > 0$. (For the purpose of deriving our corollaries, it is important to state the bounds in terms of both m and n).

We immediately obtain the hardness of approximating the rank of a matrix through the connections shown in [14]. Also, Amilhastre et al. and Gruber and Holzer [2,15] discovered (nearly tight) connections between BicliqueCover, BicliquePartition, and several minimization problems for regular languages. Combining our result with theirs yields new hardness results (proofs will appear in the full version). We summarize the consequences of our theorem below.

Corollary 1. *Unless* NP *has bounded-error randomized quasi-polynomial time algorithm, for all* $\varepsilon > 0$*, it is hard to:*

– *Approximate the Boolean rank and non-negative integer rank of an n-by-n matrix to within a factor of* $\frac{n}{2^{\log^{7/8+\varepsilon} n}}$.
– *Approximate the number of states of minimum NFA accepting a language L, specified by an input truth table of size N, to within a factor of* $\frac{\sqrt{N}}{2^{\log^{7/8+\varepsilon} N}}$.
– *Approximate the minimum number of states of the minimum DFA accepting a language L, specified by an input n-state DFA of size n, to within a factor of* $\frac{n}{2^{\log^{7/8+\varepsilon} n}}$.

All these results are essentially tight. These problems are some of the most basic problems in regular language minimization (see the survey by Holzer and Kutrib and references therein [17]). Prior to our results, similar hardness results require (much stronger) cryptographic assumptions [13]. We remark another interesting aspect of our results: It is noted in [17] that the lower bounds provided by biclique edge cover technique " *... are not always tight and can be arbitrarily worse ...*" Our results show that biclique cover techniques can in fact provide tight lower bounds for many problems listed in the survey, hence providing an evidence that biclique cover and partition capture the computational complexity of regular language minimization problems.

Our proof follows the framework of graph product techniques, as introduced and used succesfully by Chalermsook et al. [5,7,6]. Roughly speaking, this framework reduces the task of proving hardness of approximation to that of proving graph product inequalities. In our case, this amounts to bounding the quantity $bc(B[G \cdot H])$, by some slowly growing function of $bc(B[H])$ and $bc(B[G])$ where $bc(H)$ denotes the size of minimum biclique cover of H, "$\cdot$" is the lexicographic product of graphs, and $B[\cdot]$ is the *bipartite double cover* transformation respectively. The main idea of the proof is to use an optimal vertex coloring of $\bar{G}$ together with biclique covering of $B[H]$ to suggest the biclique cover of $B[G \cdot H]$. We note that, while we give lower bound results, the flavor of our proofs is rather algorithmic: It illustrates how one can algorithmically utilize the coloring of graph $\bar{G}$ in minimizing the biclique covers in $B[G^k]$.

Our hardness results rule out approximation ratios n^δ for any $\delta \in (0,1)$, so it is natural to aim at mildly sub-linear approximation factors, e.g., $\frac{n}{(\log n)^\gamma}$ for some $\gamma > 0$. We investigate this direction and obtain the following results.

Theorem 2. *There is an approximation algorithm for* BicliqueCover *that achieves an approximation ratio of*

$$O\left(\min\left\{n/\sqrt{\log(n)}, m(\log\log m)^2/(\log^3 m)\right\}\right).$$

We remark that the upper and lower bounds match up to lower-order factors (in terms of n). The second result relies on the idea that one can reduce BicliqueCover to MaxClique on the complement of the conflict graph.

Using a standard reduction, we furthermore obtain the following result.

Corollary 2. *There is no poly-time algorithm to approximate* MaxWeightedBiclique *within factors of $n^{1-\varepsilon}$ and $m^{1/2-\varepsilon}$, respectively, for all $\varepsilon > 0$ unless $P = NP$, or within a factor of $O\left(\frac{\min\{n,\sqrt{m}\}}{2^{\log^{7/8+\varepsilon} n}}\right)$ for any $\varepsilon > 0$ unless $\mathsf{NP} \subseteq \mathsf{BPTIME}(2^{poly \log n})$. This holds even when edge-weights are in $\{0,1\}$.*

2 Preliminaries

We start by a formal treatment of our problem. A *biclique* is denoted by $K_{a,b}$ which is a complete bipartite graph (A, B, F) such that $|A| = a$ and $|B| = b$. Given a graph $G = (V, E)$, we say that $S \subseteq V$ is a *biclique subgraph* of G if and only if the induced subgraph $G[S]$ is a biclique $K_{a,b}$ for some a, b.

A *biclique cover* of G is a collection of vertices $S_1, \ldots, S_k$ such that each S_i is a biclique subgraph of G and each edge $e \in E(G)$ appears at least once in some $G[S_i]$. In such case, we say that a biclique cover of size k exists for G. Let $bc(G)$ denote the minimum number k for which a biclique cover of size k exists for G. In BicliqueCover, our goal is to compute $bc(G)$ on an input graph G. A *biclique partition* of G is a biclique cover such that, each edge is covered exactly once. It follows from the definition that $bc(G) \leq bp(G)$ for any graph G.

A *clique partition* of G is a partition of vertices $V(G)$ into $V(G) = V_1 \cup V_2 \cup \ldots \cup V_k$ such that each induced subgraph $G[V_i]$ is a clique. The *clique partition number* of G, denoted by $cp(G)$, is the minimum number k such that a clique partition of $V(G)$ into k components exist. The clique partition problem (PartitionIntoCliques) asks for computing the value of $cp(G)$.

Given a graph G, let $\chi(G)$ be the *chromatic number* of G which is the minimum number of colors c such that there exists a proper c-coloring of G. Let $\mathcal{I}_G$ be the set of all independent sets in G. A valid fractional c-coloring of G is an assignment $\psi : \mathcal{I}_G \to [0,1]$ with the guarantees: (i) $\sum_{S:v\in S} \psi(S) \geq 1$ for all v and (ii) $\sum_{S\in\mathcal{I}_G} \psi(S) \leq c$. A *fractional chromatic number* of G, $\chi_f(G)$, is the minimum c such that there exists a valid fractional c-coloring for G.

Notice that for any graph G, we have $\chi(G) = cp(\bar{G})$. Similarly to the notion of fractional chromatic number, we may define *fractional clique partition number* $cp_f(G)$ as $\chi_f(\bar{G})$. This implies that $\frac{cp(G)}{\log|V(G)|} \leq cp_f(G) \leq cp(G)$.

Feige and Kilian [12] proved the NP-hardness of approximating $\chi(G)$. Since $\chi(G) = cp(\bar{G})$, the same hardness result holds for PartitionIntoCliques. Their result can be summarized formally below.

Theorem 3 ([12,29]). *Let $\varepsilon > 0$ be a constant. Given a graph $G = (V,E)$, it is NP-hard to approximate $cp(G)$ to within a factor of $|V(G)|^{1-\varepsilon}$.*

Assuming a stronger (but still standard) complexity theoretic assumption, Khot and Ponnuswami proved the following result [20].

Theorem 4. *Let $\varepsilon > 0$ be a constant. It is hard to approximate $cp(G)$ for a graph $G = (V,E)$ to within a factor of $\frac{|V(G)|}{2^{\log^{3/4+\varepsilon}|V(G)|}}$ unless $\mathsf{NP} \subseteq \mathsf{BPTIME}(2^{poly\log n})$.*

3 Hardness of Approximation

In this section, we prove our hardness results. We start by explaining graph product terminologies and tools in the next subsection.

3.1 Graph Products

Let G and H be any graphs. The lexicographic product of G and H, i.e. $G\cdot H$, is defined as follows. The vertex set of $G\cdot H$ is $V(G\cdot H) = V(G)\times V(H)$ and the edge set is $E(G\cdot H) = \{(u,a)(v,b) : uv \in E(G)\} \cup \bigcup_{u\in V(G)} \{(u,a)(u,b) : ab \in E(H)\}$. For an integer k, the term G^k denotes a k-fold lexicographic product of G, i.e. $G^k = G\cdot G \ldots \cdot G$ (k times). The following inequality is a standard fact.

Lemma 1. *For any graphs G and H, $\chi_f(G)\chi(H) \leq \chi(G\cdot H) \leq \chi(G)\chi(H)$*

We show that the clique partition number satisfies similar properties with respect to lexicographic products. The proof will appear in the full version.

Lemma 2 (Multiplicativity of cp). $cp_f(G)cp(H) \leq cp(G\cdot H) \leq cp(G)cp(H)$

3.2 Proof of the Hardness Result

We prove the following connection between PartitionIntoCliques and BicliqueCover, which will be used in deriving our hardness results.

Theorem 5. *Let G be any graph and k be an integer. There is an algorithm that runs in time $|V(G)|^{O(k)}$ and constructs a bipartite graph H such that $|V(H)| = \Theta(|V(G)|^k)$ and*

$$\left(\frac{cp(G)}{\log |V(G)|}\right)^k \leq bc(H) \leq bp(H) \leq cp(G)^k |V(G)|^3$$

Before proving this theorem, we show how to use it to derive our hardness results.

Corollary 3. *Let $\varepsilon > 0$. It is NP-hard to approximate* BicliqueCover *and* BicliquePartition *within factors of $n^{1-\varepsilon}$ and $m^{1/2-\varepsilon}$. Moreover, there are no polynomial time approximation algorithms for both problems with a guarantee in $\frac{n}{2^{\log^{7/8+\varepsilon} n}}$ or $\frac{\sqrt{m}}{2^{\log^{7/8+\varepsilon} n}}$ unless $\mathsf{NP} \subseteq \mathsf{BPTIME}(2^{poly \log n})$.*

Proof. Our reduction combines the reduction that gives hardness result PartitionIntoCliques with Thm. 5. Let $\mathcal{A}_{clique}$ be the algorithm (i.e. reduction) that takes a SAT instance φ and produces graph G, with the following properties:

– (Yes-Instance:) If φ is satisfiable, then $cp(G) \leq c$
– (No-Instance:) If φ is not satisfiable, then $cp(G) \geq s$.

Let $g = s/c$ be the gap (hardness factor) given by the reduction $\mathcal{A}_{clique}$. For instance, Thm. 3 gives such a reduction with $c = |V(G)|^{\varepsilon}$, $s = |V(G)|^{1-\varepsilon}$, $g = |V(G)|^{1-2\varepsilon}$, and $|V(G)| = |\varphi|^{O(1)}$. Our reduction $\mathcal{A}^k_{biclique}$ first runs the algorithm $\mathcal{A}_{clique}$ to get the instance G and then apply Thm. 5 on graph G. The theorem outputs graph H with $N = |V(H)| = \Theta(|V(G)|^k)$.

Now analyze the gap given by our reduction $\mathcal{A}^k_{biclique}$. Applying the lower bound of Thm. 5, for the No-Instance, we get $bc(H), bp(H) \geq \frac{s^k}{(\log |V(G)|)^k}$. For the Yes-Instance, we would get $bc(H), bp(H) \leq c^k |V(G)|^3$. So the gap between Yes-Instance and No-Instance of reduction $\mathcal{A}^k_{biclique}$ is

$$g' = \left(\frac{s}{c}\right)^k \frac{1}{|V(G)|^3 (\log |V(G)|)^k} = \frac{g^k}{|V(G)|^3 (\log |V(G)|)^k}$$

This gap holds for both BicliquePartition and BicliqueCover. Roughly speaking the gap between our Yes-Instance and No-Instance is $g' \approx g^k$. Now we plug in the appropriate values to obtain the desired hardness results.

If we start from Thm. 3, we have the starting hardness gap $g = |V(G)|^{1-2\varepsilon}$. By choosing $k = \lceil 1/\varepsilon \rceil$, we obtain a gap of $g' \geq |V(G)|^{(1-2\varepsilon)k} / |V(G)|^4 \geq |V(G)|^{(1-6\varepsilon)k}$. Since $N = |V(H)| = |V(G)|^k$, this gives us the hardness factor $N^{1-6\varepsilon}$, thus proving the first part of the theorem. This reduction runs in time $|V(G)|^{O(1/\varepsilon)} = |\varphi|^{O(1)}$ for constant $\varepsilon > 0$ (since Feige-Kilian reduction runs in polynomial time), thus implying that the hardness result here holds under assumption $\mathsf{P} \neq \mathsf{NP}$.

Similarly, if we start from Thm. 4, we have $g = \frac{n}{2^{\log^{3/4+\varepsilon} n}}$ where $n = |V(G)|$. We plug in the value of g into $g' = g^k / n^3 (\log n)^k$. By choosing $k = \log n$, we have $g' \geq g^k/n^{\Theta(\log\log n)} \geq \left(\frac{n}{2^{\log^{3/4+2\varepsilon} n}}\right)^k = \frac{N}{2^{k\log^{3/4+2\varepsilon} n}}$. Since $k = \log n$, we have $\log N = O(k \log n) = O(\log^2 n)$. We obtain the hardness factor $g' \geq \frac{N}{2^{\log^{7/8+O(\varepsilon)} N}}$. The reduction here runs in time $|V(G)|^{O(k)} = |V(G)|^{O(\log|V(G)|)}$. Khot-Ponnuswami reduction has $|V(G)| = 2^{\text{poly}\log|\varphi|}$, and it is randomized with possibly two-sided error. This implies that the running time of the reduction overall is $2^{\text{poly}\log|\varphi|}$. Therefore, this hardness result holds under the assumption that NP does not admit randomized quasi-polynomial time algorithm.

The statements w.r.t. the number of edges follow since $|E(H)| \leq N^2$. □

The rest of this section is devoted to proving Thm. 5. We use a *bipartite double cover* transformation, which transforms any graph G into a bipartite graph $B[G]$ as follows. The nodes of $B[G]$ are $V(B[G]) = \bigcup_{v\in V(G)}\{(v,1),(v,2)\}$, i.e. we make two copies of each vertex $v \in V(G)$. The edges of $B[G]$ are $E(B[G]) = \{(u,1)(v,2) : uv \in E(G)\} \cup \{(u,1)(u,2) : u \in V(G)\}$. Our algorithm simply outputs $H = B[G^k]$. Notice that $|V(H)| = 2|V(G)|^k$.

First let us show the lower bound, which is relatively straightforward to see.

Lemma 3. *For any graph G, $cp(G) \leq bc(B[G])$*

Proof. Let $S_1, \ldots, S_\ell \subseteq V(B[G])$ be the biclique subgraphs that cover $B[G]$. It is sufficient to show how to use these bicliques to define the partition of G into ℓ cliques. We name the biclique $H_j = G[S_j]$. For each j, we define the vertex set $V_j \subseteq V(G)$ by $V_j = \{v : (v,1)(v,2) \in E(H_j)\}$. First we argue that $G[V_j]$ is a clique in G: Consider $u, v \in V_j$ for some $u \neq v$. Since $(u,1)(u,2),(v,1)(v,2) \in E(H_j)$, it must be the case that $(u,1)(v,2) \in E(H_j)$, implying that $uv \in E(G)$. Moreover, the collection of cliques $V_1, \ldots, V_\ell$ together cover graph G: For each vertex $v \in V(G)$, an edge $(v,1)(v,2)$ must appear in some $H_{j'}$ (due to the fact that $S_1, \ldots, S_\ell$ are biclique cover). This means that $v \in V_{j'}$. From a collection of cliques $V_1, \ldots, V_\ell$, one can easily modify them into disjoint sets $V'_1, \ldots, V'_\ell$. □

It is easy to see that this inequality implies the lower bound: consider $H = B[G^k]$, so we have $bc(H) \geq cp(G^k) \geq (cp_f(G))^k \geq \left(\frac{cp(G)}{\log|V(G)|}\right)^k$.

Now we need to prove the upper bound that $bp(H) \leq cp(G)^k|V(G)|^3$. We present here a "light" version of our proof, showing a weaker statement that $bc(H) \leq cp(G)^k|V(G)|^3$. This proof captures most of the key ideas we need. The proof of the stronger statement will be contained in the full version.

Lemma 4. *For any graphs G and G', $bc(B[G \cdot G']) \leq 2|E(G)| + cp(G)bc(B[G'])$*

Now we can apply Lem. 4 iteratively to get the following, which completes the proof of Thm. 5.

Lemma 5. *For any graph G and integer k, $bc(B[G^k]) \leq k|V(G)|^2 cp(G^k)$.*

Proof. We will argue by induction on r that $bc(B[G^r]) \leq r|V(G)|^2 cp(G^r)$. Notice that this is true for the base case when $r = 1$, i.e. $bc(B[G]) \leq |V(G)|^2 cp(G)$, because the biclique cover number of any graph is at most the number of edges in it. Now assume that the hypothesis holds for all integers up to r. By unfolding the term G^{r+1} as $G \cdot G^r$, we can write $bc(B[G^{r+1}])$ as $bc(B[G^{r+1}]) \leq 2|E(G)| + cp(G)bc(B[G^r])$. Applying the induction hypothesis to the second term, we get

$$\begin{aligned} bc(B[G^{r+1}]) &\leq 2|E(G)| + cp(G)r|V(G)|^2 cp(G)^r \\ &\leq |V(G)|^2 + r \cdot cp(G)^{r+1}|V(G)|^2 \\ &\leq (r+1)cp(G)^{r+1}|V(G)|^2 \end{aligned}$$

This implies the proof of the statement. □

3.3 Proof of Lemma 4

Recall the statement of the lemma, that $bc(B[G \cdot G']) \leq 2|E(G)| + cp(G)bc(B[G'])$. Let $S_1, \ldots, S_h \subseteq V(B[G'])$ be the biclique cover of $B[G']$. For each S_j, we use G'_j to denote the induced subgraph of S_j in $B[G']$ (so G'_j is a clique). We will use these graphs to "suggest" the cover for $B[G \cdot G']$. First, we look at the edges $E(B[G \cdot G'])$ as the union of two edge sets $E_1 \cup E_2$ where

$$E_1 = \{(u,a,1)(v,b,2) : u \neq v, uv \in E(G)\}$$

and

$$E_2 = \bigcup_{u \in V(G)} \{(u,a,1)(u,b,2) : a = b \vee ab \in E(G')\}.$$

To cover edges in E_1, we define the collection of vertices $\{X_{uv}\}_{uv \in E(G)}$ as $X_{uv} = \{(u,a,1) : a \in V(G')\} \cup \{(v,b,2) : b \in V(G')\}$. Notice that each X_{uv} is a biclique subgraph of $B[G \cdot G']$: For each pair $(u,a,1)$ and $(v,b,2)$ in X_{uv}, since $uv \in E(G)$, there must be an edge $(u,a,1)(v,b,2)$. Thus, the following claim holds.

Claim. The collection $\{X_{uv}\}_{uv \in E(G)}$ covers all edges in E_1.

Now we define another collection of bicliques $\{Y_{c,j}\}$ to cover edges in E_2 as follows. Let $C_1, \ldots, C_\ell$ be the partition of vertices of G into cliques. For each clique $c = 1, \ldots, \ell$, for each $j = 1, \ldots, h$, define a subset of vertices $Y_{c,j} \subseteq V(B[G \cdot G'])$ where $Y_{c,j} = \{(u,a,1) : u \in C_c, (a,1) \in S_j\} \cup \{(u,b,2) : u \in C_c, (b,2) \in S_j\}$. Now we verify that the induced subgraph of each $Y_{c,j}$ is biclique: For any pair of vertices $(u,a,1), (v,b,2) \in Y_{c,j}$,

- If $u = v$, then it must hold that $(a,1)(b,2) \in E(G'_j)$ (because both $(a,1)$ and $(b,2)$ belong to biclique S_j). There are two cases again. If $a = b$, we have an edge $(u,a,1)(u,a,2) \in B[G \cdot G']$ by definition; otherwise, if $a \neq b$, there must be an edge $ab \in E(G')$, implying that $(u,a,1)(u,b,2)$ is an edge in $B[G \cdot G']$.
- If $u \neq v$, the fact that both u and v belong to the same clique C_c means that an edge $uv \in E(G)$, implying that $(u,a,1)(v,b,2)$ is an edge in $B[G \cdot G']$.

Claim. The collection of bicliques $Y_{c,j}$ covers all edges in E_2.

Proof. Fix some $u \in V(G)$. Consider an edge $(u,a,1)(u,b,2) \in E_2$. Let C_c be the clique that contains vertex u. Since $ab \in E(G')$ or $a = b$, we have $(a,1)(b,2)$ as an edge in $B[G']$. Therefore, it is covered by some biclique G'_j, i.e. $(a,1),(b,2) \in S_j$. This implies that both $(u,a,1)$ and $(u,b,2)$ belong to $Y_{c,j}$, hence covered. □

4 Algorithmic Results

We will now give two approximation algorithms for BicliqueCover. Thereby, we achieve two mutually non-dominating approximation guarantees in terms of the number of nodes and edges, respectively.

4.1 An Approximation Guarantee of $O(n/\sqrt{\log(n)})$

We first describe a simple approximation algorithm for BicliqueCover that achieves a performance ratio of $O(n_U/\sqrt{\log(n_U)})$ where n_U is the number of left vertices in the bipartite input graph $G = (U \cup V, E)$ (we assume w.l.o.g that the left side of the graph is the smaller one, i.e. $|U| \leq |V|$). Moreover, we will apply exactly the same scheme to solve BicliquePartition, thereby achieving the same performance guarantee for BicliquePartition.

The main idea behind the algorithm is to split the left vertex set U in parts of equal size r (to be fixed later) and run an $\alpha(r)$-approximation algorithm for finding a biclique cover in each of these subgraphs. The results of all n_U/r subproblems are then put together to form a biclique cover of the whole graph G. This also works for biclique partition, as the subgraphs are edge-disjoint. The following theorem relates the approximation guarantee for the subproblems to the guarantee for the overall problem.

Lemma 6. *Let $G = (U, V, E)$ be a bipartite graph with $n_U = |U| \leq |V|$. If we can solve* BicliqueCover *on a graph G' with r left vertices with an approximation guarantee of $\alpha(r)$, then we can solve the problem on G with approximation guarantee $\frac{n_U}{r}\alpha(r)$. The same holds for* BicliquePartition.

Proof. Partition U arbitrarily into n_U/r sets $U_1, \ldots, U_{n_U/r}$ of size r and run the approximation algorithm with performance guarantee $\alpha(r)$ on the subgraphs induced by the sets U_i and their neighborhoods. Let G_i denote the i-th subgraph and APX_i the size of the solution produced by the approximation algorithm on G_i. Furthermore, let OPT_i be the size of the optimal solution on subgraph G_i and OPT be the size of the optimal solution for G. Notice that the union of the biclique covers of the subgraphs gives a biclique cover for G. Therefore, we have that the size of this combined solution is $APX = \sum_{i=1}^{n_U/r} APX_i \leq \alpha(r) \sum_{i=1}^{n_U/r} \mathsf{OPT}_i \leq \alpha(r) \frac{n_U}{r} \mathsf{OPT}$. The last inequality follows as the optimal solution of a subgraph of G is at most as large as the optimal solution of G. This analysis also applies to BicliquePartition. □

Theorem 6. *There are $O(n/\sqrt{\log n})$ approximation algorithms for* Biclique-Cover *and* BicliquePartition.

Proof. For solving the subproblems on $G' = (U', V', E')$ with $r = |U'|$ left vertices, we run a brute-force algorithm: Enumerate all 2^r subsets of the left vertices and enumerate all r-tuples of such subsets. Such a subset $S \subseteq U'$ induces a biclique together with the intersection of the neighborhoods of all vertices $v \in S$. Then return the smallest tuple of vertex sets that covers all edges.

For BicliquePartition, additionally ensure that the bicliques are edge-disjoint. As the optimal solution needs at most r bicliques (simply take all the bicliques induced by one of the left vertices) and we enumerate all bicliques of the graph by enumerating all subsets of left vertices, this will return the optimal solution. Hence the approximation factor on the subproblems is $\alpha(r) = 1$. Thus, for the whole algorithm on G, we get a guarantee of $\frac{n_U}{r} \leq \frac{n}{r}$. The running time of the brute-force algorithm is $O((2^r)^r)$, hence by choosing $r = \sqrt{\log(n)}$ we get a polynomial running time of the algorithm and a guarantee of $O(n/\sqrt{\log(n)})$. □

4.2 An Approximation Guarantee w.r.t. the Number of Edges

A different approach to obtain an approximation guarantee, which dominates the previous one on sparse graphs, is obtained via the following construction.

Definition 1. *For a given undirected graph $G = (V, E)$, the* conflict graph *$\mathcal{G} = (\mathcal{V}, \mathcal{E})$ contains a node for each edge of G, i.e. $|\mathcal{V}| = |E|$. Two nodes of $\mathcal{G}$ are connected by an edge if and only if the two corresponding edges of G are not contained in a common biclique.*

A node coloring of $\mathcal{G}$ corresponds to an edge coloring of G such that each color-class is contained in a common biclique. Thus, the chromatic number of $\mathcal{G}$ is equal to $bc(G)$ and we can use [16] to obtain a guarantee in $O\left(\frac{m \log^2 \log m}{\log^3 m}\right)$. Together with Thm. 6, this concludes the proof of Thm. 2.

However, we present another perspective, which not only gives algorithmic insights but also leads to an improved hardness result for MaxWeightedBiclique. To this end, recall that the chromatic number of $\mathcal{G}$ is equal to $cp(\overline{\mathcal{G}})$, so that a greedy algorithm that covers $\overline{\mathcal{G}}$ with cliques also covers G with bicliques.

Thus, we analyze the family of greedy algorithms that pick a biclique in each iteration containing as many uncovered edges as possible until every edge is covered. To this end, we reexamine the relation between master and slave problems in Johnson's framework [19] under the premise that the approximation guarantee $\alpha(\cdot)$ is an increasing function on the number of uncovered elements. That is, the approximation guarantee improves over iterations as the number of uncovered edges shrinks. Hence, our master problem is BicliqueCover and its slave problem is the problem of finding the heaviest biclique with edge-weights in $\{0, 1\}$ being 0 if an edge is already covered and 1 if not. Our result is summarized in the following theorem, whose proof will appear in the full version of this paper.

Theorem 7. *Let $G = (V, E)$ be a bipartite graph. If there is an α-approximation algorithm for* MaxWeightedBiclique*, then there is a greedy algorithm that computes a* BicliqueCover *of size*

$$O\left(\alpha \log\left(\frac{|E|}{\alpha}\right) bc_f(G)\right),$$

where $bc_f(G)$ is the fractional biclique cover number.

Such an α-approximation can be obtained from an approximation algorithm for MaxClique operating on the complement of a conflict graph. By dropping all the nodes of $\overline{\mathcal{G}}$ that correspond to edges with weight 0 and finding an approximation of the largest clique in the remainder, we obtain a set of edges of G that belongs to a common biclique, which has a weight of at least the maximum weight divided by α. Using the MaxClique algorithm of Feige [11], we obtain an approximation factor for $\{0,1\}$-weighted biclique in $O\left(\frac{m\log^2\log m}{\log^3 m}\right)$. This is also essentially the best one can hope for as our new hardness result shows.

Corollary 2. *There is no poly-time algorithm to approximate* MaxWeightedBiclique *within factors of $n^{1-\varepsilon}$ and $m^{1/2-\varepsilon}$, respectively, for all $\varepsilon > 0$ unless $P = NP$, or within a factor of $O\left(\frac{\min\{n,\sqrt{m}\}}{2^{\log^{7/8+\varepsilon} n}}\right)$ for any $\varepsilon > 0$ unless $\mathsf{NP} \subseteq \mathsf{BPTIME}(2^{poly\log n})$. This holds even when edge-weights are in $\{0,1\}$.*

A further consequence of Thm. 7 is that $bc(G) = O(\log(n)bc_f(G))$, which yields the following corollary.

Corollary 4. *It is NP-hard to approximate the fractional biclique number within $n^{1-\varepsilon}$ or $m^{1/2-\epsilon}$ for all $\varepsilon > 0$.*

References

1. Ambühl, C., Mastrolilli, M., Svensson, O.: Inapproximability results for maximum edge biclique, minimum linear arrangement, and sparsest cut. SIAM J. Comput. 40(2), 567–596 (2011)
2. Amilhastre, J., Janssen, P., Vilarem, M.-C.: FA minimisation heuristics for a class of finite languages. In: Boldt, O., Jürgensen, H. (eds.) WIA 1999. LNCS, vol. 2214, pp. 1–12. Springer, Heidelberg (2001)
3. Amilhastre, J., Vilarem, M., Janssen, P.: Complexity of minimum biclique cover and minimum biclique decomposition for bipartite domino-free graphs. Discrete Applied Mathematics 86(2-3), 125–144 (1998)
4. Brandstädt, A., Le, V.B., Spinrad, J.P.: Graph Classes: A Survey. Society for Industrial and Applied Mathematics, Philadelphia (1999)
5. Chalermsook, P., Laekhanukit, B., Nanongkai, D.: Graph products revisited: Tight approximation hardness of induced matching, poset dimension and more. In: Khanna, S. (ed.) SODA, pp. 1557–1576. SIAM (2013)
6. Chalermsook, P., Laekhanukit, B., Nanongkai, D.: Independent set, induced matching, and pricing: Connections and tight (subexponential time) approximation hardnesses. In: FOCS, pp. 370–379. IEEE Computer Society (2013)
7. Chalermsook, P., Laekhanukit, B., Nanongkai, D.: Coloring graph powers: Graph product bounds and hardness of approximation. In: Pardo, A., Viola, A. (eds.) LATIN 2014. LNCS, vol. 8392, pp. 409–420. Springer, Heidelberg (2014)
8. Ene, A., Horne, W., Milosavljevic, N., Rao, P., Schreiber, R., Tarjan, R.E.: Fast exact and heuristic methods for role minimization problems. In: SACMAT 2008, pp. 1–10. ACM, New York (2008)
9. Eppstein, D., Goodrich, M.T., Meng, J.Y.: Confluent layered drawings. Algorithmica 47(4), 439–452 (2007)

10. Feige, U.: Relations between average case complexity and approximation complexity. In: Reif, J.H. (ed.) STOC, pp. 534–543. ACM (2002)
11. Feige, U.: Approximating maximum clique by removing subgraphs. SIAM Journal on Discrete Mathematics 18(2), 219–225 (2004)
12. Feige, U., Kilian, J.: Zero knowledge and the chromatic number. Journal of Computer and System Sciences 57, 187–199 (1998)
13. Gramlich, G., Schnitger, G.: Minimizing NFA's and regular expressions. J. Comput. Syst. Sci. 73(6), 908–923 (2007)
14. Gregory, D.A., Pullman, N.J., Jones, K.F., Lundgren, J.R.: Biclique coverings of regular bigraphs and minimum semiring ranks of regular matrices. J. Comb. Theory, Ser. B 51(1), 73–89 (1991)
15. Gruber, H., Holzer, M.: Inapproximability of nondeterministic state and transition complexity assuming P $\neq$ NP. In: Harju, T., Karhumäki, J., Lepistö, A. (eds.) DLT 2007. LNCS, vol. 4588, pp. 205–216. Springer, Heidelberg (2007)
16. Halldórsson, M.M.: A still better performance guarantee for approximate graph coloring. Information Processing Letters 45(1), 19–23 (1993)
17. Holzer, M., Kutrib, M.: Descriptional and computational complexity of finite automata–a survey. Information and Computation 209(3), 456–470 (2011)
18. Jiang, T., Ravikumar, B.: Minimal NFA problems are hard. In: Albert, J.L., Monien, B., Rodríguez-Artalejo, M. (eds.) ICALP 1991. LNCS, vol. 510, pp. 629–640. Springer, Heidelberg (1991)
19. Johnson, D.S.: Approximation algorithms for combinatorial problems. Journal of Computer and System Sciences 9(3), 256–278 (1974)
20. Khot, S., Ponnuswami, A.K.: Better inapproximability results for maxclique, chromatic number and min-3lin-deletion. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4051, pp. 226–237. Springer, Heidelberg (2006)
21. Milind Dawande, P.K., Tayur, S.: On the biclique problem in bipartite graphs. GSIA Working Paper, Carnegie Mellon University, Pittsburgh (1996)
22. Müller, H.: Alternating cycle-free matchings. Order 7(1), 11–21 (1990)
23. Müller, H.: On edge perfectness and classes of bipartite graphs. Discrete Mathematics 149(1-3), 159–187 (1996)
24. Nau, D.S., Markowsky, G., Woodbury, M.A., Amos, D.B.: A mathematical analysis of human leukocyte antigen serology. Mathematical Biosciences 40(3-4), 243–270 (1978)
25. Nor, I., Hermelin, D., Charlat, S., Engelstadter, J., Reuter, M., Duron, O., Sagot, M.-F.: Mod/Resc parsimony inference: Theory and application. Inf. Comput. 213, 23–32 (2012)
26. Orlin, J.: Contentment in graph theory: Covering graphs with cliques. Indagationes Mathematicae (Proceedings) 80(5), 406–424 (1977)
27. Peeters, R.: The maximum edge biclique problem is NP-complete. Discrete Applied Mathematics 131(3), 651–654 (2003)
28. Simon, H.: On approximate solutions for combinatorial optimization problems. SIAM Journal on Discrete Mathematics 3(2), 294–310 (1990)
29. Zuckerman, D.: Linear degree extractors and the inapproximability of max clique and chromatic number. In: Proceedings of the Thirty-Eighth Annual ACM Symposium on Theory of Computing, STOC 2006, pp. 681–690. ACM, New York (2006)

Succinct Indices for Path Minimum, with Applications to Path Reporting*

Timothy M. Chan[1], Meng He[2], J. Ian Munro[1], and Gelin Zhou[1]

[1] David R. Cheriton School of Computer Science, University of Waterloo, Canada
{tmchan,imunro,g5zhou}@uwaterloo.ca
[2] Faculty of Computer Science, Dalhousie University, Canada
mhe@cs.dal.ca

Abstract. In the path minimum query problem, we preprocess a tree on n weighted nodes, such that given an arbitrary path, we can locate the node with the smallest weight along this path. We design novel succinct indices for this problem; one of our index structures supports queries in $O(\alpha(m, n))$ time, and occupies $O(m)$ bits of space in addition to the space required for the input tree, where m is an integer greater than or equal to n and $\alpha(m, n)$ is the inverse-Ackermann function. These indices give us the first succinct data structures for the path minimum problem, and allow us to obtain new data structures for path reporting queries, which report the nodes along a query path whose weights are within a query range. We achieve three different time/space tradeoffs for path reporting by designing (a) an $O(n)$-word structure with $O(\lg^\epsilon n + occ \cdot \lg^\epsilon n)$ query time, where occ is the number of nodes reported; (b) an $O(n \lg\lg n)$-word structure with $O(\lg\lg n + occ \cdot \lg\lg n)$ query time; and (c) an $O(n \lg^\epsilon n)$-word structure with $O(\lg\lg n + occ)$ query time. These tradeoffs match the state of the art of two-dimensional orthogonal range reporting queries [8] which can be treated as a special case of path reporting queries. When the number of distinct weights is much smaller than n, we further improve both the query time and the space cost of these three results.

1 Introduction

As one of the most fundamental structures in computer science, trees have been widely used in modeling and representing different types of data in numerous computer applications. In many cases, objects are represented by nodes and their properties are characterized by weights assigned to nodes. Researchers have studied the problems of maintaining a weighted tree, such that various types of *path queries* can be computed efficiently [1, 9, 24, 25, 23, 6, 20, 28, 22, 11]. In this paper, we consider *path minimum (maximum)* queries and *path reporting* queries.

– Path minimum(maximum): Given nodes u and v, return the minimum (maximum) node along the path from u to v, i.e., the node along the path whose weight is the minimum (maximum) one;

* This work was supported by NSERC and the Canada Research Chairs Program. Part of the first author's work was done during his visit to the Department of Computer Science and Engineering, Hong Kong University of Science and Technology.

– Path reporting: Given nodes u and v along with a range $[p, q]$, report the nodes along the path from u to v whose weights are between p and q.

When the given tree is a path, the above queries become range minimum (maximum) queries [14,11] and two-dimensional orthogonal range reporting queries [8], respectively. As stated in [20], the path queries we consider generalize these fundamental range queries to weighted trees.

In this paper, we represent the input tree as an ordinal one, i.e., a rooted tree in which siblings are ordered. The weights of nodes are assumed to be drawn from $[1..\sigma]$. We use lg to denote the base-2 logarithm and use ϵ to denote a constant in $(0, 1)$. Unless otherwise specified, the underlying model of computation is the standard word RAM model with word size $w = \Omega(\lg n)$.

Path Minimum. The problem of supporting path minimum queries has been heavily studied [1,9,2,29,23,6,11]. Unlike our formulation, previous work considers trees on weighted edges instead of weighted nodes. However, it is not hard to see that these two formulations are equivalent.

The *minimum spanning tree verification* problem is a special offline case of the path minimum problem, for which one should determine whether a given spanning tree is minimum. This problem can be solved using $O(n + m)$ comparisons and linear overhead under the word RAM model [24]. The online version of this problem, i.e., the path minimum problem, was considered by Alon and Schieber [1]. In the pointer machine model, they designed a structure that uses super-linear space. Concurrently, Chazelle [9] presented a linear space data structure under word RAM. Both structures require $O(\alpha(n))$ query time to sum up weights along a given path, where $\alpha(n)$ is the inverse-Ackermann function, and weights are drawn from a semigroup. Thus the online version of the path minimum problem can also be supported in the same time and space. More recently, several solutions using $O(n)$ words, i.e., $O(n \lg n)$ bits, with $O(1)$ query time have been designed under word RAM [2,23,6,11]. Pettie [29] studied the lower bound in terms of comparisons of edge weights, and showed that $\Omega(q \cdot \alpha(q, n) + n)$ comparisons are necessary to serve q queries over a tree of size n.

In this paper we present lower and upper bounds for path minimum queries. In Lemma 4 we show that $\Omega(n \lg n)$ bits of space are necessary to encode the answers to path minimum queries over a tree of size n. This distinguishes path minimum queries from range minimum queries in terms of space cost.

We adopt the *indexing model* (also called the *systematic model*) [3,7,5] in designing new data structures for path minimum queries. Applying this model to weighted trees, we assume that weights are represented in an arbitrary given raw form; the only requirement is that the given data support access to the weight of a node given its preorder rank. Auxiliary data structures called *indices* are then constructed, and query algorithm uses indices and the access operator provided for the raw data. Not only is this an important theoretical model (its variants are frequently used to prove lower bounds [12,26,17]), it is also of practical importance as it addresses cases in which the (large) raw data are stored in slower external memory or even remotely, while the (smaller) indices could be stored in memory or locally. The space of an index is called *additional space*. Note that the lower bound in the previous paragraph is proved under the encoding model, and thus does not apply to the indexing model.

To present our results, we assume the following definition for the Ackermann function: $A_0(i) = i+1$ and $A_{\ell+1}(i) = A_\ell^{(i+1)}(i+8)$, where $A_\ell^{(0)}(i) = i$ and $A_\ell^{(i)}(j) = A_\ell(A_\ell^{(i-1)}(j))$ for $i \geq 1$. This is faster growing than the one defined by Cormen et al. [10]. Let $\alpha(m,n)$ be the smallest L such that $A_L(\lfloor m/n \rfloor) > n$, and $\alpha(n)$ be $\alpha(n,n)$. The following theorem presents our indices for path minimum:

Theorem 1. *An ordinal tree on n weighted nodes can be indexed (a) using $O(m)$ bits of space to support path minimum queries in $O(\alpha(m,n))$ time and $O(\alpha(m,n))$ accesses to the weights of nodes, for any $m \geq n$; or (b) using $2n+o(n)$ bits of space to support path minimum queries in $O(\alpha(n))$ time and $O(\alpha(n))$ accesses to the weights of nodes.*

To better understand variant (a) of this result, we discuss the time and space costs for the following possible values of m. When $m = n$, then we have an index of $O(n)$ bits that supports path minimum queries in $O(\alpha(n))$ time. When $m = O(n(\lg^*)^*n)$, for example, then it is well-known that $\alpha(m,n) = O(1)$, and thus we have an index of $O(n(\lg^*)^*n)$ [1] bits that supports path minimum queries in $O(1)$ time. Previous solutions [2, 23, 6, 11] to the same problem with constant query time occupy $\Omega(n \lg n)$ bits of space in addition to the space required for the input tree. Combining the above results with a trivial encoding of node weights, we obtain the first succinct data structures for path minimum queries. With a little extra work, we can even represent a weighted tree using $n \lg \sigma + 2n + o(n)$ bits only, i.e., within an $o(n)$ additive term of the information-theoretic lower bound, to support queries in $O(\alpha(n))$ time. Considering the construction time is $O(n)$, this variant almost matches the lower bound of Pettie [29].

Path Reporting. Path reporting queries were proposed by He et al. [20]. They obtained two solutions: one uses $O(n)$ words and $O(\lg \sigma + occ \cdot \lg \sigma)$ query time, and the other uses $O(n \lg \lg \sigma)$ words but $O(\lg \sigma + occ \cdot \lg \lg \sigma)$ query time, where σ is the number of distinct weights and *occ* is output size. For the same problem, Patil et al. [28] designed a succinct structure based on *heavy path decomposition* [31, 18]. Their structure requires only $n \lg \sigma + 6n + o(n \lg \sigma)$ bits but $O(\lg \sigma \lg n + occ \cdot \lg \sigma)$ time. Concurrently, He et al. [22] designed another succinct structure based on a different idea. This structure, requiring $O(\lg \sigma / \lg \lg n + occ \cdot \lg \sigma / \lg \lg n)$ query time, is the best previously known linear space solution.

In this paper, we design three new data structures for path reporting queries:

Theorem 2. *An ordinal tree on n nodes whose weights are drawn from a set of σ distinct weights can be represented using $O(n \lg \sigma \cdot \mathtt{s}(\sigma))$ bits of space, such that path reporting queries can be supported in $O(\min\{\lg \lg \sigma + \mathtt{t}(\sigma), \lg \sigma / \lg \lg n\} + occ \cdot \min\{\mathtt{t}(\sigma), \lg \sigma / \lg \lg n\})$ time, where occ is the size of output, and $\mathtt{s}(\sigma)$ and*

[1] $(\lg^*)^*$ is the number of times $\lg^*$ must be iteratively applied before the result becomes less than or equal to 1.

$\mathtt{t}(\sigma)$ *are: (a)* $\mathtt{s}(\sigma) = O(1)$ *and* $\mathtt{t}(\sigma) = O(\lg^{\epsilon}\sigma)$*; (b)* $\mathtt{s}(\sigma) = O(\lg\lg\sigma)$ *and* $\mathtt{t}(\sigma) = O(\lg\lg\sigma)$*; or (c)* $\mathtt{s}(\sigma) = O(\lg^{\epsilon}\sigma)$ *and* $\mathtt{t}(\sigma) = O(1)$.

These results completely subsume almost all previous results; the only exceptions are the succinct data structures for this problem designed in previous work, whose query times are worse than our linear-space solution. Furthermore, our data structures match the state of the art of 2D range reporting queries [8] when $\sigma = n$, and have better performance when σ is much less than n.

Overview of Techniques. Unlike previous succinct tree structures [16, 28, 22, 19, 13], our approach for path minimum is based on *topological partitions* [15] which transform the input tree into a binary tree and further recursively decomposes it into a hierarchy of clusters with constant external degrees. Our main strategy of constructing path minimum query structures (in Section 3) is to recursively divide the set of levels of decomposition into multiple subsets of levels; with a carefully-defined version of the path minimum query problem which takes levels in the decomposition as parameters, the query over the entire structure can be answered by conquering the subproblems local to the subsets of levels. Solutions to special cases of the query problem are also designed, so that we can present the time and space costs of our solution using recursive formulas. Then, by carefully constructing a number series and using it in the division of levels into subsets, we can prove that our structures achieve the tradeoff presented in Theorem 1 using the inverse-Ackermann function. This approach is novel and exciting, and it does not directly use standard techniques for word RAM at all.

The above strategy would not achieve the desired space bound without a succinct data structure that supports navigations in the input tree, the binary tree that it is transformed into and the clusters in the topological partition. We design such a structure (in Section 4) occupying only $2n + o(n)$ bits, which is of independent interest. Finally, to design solutions to path reporting (in Section 5), we follow the general framework of He et al. [22] to extract subtrees based on the partitions of the entire weight range, and make use of a conceptual structure that borrows ideas from the classical range tree. One strategy of achieving improved results is to further reduce path reporting into queries in which the weight ranges are one-sided, which allows us to apply our succinct index for path minimum queries to achieve the tradeoffs presented in the second half of the abstract. We further apply a tree covering strategy to reduce the space cost for the case in which the number of distinct weights is much smaller than n, and hence prove Theorem 2.

2 Preliminaries

Succinct Data Structures. Bit vectors are one of the main building blocks in many space efficient data structures. Let $B[1..n]$ denote a bit vector of size n. For $\alpha \in \{0, 1\}$, $\mathtt{rank}_{\alpha}(B, i)$ counts α-bits in $B[1..i]$, while $\mathtt{select}_{\alpha}(B, i)$ finds the i-th α-bit in B. The following lemma presents succinct bit vector representations:

Lemma 1 ([30]). *A bit vector with $n-m$ zeros and m ones can be represented using $\lg\binom{n}{m}+O(n\lg\lg n/\lg n)$ bits of space to support* $\mathtt{rank}_\alpha$, $\mathtt{select}_\alpha$, *and the access to each bit in constant time.*

The next lemma presents succinct ordinal trees over an alphabet of size $\sigma = o(\lg\lg n)$, and unlabeled trees can be considered as a special case:

Lemma 2 ([16, 19, 13]). *An ordinal tree T on n nodes over an alphabet of size σ can be encoded in $n(\lg\sigma+2)+O(\sigma n\lg\lg\lg n/\lg\lg n)$ bits of space to support the following operations in $O(1)$ time. Here x and y, which are nodes in T, are identified by preorder ranks. A node is its own 0-th ancestor. In addition, a node with label α is an α-node, and an α-node is an α-ancestor of its descendants.*

- $\mathtt{depth}(T,x)$: *the depth of x (i.e., the number of ancestors of x);*
- $\mathtt{depth}_\alpha(T,x)$: *the number of α-ancestors of x;*
- $\mathtt{parent}(T,x)$: *the parent of x;*
- $\mathtt{level\_anc}(T,x,i)$: *the i-th lowest ancestor of x;*
- $\mathtt{level\_anc}_\alpha(T,x,i)$: *the i-th lowest α-ancestor of x;*
- $\mathtt{LCA}(T,x,y)$: *the lowest common ancestor of x and y.*

Topological Partitions and Topology Trees. Frederickson [15] presented topological partitions for online updating of minimum spanning trees. An input tree T, which may have arbitrary degree, is transformed into a binary tree $\mathcal{B}$; and then $\mathcal{B}$ is partitioned as follows:

Lemma 3 ([15]). *A binary tree $\mathcal{B}$ on n nodes can be partitioned into a hierarchy of clusters with $h+1$ levels for some $h=O(\lg n)$: each cluster is a connected component of $\mathcal{B}$; the only cluster at level h contains all the nodes in $\mathcal{B}$, and each cluster at level 0 contains a single node; a cluster at level $i>0$ is the disjoint union of at most 4 clusters at level $i-1$; at level i, there are at most $(3/4)^i n$ clusters of size at most 4^i, which form a partition of the nodes in the binary tree; and each cluster has at most 3 nodes that are connected to the outside, which are called its endpoints.*

The hierarchy of clusters is referred to as the topology tree of T and $\mathcal{B}$, which is denoted by $\mathcal{H}$. A node at level $i>0$ of $\mathcal{H}$ represents a cluster C at level i of the hierarchy, and its children represent the clusters at the lower level that partition C. In particular, the leaf nodes of $\mathcal{H}$, which are at level 0, represent individual nodes of the binary tree $\mathcal{B}$.

Tree Extraction. He et al. [20, 22] introduced tree extraction to support path queries. This technique is based on the deletion operation of tree edit distance [4]. To delete a non-root node u, its children are inserted in place of u into the list of children of its parent, preserving the original left-to-right order. Let T be an ordinal tree and X be a subset of nodes in T. The X-extraction of T, F_X, is defined to be the ordinal forest obtained by deleting all the nodes that are not in X from T. There is a natural one-to-one correspondence between the nodes in X and the nodes in F_X, and the ancestor-descendant and preorder relationships among the remaining nodes are preserved. If X contains the root of T, then F_X consists of a single ordinal tree only, which is denoted by T_X.

3 Path Minimum Queries

We first give a simple lower bound for path minimum queries under the *encoding model*. Due to page limitation, the proof is omitted here.

Lemma 4. *In the worst case, $\Omega(n \lg n)$ bits are required to encode the answers to all possible path minimum queries over a tree on n weighted nodes.*

Unlike the lower bound of Pettie [29], Lemma 4 provides a separation between path minimum and range minimum in terms of space: $\Omega(n \lg n)$ bits are required to encode path minimum queries over a tree on n weighted nodes, while range minimum over an array of length n can always be encoded in $2n$ bits [14].

Now we consider the support for path minimum queries. The space cost of maintaining a weighted tree is dominated by storing the weights of nodes. Thus we represent the input tree as an ordinal one, for which the nodes are identified by their preorder ranks. This does not significantly affect the space cost.

We will assume the indexing model described in Section 1 and develop several novel succinct indices for path minimum queries. Thus the weights of nodes are assumed to be stored separately from the index for queries, and can be accessed with the preorder ranks of nodes. The time cost to answer a given query is measured by the number of accesses to the index and that to node weights.

Let T be an input tree on n nodes. Here T is represented as an ordinal one, and its nodes are identified by preorder ranks. We transform T into a binary tree $\mathcal{B}$ as follows (essentially as in the usual way but with added dummy nodes): For each node u with $d > 2$ children, where $v_1, v_2, \cdots, v_d$ are children of u, we add $d-2$ dummy nodes $x_1, x_2, \cdots, x_{d-2}$. The first and the second child of u are set to be v_1 and x_1, respectively. For $1 \leq k < d-2$, the first and the second child of x_k are set to be v_{k+1} and x_{k+1}, respectively. Finally, the first and the second child of x_{d-2} are set to be v_{d-1} and v_d, respectively. In this way we have replaced u and its children with a right-leaning binary tree, where the leaf nodes are children of u. This transformation does not change the preorder relationship among the nodes in T. In addition, the set of non-dummy nodes along the path between any two non-dummy nodes remain the same after transformation.

We decompose $\mathcal{B}$ and obtain the topology tree $\mathcal{H}$ using Lemma 3. For simplicity, a cluster at level i is called a *level-i cluster*, and its endpoints are called *level-i endpoints*. Since T and $\mathcal{B}$ are both rooted trees, each cluster contains a node that is the ancestor of all the other nodes in the same cluster. This node is referred to as the *root* of the cluster. In the topology tree $\mathcal{H}$, sibling clusters are ordered by the preorder ranks of their roots. Each cluster C is identified by its *topological rank*, i.e., the preorder rank of the node in $\mathcal{H}$ that represents C.

To facilitate the use of topology trees, we define operations relevant to nodes, clusters and endpoints. Here we assume that x and y are nodes in $\mathcal{B}$.

- conversions between nodes in $\mathcal{B}$ and T;
- `level_cluster`$(\mathcal{H}, i, x)$: the level-i cluster that contains node x;
- `LCC`$(\mathcal{H}, x, y)$: the lowest-level cluster C that contains nodes x and y;
- `cluster_root`$(\mathcal{H}, C)$: the root of level-i cluster C;

– `cluster_endpoints`$(\mathcal{H}, C)$: the endpoints of level-i cluster C;
– `closest_endpoint`$(\mathcal{H}, C, x)$: the endpoint of C that is the closest to node x, given that x is outside of C;
– `parent`$(\mathcal{B}, x)$: the parent node of x;
– `LCA`$(\mathcal{B}, x, y)$: the lowest common ancestor of x and y;
– `endpoint_rank`$(\mathcal{B}, i, x)$: the number of level-i endpoints preceding x in preorder of $\mathcal{B}$;
– `endpoint_select`$(\mathcal{B}, i, j)$: the j-th level-i endpoint in preorder of $\mathcal{B}$.

Lemma 5. *Let T be an ordinal tree on n nodes. Then T, the transformed binary tree $\mathcal{B}$, and their topology tree $\mathcal{H}$ can be encoded in $2n + o(n)$ bits of space such that the operations listed above can be supported in $O(1)$ query time.*

To present our key strategy first, we defer the proof of Lemma 5 to Section 4. As the conversion between nodes in T and $\mathcal{B}$ can be performed in $O(1)$ time, we assume that each given query is specified by two nodes in $\mathcal{B}$. Let h denote the highest level of $\mathcal{H}$. The following two query problems are defined in terms of clusters and endpoints, for $0 \le i < j \le h$:

– $P_{i,j}(C_0, C_1, C_2)$: find the minimum node along the path from an endpoint of a level-i cluster C_1 to an endpoint of another level-i cluster C_2, where both C_1 and C_2 are contained in the same level-j cluster C_0;
– $P'_{i,j}(C_0, C_1)$: find the minimum node along the path from an endpoint of a level-i cluster C_1 to an endpoint of a level-j cluster C_0, where C_1 is in C_0.

For simplicity, we drop the parameters when referring to these problems in the rest of this section. Thus the original problem is $P_{0,h}$. If $P_{i,j}$ is solved, then $P'_{i,j}$ and $P_{i',j}$ for $i' > i$ are also naturally solved.

Let $h_0 > 0$ be a parameter whose value will be determined later. We will solve P_{0,h_0} using brute-force search, and support $P_{h_0,h}$ using a novel recursive approach as described below. For each cluster C whose level is higher than or equal to h_0, we explicitly store the minimum node on the *bridges* of C, where bridges are paths connecting pairs of endpoints of C (excluding the endpoints). This requires $6i$ bits for a cluster at level i, as it has at most three bridges. The overall space cost is $\sum_{i=h_0}^{h} \left(6i \cdot (3/4)^i n\right) = O(h_0 (3/4)^{h_0} n)$ bits, which is $o(n)$ bits when $h_0 = \omega(1)$. We have the following lemmas as base cases of recursion.

Lemma 6. *For $i \ge h_0$, $P_{i,i+8}$ can be solved in $O(1)$ query time and 0 extra bits.*

The correctness of this lemma follows from the fact that each level-$(i+8)$ cluster contains a constant number of level-i clusters, which makes it possible to split a query path into a constant number of subpaths, each being either a level-i endpoint or a bridge of some level-i cluster which have been preprocessed.

Lemma 7. *$P'_{i,j}$ can be solved using $O(1)$ query time and $O((3/4)^i n)$ extra bits.*

Proof. We construct an ordinal tree T_i by extracting all level-i endpoints from $\mathcal{B}$. The size of T_i is $O((3/4)^i n)$. For convenience, we denote a node in T_i by u' iff it corresponds to a level-i endpoint u in $\mathcal{B}$. The conversion between u and u'

can be performed in $O(1)$ time using `endpoint_rank` and `endpoint_select`.

Next we assign labels from alphabet $\{0,1\}$ to the nodes of T_i. We only consider the case in which the level-j endpoint is the first one in preorder of its cluster; the other cases can be handled similarly. Let u be any level-i endpoint and let v be the first endpoint of $C_0 = \texttt{level\_cluster}(\mathcal{B}, j, u)$, i.e., the level-$j$ cluster that contains u. Like the proof of Lemma 6, the path from u to v in $\mathcal{B}$ can be split into a sequence of level-i endpoints and bridges of level-i clusters. Let x be the next level-i endpoint on the path. We assign 1 to u' in T_i if the minimum node between u and v is smaller than that between x and v; otherwise we assign 0 to u'. We represent this labeled tree in $O((3/4)^i n)$ bits using Lemma 2.

To find the minimum node between u and v, we need only find the closest 1-node to u' along the path from u' to v' in T_i. This can be done in $O(1)$ time by performing $\texttt{level\_anc}_\alpha$ and $\texttt{depth}_\alpha$ operations on T_i. Let x' be such a node. Then the minimum node between u and v must be x or appear on some bridge of the level-i cluster that contains x, and thus can be retrieved in $O(1)$ time. □

Now we turn to consider general $P_{i,j}$, for which we will develop a recursive strategy with multiple iterations. At each iteration, we pick a sequence $i = i_0 < i_1 < i_2 < \ldots < i_k = j$, for which $P_{i_0,i_1}, P_{i_1,i_2}, \ldots, P_{i_{k-1},i_k}$ are assumed to be solved at the previous iteration. By Lemma 7, we solve $P'_{i,i_1}, P'_{i,i_2}, \ldots, P'_{i,i_k}$ using $O(k(3/4)^i n)$ bits of additional space.

Consider the support for a query of $P_{i,j}$, for which level-i endpoints u and t are endpoints of the query path, and u and t are contained in the same level-j cluster. W.l.o.g, we assume that t is an ancestor of u (the case in which neither node is an ancestor of the other can be reduced to this case easily). We compute $C_0 = \texttt{LCC}(\mathcal{B}, u, t)$, which is the lowest level cluster that contains both u and t. Let i' be the level of C_0. Then we determine s such that $i_s < i' \leq i_{s+1}$; s can be computed in constant time by precomputing the result for each of the $h + 1 = O(\lg n)$ levels. Let C_1 be the level-i_s cluster that contains u and let x be an endpoint of C_1 that is between u and t. Similarly, let C_2 be the level-i_s cluster that contains t and let z be an endpoint of C_2 that is between u and t. Note that x and z can be found in constant time using `level_cluster` and `closest_endpoint`. Thus the query path can be decomposed into $u \sim x \sim z \sim t$. The minimum node on $u \sim x$ and that on $z \sim t$ can be found by querying P'_{i,i_s}. The minimum node on $x \sim z$ can be found by recursively querying $P_{i_s,i_{s+1}}$.

Summarizing the discussion above, we have the following recurrences. Here ℓ is the number of iterations, and $Q_\ell(i,j)$ and $S_\ell(i,j)$ are time and space costs for solving $P_{i,j}$ at the ℓ-th iteration.

$$S_{\ell+1}(i,j) = \sum_{s=0}^{k-1} S_\ell(i_s, i_{s+1}) + O(k(3/4)^i n) \tag{1}$$

$$Q_{\ell+1}(i,j) = \max_{s=0}^{k-1} Q_\ell(i_s, i_{s+1}) + O(1) \tag{2}$$

Lemma 8. *Given a fixed value L, there exists a recursive strategy and some constant c such that, for $0 \leq \ell \leq L$, $S_\ell(i, A_\ell(i)) \leq c(4/5)^i n$ and $Q_\ell(i, A_\ell(i)) \leq c\ell$.*

Proof. At the 0-th iteration, we have $A_0(i) = i + 1$. This can be used as the base case. By Lemma 6, $P_{i,i+1}$ can be supported using constant query time at no extra space cost. Thus the statement holds for $\ell = 0$.

At the $(\ell + 1)$-th iteration, we choose the sequence $i, i + 8, A_\ell(i + 8), A_\ell^{(2)}(i + 8), \ldots, A_\ell^{(i)}(i + 8), A_\ell^{(i+1)}(i + 8)$. The last term is $A_{\ell+1}(i)$. Then by Equation 1:

$$\begin{aligned} S_{\ell+1}(i, A_{\ell+1}(i)) &\le \sum_{0\le j\le i} S_\ell(A_\ell^{(j)}(i+8), A_\ell^{(j+1)}(i+8)) + O(i(3/4)^i n) \\ &\le O(i(3/4)^i n) + \sum_{0\le j\le i} c(4/5)^{A_\ell^{(j)}(i+8)} n \\ &\le O(i(3/4)^i n) + 5c(4/5)^{i+8} n \ \le \ c(4/5)^i n \end{aligned}$$

for some sufficiently large constant c. This convergence follows because $5(4/5)^8$ is less than 1. Equation 2 implies $Q_{\ell+1}(i, A_{\ell+1}(i)) \le O(1) + \max_{0\le j\le i} Q_\ell(A_\ell^{(j)}(i + 8), A_\ell^{(j+1)}(i+8)) \le O(1)+c\ell \le c(\ell+1)$ for some sufficiently large c. The induction thus carries through. □

To achieve desired time-space tradeoffs, we recurse one more iteration. Given a parameter $m \ge n$, we set $L = \alpha(m, n)$ and $h_0 = 0$. At the final $(L + 1)$-th iteration, choose the sequence $0, 1, 2, \ldots, \lfloor m/n \rfloor, A_L(\lfloor m/n \rfloor)$. This gives

$$S_{L+1}(0, A_L(\lfloor m/n\rfloor)) \le S_L(\lfloor m/n\rfloor, A_L(\lfloor m/n\rfloor)) + O(\lfloor m/n\rfloor n) \ \le \ O(m)$$

and $Q_{L+1}(0, A_L(\lfloor m/n\rfloor)) \le O(L) = O(\alpha(m, n))$. Thus we have proved part (a) of Theorem 1.

To further decrease the space cost of our index to $2n + o(n)$ bits, we still recurse L levels. We choose $L = \alpha(n)$ and $h_0 = \lceil \log_4 L\rceil$. Note that $h_0 = \omega(1)$ and $A_L(h_0) \ge h$. Therefore we have $S_L(h_0, A_L(h_0)) = O((4/5)^{h_0} n) = o(n)$, and $Q_L(h_0, A_L(h_0)) = O(L) = O(\alpha(n))$. To solve P_{0,h_0} and P'_{0,h_0}, we perform brute-force search on the query path in $O(4^{h_0}) = O(\alpha(n))$ time, as a level-h_0 cluster has at most 4^{h_0} nodes. Thus we have proved part (b) of Theorem 1.

By further constructing the preorder label sequence [22, 21] of T, we have:

Corollary 1. *Let T be an ordinal tree on n nodes, each having a weight drawn from $[1..\sigma]$. Then T can be represented (a) using $n \lg \sigma + O(m)$ bits of space to support path minimum queries in $O(\alpha(m, n))$ time, for any $m \ge n$; or (b) using $n(\lg \sigma + 2) + o(n)$ bits of space to support path minimum queries in $O(\alpha(n))$ time.*

4 Encoding Topology Trees

Let T be an ordinal tree on n nodes. As described in Section 3, we transform T into a binary tree $\mathcal{B}$, and compute the topology tree of $\mathcal{B}$ as $\mathcal{H}$. Let $n_\mathcal{H}$ denote the number of nodes in $\mathcal{H}$; clearly $n_\mathcal{H} = O(n)$. Let $i_1 = \lceil 8 \lg\lg n\rceil$ and $i_2 = \lfloor (1/2) \lg\lg n\rfloor - 1$. By Lemma 3, there are at most $n_1 = (3/4)^{i_1} n = O(n/(4/3)^{8 \lg\lg n}) = O(n/ \lg^{8 \lg(4/3)} n) < O(n/ \lg^3 n)$ clusters at level i_1, each being of size at most $m_1 = 4^{i_1} \leq 4^{8 \lg \lg n + 1} = 4 \lg^{16} n$. Similarly, there are at most $n_2 = (3/4)^{i_2} n = O(n/\lg^{(1/2)\lg(4/3)} n) < O(n/(\lg^{1/5} n))$ clusters at level i_2, each being size of at most $m_2 = 4^{i_2} \leq 4^{(1/2)\lg \lg n - 1} = (\lg n)/4$. Level-$i_1$ clusters are called *mini-clusters*, and level-i_2 ones are called *micro-clusters*. We first store the encodings of micro-clusters.

Lemma 9. *All micro-clusters can be encoded in* $2n + o(n)$ *bits of space such that given the topological rank of a cluster, its encoding can be retrieved in* $O(1)$ *time if it is a micro-cluster.*

Proof. Note that $\mathcal{B}$ has at most $2n$ nodes. Given a micro-cluster C, we do not store its encoding directly because it could require about $4n$ bits of space for all micro-clusters. Instead, we define X to be the union of non-dummy nodes and endpoints of C and store only C_X, where C_X is the X-extraction of C as defined in Section 2. We also mark the (at most 3) dummy nodes in C_X, which requires $O(\lg m_2) = O(\lg \lg n)$ bits per node. Encoding C_X as *balanced parentheses* [27], the overall space cost of encoding C is $2n_C + O(\lg \lg n)$ bits, where n_C is the number of non-dummy nodes in C. We concatenate the above encodings of all micro-clusters ordered by topological rank and store them in a sequence, P, of $n' = 2n + O(n \lg \lg n/(\lg^{1/5} n))$ bits. We construct a sparse bit vector, P', of the same length, and set $P'[i]$ to 1 iff $P[i]$ is the first bit of the encoding of a micro-cluster. P' can be represented using Lemma 1 in $\lg \binom{n'}{n_2} + O(n \lg \lg n/\lg n) = O(n \lg \lg n/(\lg^{1/5} n))$ bits to support $\texttt{rank}_\alpha$ and $\texttt{select}_\alpha$ in constant time. We construct another bit vector $B_0[1..n_{\mathcal{H}}]$, in which $B_0[j] = 1$ iff the cluster with topological rank j is a micro-cluster, which has the same asymptotic space cost.

To retrieve the encoding of a cluster, C, whose topological rank is j, we first use B_0 to check if C is a micro-cluster. If it is, let $r = \texttt{rank}_1(B_0, j)$. Then the encoding of C_X is $P[\texttt{select}_1(P', r)..\texttt{select}_1(P', r+1) - 1]$. To recover C from C_X, we need only follow the procedure described at the beginning of Section 3. This can be done in $O(1)$ time using a lookup table F_0 of $o(n)$ bits. $\square$

To support operations, our main strategy is to encode global information at levels on or above i_1, information local to a mini-cluster among levels between i_2 and i_1, and lookup tables for the levels contained in a micro-cluster. Extra care is needed as the topological order of clusters is not the same as the relative order of their roots in preorder. Details are omitted due to page limitation.

5 A Sketch of Supporting Path Reporting Queries

In this section we sketch our improved data structures for path reporting queries. The details of supporting path reporting queries are deferred to the full version of this paper. Our solutions present various algorithmic techniques we developed to prove Theorem 2, as well as a simplified approach to achieve similar time-space tradeoffs for the 2D orthogonal range reporting problem on an $n \times \sigma$ grid.

Following the approach of He et al. [22], we build a conceptual range tree on $[1..\sigma]$ with branching factor $f = \lceil \lg^{\epsilon} n \rceil$. We keep splitting ranges until the bottom level contains σ leaf ranges. This conceptual range tree has $h = \lceil \log_f \sigma \rceil + 1$ levels, among which the top level is the first level, and the bottom level is the h-th level. For each range $[a..b]$ in the conceptual range tree, we obtain $F_{a,b}$ by extracting all nodes whose weights are in $[a..b]$ from T. Each node x in $F_{a,b}$ is assigned a label between 1 and f, which indicates the child range at the lower level that contains the node that corresponds to x. All these $F_{a,b}$'s are maintained in succinct representations such that path reporting queries with respect to labels can be answered in constant time per node.

Let u and v be the endpoints of query path and $[p..q]$ be the query range. The algorithm of He et al. [22] traverses the conceptual range tree from top to bottom, and splits $[p..q]$ into $O(\lg \sigma / \lg \lg n)$ canonical ranges, each being a single range, or the union of consecutive sibling ranges in the conceptual range tree. Thus the original query can be transformed into $O(\lg \sigma / \lg \lg n)$ subqueries on different $F_{a,b}$'s. This relatively simple algorithm requires an overhead of $O(\lg \sigma / \lg \lg n)$ time, in addition to $O(\lg \sigma / \lg \lg n)$ time per node in the output, since we need convert a node in $F_{a,b}$ to that of T level by level.

Our new algorithm avoids traversing the conceptual range tree level by level. We make use of the *ball-inheritance* problem, such that, given a node x in some $F_{a,b}$, we can determine the node in T that corresponds to x much faster than $O(\lg \sigma / \lg \lg n)$ time. To support the given query, we first find the lowest range $[a..b]$ in the conceptual range tree that completely contains $[p..q]$. Let $[a_1..b_1], [a_2..b_2], \ldots, [a_f..b_f]$ be the child ranges of $[a..b]$ in increasing order of left endpoints. We determine $\alpha, \beta \in [1..f]$, such that $[a_\alpha..b_\beta]$ covers $[p..q]$, and $\beta - \alpha$ is minimized. The query range can thus be decomposed into $[p..b_\alpha]$, $[a_{\alpha+1}..b_{\beta-1}]$ and $[a_\beta..q]$. The support for the second subrange is the same as that of He et al. [22]. For the third subrange, we employ the succinct index designed in Theorem 1 with $m = O(n \lg^* n)$, such that we can enumerate the nodes in $F_{a_\beta,q}$ that are on the query path in increasing order of weights. For each of them, we use the auxiliary data structures for the ball-inheritance problem to find the corresponding node in T. This procedure is terminated if the weight of the current node is above q. The support for the first subrange is similar.

References

1. Alon, N., Schieber, B.: Optimal preprocessing for answering on-line product queries. Tech. rep., Tel Aviv University (1987)
2. Alstrup, S., Holm, J.: Improved algorithms for finding level ancestors in dynamic trees. In: Welzl, E., Montanari, U., Rolim, J.D.P. (eds.) ICALP 2000. LNCS, vol. 1853, pp. 73–84. Springer, Heidelberg (2000)
3. Barbay, J., He, M., Munro, J.I., Satti, S.R.: Succinct indexes for strings, binary relations and multilabeled trees. ACM Transactions on Algorithms 7(4), 52 (2011)
4. Bille, P.: A survey on tree edit distance and related problems. Theor. Comput. Sci. 337(1-3), 217–239 (2005)
5. Bringmann, K., Larsen, K.G.: Succinct sampling from discrete distributions. In: STOC, pp. 775–782 (2013)

6. Brodal, G.S., Davoodi, P., Srinivasa Rao, S.: Path minima queries in dynamic weighted trees. In: Dehne, F., Iacono, J., Sack, J.-R. (eds.) WADS 2011. LNCS, vol. 6844, pp. 290–301. Springer, Heidelberg (2011)
7. Brodal, G.S., Davoodi, P., Rao, S.S.: On space efficient two dimensional range minimum data structures. Algorithmica 63(4), 815–830 (2012)
8. Chan, T.M., Larsen, K.G., Pătraşcu, M.: Orthogonal range searching on the RAM, revisited. In: Symposium on Computational Geometry, pp. 1–10 (2011)
9. Chazelle, B.: Computing on a free tree via complexity-preserving mappings. Algorithmica 2, 337–361 (1987)
10. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press (2009)
11. Demaine, E.D., Landau, G.M., Weimann, O.: On cartesian trees and range minimum queries. Algorithmica 68(3), 610–625 (2014)
12. Demaine, E.D., López-Ortiz, A.: A linear lower bound on index size for text retrieval. J. Algorithms 48(1), 2–15 (2003)
13. Farzan, A., Munro, J.I.: A uniform paradigm to succinctly encode various families of trees. Algorithmica 68(1), 16–40 (2014)
14. Fischer, J., Heun, V.: Space-efficient preprocessing schemes for range minimum queries on static arrays. SIAM J. Comput. 40(2), 465–492 (2011)
15. Frederickson, G.N.: Data structures for on-line updating of minimum spanning trees, with applications. SIAM J. Comput. 14(4), 781–798 (1985)
16. Geary, R.F., Raman, R., Raman, V.: Succinct ordinal trees with level-ancestor queries. ACM Transactions on Algorithms 2(4), 510–534 (2006)
17. Golynski, A.: Optimal lower bounds for rank and select indexes. Theor. Comput. Sci. 387(3), 348–359 (2007)
18. Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM J. Comput. 13(2), 338–355 (1984)
19. He, M., Munro, J.I., Satti, S.R.: Succinct ordinal trees based on tree covering. ACM Transactions on Algorithms 8(4), 42 (2012)
20. He, M., Munro, J.I., Zhou, G.: Path queries in weighted trees. In: Asano, T., Nakano, S.-I., Okamoto, Y., Watanabe, O. (eds.) ISAAC 2011. LNCS, vol. 7074, pp. 140–149. Springer, Heidelberg (2011)
21. He, M., Munro, J.I., Zhou, G.: A framework for succinct labeled ordinal trees over large alphabets. In: Chao, K.-M., Hsu, T.-S., Lee, D.-T. (eds.) ISAAC 2012. LNCS, vol. 7676, pp. 537–547. Springer, Heidelberg (2012)
22. He, M., Munro, J.I., Zhou, G.: Succinct data structures for path queries. In: Epstein, L., Ferragina, P. (eds.) ESA 2012. LNCS, vol. 7501, pp. 575–586. Springer, Heidelberg (2012)
23. Kaplan, H., Shafrir, N.: Path minima in incremental unrooted trees. In: Halperin, D., Mehlhorn, K. (eds.) ESA 2008. LNCS, vol. 5193, pp. 565–576. Springer, Heidelberg (2008)
24. King, V.: A simpler minimum spanning tree verification algorithm. Algorithmica 18(2), 263–270 (1997)
25. Krizanc, D., Morin, P., Smid, M.H.M.: Range mode and range median queries on lists and trees. Nord. J. Comput. 12(1), 1–17 (2005)
26. Miltersen, P.B.: Lower bounds on the size of selection and rank indexes. In: SODA, pp. 11–12 (2005)

27. Munro, J.I., Raman, V.: Succinct representation of balanced parentheses and static trees. SIAM J. Comput. 31(3), 762–776 (2001)
28. Patil, M., Shah, R., Thankachan, S.V.: Succinct representations of weighted trees supporting path queries. J. Discrete Algorithms 17, 103–108 (2012)
29. Pettie, S.: An inverse-ackermann type lower bound for online minimum spanning tree verification. Combinatorica 26(2), 207–230 (2006)
30. Raman, R., Raman, V., Satti, S.R.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Transactions on Algorithms 3(4) (2007)
31. Sleator, D.D., Tarjan, R.E.: A data structure for dynamic trees. J. Comput. Syst. Sci. 26(3), 362–391 (1983)

Online Bipartite Matching with Decomposable Weights

Moses Charikar[1], Monika Henzinger[2], and Huy L. Nguyễn[1]

[1] Princeton University, USA
{moses,hlnguyen}@cs.princeton.edu
[2] University of Vienna, Austria
monika.henzinger@univie.ac.at

Abstract. We study a weighted online bipartite matching problem: $G(V_1, V_2, E)$ is a weighted bipartite graph where V_1 is known beforehand and the vertices of V_2 arrive online. The goal is to match vertices of V_2 as they arrive to vertices in V_1, so as to maximize the sum of weights of edges in the matching. If assignments to V_1 cannot be changed, no bounded competitive ratio is achievable. We study the weighted online matching problem with *free disposal*, where vertices in V_1 can be assigned multiple times, but only get credit for the maximum weight edge assigned to them over the course of the algorithm. For this problem, the greedy algorithm is 0.5-competitive and determining whether a better competitive ratio is achievable is a well known open problem.

We identify an interesting special case where the edge weights are decomposable as the product of two factors, one corresponding to each end point of the edge. This is analogous to the well studied related machines model in the scheduling literature, although the objective functions are different. For this case of decomposable edge weights, we design a 0.5664 competitive randomized algorithm in complete bipartite graphs. We show that such instances with decomposable weights are non-trivial by establishing upper bounds of 0.618 for deterministic and 0.8 for randomized algorithms.

A tight competitive ratio of $1 - 1/e \approx 0.632$ was known previously for both the 0-1 case as well as the case where edge weights depend on the offline vertices only, but for these cases, reassignments cannot change the quality of the solution. Beating 0.5 for weighted matching where reassignments are necessary has been a significant challenge. We thus give the first online algorithm with competitive ratio strictly better than 0.5 for a non-trivial case of weighted matching with free disposal.

1 Introduction

In recent years, online bipartite matching problems have been intensely studied. Matching itself is a fundamental optimization problem with several applications, such as matching medical students to residency programs, matching men and women, matching packets to outgoing links in a router and so on. There is a rich body of work on matching problems, yet there are basic problems we don't understand and we study one such question in this work. The study of the online setting goes back to the seminal work of Karp, Vazirani and Vazirani [26] who gave an optimal $1 - 1/e$ competitive algorithm for the unweighted case. Here $G(V_1, V_2, E)$ is a bipartite graph where V_1 is known beforehand and the vertices of V_2 arrive online. The goal of the algorithm is to match vertices of V_2 as they arrive to vertices in V_1, so as to maximize the size of the matching.

In the weighted case, edges have weights and the goal is to maximize the sum of weights of edges in the matching. In the application of assigning ad impressions to advertisers in display advertisement, the weights could represent the (expected) value of an ad impression to an advertiser and the objective function for the maximum matching problem encodes the goal of assigning ad impressions to advertisers to as to maximize total value. If assignments to V_1 cannot be changed and if edge weights depend on the *online* node to which they are adjacent, it is easy to see that no competitive ratio bounded away from 0 is achievable.

Feldman et al [18] introduced the *free disposal* setting for weighted matching, where vertices in V_1 can be assigned multiple times, but only get credit for the maximum weight edge assigned to them over the course of the algorithm. (On the other hand, a vertex in V_2 can only be assigned at the time that it arrives with no later reassignments permitted). [18] argues that this is a realistic model for assigning ad impressions to advertisers. The greedy algorithm is 0.5 competitive for the online weighted matching problem with free disposal. They study the weighted matching problem with capacities – here each vertex $v \in V_1$ is associated with a capacity $n(v)$ and gets credit for the largest $n(v)$ edge weights from vertices in V_2 assigned to v. They designed an algorithm with competitive ratio approaching $1 - 1/e$ as the capacities approach infinity. Specifically, if all capacities are at least k, their algorithm gets competitive ratio $1 - 1/e_k$ where $e_k = (1 + 1/k)^k$. If all capacities are 1, their algorithm is 1/2-competitive.

Aggarwal et al [1] considered the online weighted bipartite matching problem where edge weights are only dependent on the end point in V_1, i.e. each vertex $v \in V_1$ has a weight $w(v)$ and the weight of all edges incident on v is $w(v)$. This is called the *vertex weighted setting.* They designed a $1 - 1/e$ competitive algorithm. Their algorithm can be viewed as a generalization of the Ranking algorithm of [26].

It is remarkable that some basic questions about a fundamental problem such as matching are still open in the online setting. Our work is motivated by the following tantalizing open problem; *Is it possible to achieve a competitive ratio better than* 0.5 *for weighted online matching ?* Currently no upper bound better than $1 - 1/e$ is known for the setting of general weights – in fact this bound holds even for the setting of 0-1 weights. On the other hand, no algorithm with competitive ratio better than 0.5 (achieved by the greedy algorithm) is known for this problem. By the results of [18], the case where the capacities are all 1 seems to be the hardest case and this is what we focus on.

1.1 Our Results

We identify an interesting special case of this problem where we have a complete graph between V_1 and V_2 and the edge weights are decomposable as the product of two factors, one corresponding to each end point of the edge. This is analogous to the well studied related machines model in the scheduling literature [5,6,9,17] where the load of a job of size p on a machine of speed s is p/s although the objective functions are different. Scheduling problems typically involving minimizing the maximum machine load (makespan) or minimizing the ℓ_p norm of machine loads, where the load on a machine is the sum of loads of all jobs placed on the machine. By contrast, in the problem we study, the objective (phrased in machine scheduling terminology) is to maximize the sum of machine loads where the load of a machine is the load of the largest job placed on the machine. For this case of decomposable edge weights, we design a 0.5664 competitive algorithm (Section 3). For display advertisement using a complete graph models the setting of a specific market segment (such as impressions for males between 20 and 30), where every advertiser is interested in every impression. The weight factor of the offline node u can model the value that a click has for advertiser u, the weight factor of the online node v can model the clickthrough probability of the user to which impression v is shown. Thus, the maximum weight matching in the setting we study corresponds to maximizing the sum of the expected values of all advertisers.

Our algorithm uses a now standard *randomized doubling technique* [8,20,12,23]; however the analysis is novel and non-trivial. We perform a recursive analysis where each step proceeds as follows: We lower bound the profit that the algorithm derives from the fastest machine (i.e. the load of the largest job placed on it) relative to the difference between two optimum solutions - one corresponding to the original instance and the other corresponding to a modified instance obtained by removing this machine and all the jobs assigned to it. This is somewhat reminiscent of, but different from the local ratio technique used to design approximation algorithms. Finally, to exploit the randomness used by the algorithm we need to establish several structural properties of the worst case sequence of jobs – this is a departure from previous applications of this *randomized doubling technique*. While all previous online matching algorithms were analyzed using a *local*, step-by-step analysis, we use a *global* technique, i.e. we reason about the entire sequence of jobs at once. This might be useful for solving the case of online weighted matching for *general* weights. The algorithm and analysis is presented in Section 3 and an outline of the analysis is presented in Section 3.1.

A priori, it may seem that the setting of decomposable weights ought to be a much easier case of weighted online matching since it does not capture the well studied setting of 0-1 weights. We show that such instances with decomposable weights are non-trivial by establishing an upper bound of $(\sqrt{5}-1)/2 \approx 0.618$ on the competitive ratios of deterministic algorithms (Section 4) and an upper bound of 0.8 on the competitive ratio of randomized algorithms (Section 5). The deterministic upper bound constructs a sequence of jobs that is the solution to a certain recurrence relation. Crucial to the success of this approach is a delicate choice of parameters to ensure that the solution of the recurrence is oscillatory (i.e. the roots are complex). In contrast to the setting with capacities, for which a deterministic algorithm with competitive ratio approaching $1 - 1/e \approx 0.632$ exists [18], our upper bound of $(\sqrt{5}-1)/2 < 1-1/e$ for deterministic algorithms shows that no such competitive ratio can be achieved for the decomposable case with unit capacities. Note that the upper bound of $1 - 1/e$ for the unweighted case [26] is for randomized algorithms and does not apply to the setting of decomposable weights that we study here.

In contrast to the vertex weighted setting (and the special case of 0-1 weights) where reassignments to vertices in V_1 cannot improve the quality of the solution, any algorithm for the decomposable weight setting must necessarily exploit reassignments in order to achieve a competitive ratio bounded away from 0. For this class of instances, we give an upper bound approaching 0.5 for the competitive ratio of the greedy algorithm. This shows that for decomposable weights greedy's performance cannot be better than for general weights, where it is 0.5-competitive (Section 2).

1.2 Related Work

Goel and Mehta [21] and Birnbaum and Mathieu [10] simplified the analysis of the Ranking algorithm considerably. Devanur et al [15] recently gave an elegant randomized primal-dual interpretation of [26]; their framework also applies to the generalization to the vertex weighted setting by [1]. Haeupler et al [24] studied online weighted matching in the stochastic setting where vertices from V_2 are drawn from a known distribution. The stochastic setting had been previously studied in the context of unweighted bipartite matching in a sequence of papers [19,28]. Recent work has also studied the random arrival model (for unweighted matching) where the order of arrival of vertices in V_2 is assumed to be a random permutation: In this setting, Karande, at al [25] and Mahdian and Yan [27] showed that the Ranking algorithm of [26] achieves a competitive ratio better than $1-1/e$. A couple of recent papers analyze the performance of a randomized greedy algorithm and an analog of the Ranking algorithm for matching in general graphs [32,22]. Another recent paper introduces a stochastic model for online matching where the goal is to maximize the number of successful assignments (where success is governed by a stochastic process) [30].

A related model allowing cancellation of previously accepted online nodes was studied in [13,7,4] and optimal deterministic and randomized algorithms were given. In their setting the weight of an edge depends *only* on the online node. Additionally in their model they decide in an online fashion only which online nodes to accept, *not* how to match these nodes to offline nodes. If a previously accepted node is later rejected, a non-negative cost is incurred. Since the actual matching is only determined after all online nodes have been seen, their model is very different from ours: Even if the cost of rejection of a previously accepted node is set to 0, the key difference is that they do not commit to a matching at every step and the intended matching can change dramatically from step to step. Thus, it does *not* solve the problem that we are studying.

A related problem that has been studied is online matching with preemption [29,3,16]. Here, the edges of a graph arrive online and the algorithm is required to maintain a subset of edges that form a matching. Previously selected edges can be rejected (preempted) in favor of newly arrived edges. This problem differs from the problem we study in two ways: (1) the graph is not necessarily bipartite, and (2) edges arrive one by one. In our (classic) case, vertices arrives online and all incident edges to a newly arrived vertex v are revealed when v arrives.

Another generalization of online bipartite matching is the Adwords problem [31,14]. In addition, several online packing problems have been studied with applications to the Adwords and Display Advertisement problem [11,21,2].

1.3 Notation and Preliminaries

We consider the following variant of the online bipartite matching problem. The input is a complete bipartite graph $G = (V_1 \cup V_2, V_1 \times V_2)$ along with two weight functions $s : V_1 \to \mathbb{R}_+$ and $w : V_2 \to \mathbb{R}_+$. The weight of each edge $e = (u, v)$ is the product $s(u) \cdot w(v)$. At the beginning, only s is given to the algorithm. Then, the vertices of V_2 arrive one by one. When a new vertex v arrives, $w(v)$ is revealed and the algorithm has to match it to a vertex in V_1. At the end, the reward of each vertex $u \in V_1$ is the maximum weight assigned to u times $s(u)$. The goal of the algorithm is to maximize the sum of the rewards. To simplify the presentation we will call vertices of V_1 *machines* and vertices of V_2 *jobs*. The s-value of a machine u will be called the *speed* of the machines and the w-value of a job v is called the *size* of the job. Thus, the goal of the online algorithm is to assign jobs to machines. However, we are not studying the "classic" variant of the problem since we are using a different optimization function, motivated by display advertisements.

2 Upper Bound for the Greedy Algorithm

We begin by addressing an obvious question, which is how well a greedy approach would solve our problem, and using the proof to provide some intuition for our algorithm in the next section. We analyze here the following simple greedy algorithm: When a job v arrives, the algorithm computes for every machine u the difference between the weight of (u, v) and the weight (u, v'), where v' is the job currently assigned to u. If this difference is positive for at least one machine, the job is assigned to a machine with maximum difference.

Theorem 1. *The competitive ratio of the greedy algorithm is at most* $\frac{1}{2-\epsilon}$ *for any* $\epsilon > 0$.

Proof. Consider the following instance. V_1 consists of a vertex a with $s(a) = 1$ and $t = 1/\epsilon^2$ vertices $b_1, \ldots, b_t$ with $s(b_i) = \epsilon/2\ \forall i$. V_2 consists of the following vertices arriving in the same order $d_1, \ldots, d_{1+t}$ where $w(d_i) = (1-\epsilon/2)^{-i}$. We will prove by induction that all vertices d_i are assigned to a. When d_1 arrives, nothing is assigned so it is assigned to a. Assume that all the first t vertices are assigned to a when d_{t+1} arrives. The gain by assigning d_{i+1} to a is $(w(d_{i+1})-w(d_i))s(a) = \epsilon(1-\epsilon/2)^{-i-1}/2$. The gain by assigning d_{i+1} to some b_j is $w(d_{i+1})s(b_j) = \epsilon(1-\epsilon/2)^{-i-1}/2$. Thus, the algorithm can assign d_{i+1} to a. The total reward of the algorithm is $(1-\epsilon/2)^{-1-t}$. The optimal solution is to assign d_{1+t} to a and the rest to b_i's, getting $(1-\epsilon/2)^{-1-t} + (1-\epsilon/2)^{-t} - 1 \geq (2-\epsilon)(1-\epsilon/2)^{-1-t}$. Thus, the competitive ratio is at most $\frac{1}{2-\epsilon}$. □

The instance used in the proof above suggests some of the complications an algorithm has to deal with in the setting of decomposable weights: in order to have competitive ratio bounded away from 0.5, an online algorithm must necessarily place some jobs on the slow machines. In fact it is possible to design an algorithm with competitive ratio bounded away from 0.5 for the specific set of machines used in this proof (for any sequence of jobs). The idea is to ensure that a job is placed on the fast machine only if its size is larger than $(1+\gamma)$ times the size of the largest job currently on the fast machine (for an appropriately chosen parameter γ). Such a strategy works for any set of machines consisting of one fast machine and several slow machines of the same speed. However, we do not know how to generalize this approach to an arbitrary set of machines. Still, this strategy (i.e. ensuring that jobs placed on a machine increase in size geometrically) was one of the motivations behind the design of the randomized online algorithm to be presented next.

3 Randomized Algorithm

We now describe our randomized algorithm which uses a parameter c we will specify later: The algorithm picks values $x_i \in (0,1]$ uniformly and at random, independently for each machine i. Each job of weight w considered by machine i is placed in the unique interval $w \in (c^{k+x_i}, c^{k+1+x_i}]$ where k ranges over all integers. When a new job w arrives, the algorithm checks the machines in the order of decreasing speed (with ties broken in an arbitrary but fixed way). For machine i it first determines the unique interval into which w falls, which depends on its choice of x_i. If the machine currently does not have a job in this or a bigger interval (with larger k), w is assigned to i and the algorithm stops, otherwise the algorithm checks the next machine.

The following function arises in our analysis:

Definition 1. *Define* $h(c) = 1 - \dfrac{1}{\beta} W\left(\dfrac{\beta e^{\beta}}{c}\right)$ *where* $\beta = \frac{c\ln(c)}{c-1} - 1$ *and* $W()$ *is the Lambert W function (i.e. inverse of* $f(x) = xe^x$*).*

We will prove the following theorem:

Theorem 2. *For* $c \geq e$*, the randomized algorithm has competitive ratio* $\min\left(\frac{c-1}{c\ln(c)}, h(c)\right)$*. In particular, for* $c = 3.55829$*, the randomized algorithm has a competitive ratio* 0.5664*.*

3.1 Analysis Outline

We briefly outline the analysis strategy before describing the details. An instance of the problem consists of a set of jobs and a set of machines. The (offline) optimal solution to an instance is obtained by ordering machines from fastest to slowest, ordering jobs from largest to smallest and assigning the ith largest job to the ith fastest machine. Say the machines are numbered $1, 2, \ldots n$, from fastest to slowest. Let OPT_i denote the value of the optimal solution for the instance seen by the machines from i onwards, i.e. the instance consisting of machines $i, i+1, \ldots n$, and the set of jobs passed by the $(i-1)$st machine to the ith machine in the online algorithm. Then $OPT_1 = OPT$, the value of the optimal solution for the original instance. Even though we defined OPT_i to be the value of the optimal solution, we will sometimes use OPT_i to denote the optimal assignment, although the meaning will be clear from context. Define OPT_{n+1} to be 0. For $2 \leq i \leq n$, OPT_i is a random variable that depends on the random values $x_{i'}$ picked by the algorithm for $i' < i$. In the analysis, we will define random variables Δ_i such that $\Delta_i \geq OPT_i - OPT_{i+1}$ (see Lemma 1 later). Let A_i denote the profit of the online algorithm derived from machine i (i.e. the size of the largest job assigned to machine i times the speed of machine i). Let $A = \sum_{i=1}^{n} A_i$ be the value of the solution produced by the online algorithm. We will prove that for $1 \leq i \leq n$,

$$\mathbb{E}[A_i] \geq \alpha\, \mathbb{E}[\Delta_i] \geq \alpha(\mathbb{E}[OPT_i] - \mathbb{E}[OPT_{i+1}]) \tag{1}$$

for a suitable choice of $\alpha > 0.5$. The expectations in (1) are taken over the random choices of machine $1, \ldots i$. Note that $OPT_i - OPT_{i+1}$ is a random variable, but the sum of these quantities for $1 \leq i \leq n$ is $OPT_1 - OPT_{n+1} = OPT$, a deterministic quantity. Summing up (1) over $i = 1, \ldots n$, we get $\mathbb{E}[A] \geq \alpha \cdot OPT$, proving that the algorithm gives an α approximation.

Inequality (1) applies to a recursive application of the algorithm to the subinstance consisting of machines $i, \ldots n$ and the jobs passed from machine $i-1$ to machine i. The subinstance is a function of the random choices made by the first $i-1$ machines. We will prove that for any instance of the random choices made by the first $i-1$ machines,

$$\mathbb{E}[A_i] \geq \alpha\, \mathbb{E}[\Delta_i]. \tag{2}$$

Here, the expectation is taken over the random choice of machine i. (2) immediately implies (1) by taking expectation over the random choices made by the first $i-1$ machines.

We need to establish (2). In fact, it suffices to do this for $i = 1$ and the proof applies to all values of i since (2) is a statement about a recursive application of the algorithm.

Wlog, we normalize so that the fastest machine has speed 1 and the largest job is c. Note that this is done by simply multiplying all machine speeds by a suitable factor and all job sizes by a suitable factor – both the LHS and the RHS of (2) are scaled by the same quantity.

In order to compare Δ_1 with the profit of the algorithm, we decompose the instance into a convex combination of simpler threshold instances in Lemma 4. Here, the speeds are either all the same or take only two different values, 0 and 1. It suffices to compare the profit of the algorithm to OPT on such threshold instances.

Intuitively, if there are so few fast machines that even a relatively large job (job of weight at least 1) got assigned to a slow machine in OPT, then the original instance is mostly comparable to the threshold instance where only a few machines have speed 1 and the rest have speed 0. Even if the fastest machine gets jobs assigned to machines of speed 0 in OPT, this does not affect the profit of the algorithm relative to OPT because OPT does not profit from these jobs either. Thus we only care about jobs of weight at least 1. Because a single machine can get at most two jobs of value in the range $[1, c]$, handling this case only requires analyzing at most two jobs. The proof for this case is contained in Lemma 3.

On the other hand, if there are a lot of fast machines so that all large jobs are assigned to fast machines in OPT, then the original instance is comparable to the threshold instance where all machines have speed 1. In this case, the fastest machine can get assigned many jobs that all contribute to OPT. However, because all speeds are the same, we can deduce the worst possible sequence of jobs: after the first few jobs, all other jobs have weights forming a geometric sequence. The rest of the proof is to analyze the algorithm on this specific sequence. The detailed proof is contained in Lemma 4.

The proofs of both Lemmata 3 and 4 use the decomposable structure of the edge weights.

3.2 Analysis Details

Recall that OPT_1 is the value of the optimal solution for the instance, and OPT_2 is the value of the optimal solution for the subinstance seen by machine 2 onwards. Assume wlog that all job sizes are distinct (by perturbing job sizes infinitesimally). For $y \leq c$, let $j(y)$ be the size of the largest job $\leq y$ or 0, if no such job exists. Let $s(y)$ be the speed of the machine in the optimal solution that $j(y)$ is assigned to or 0 if $j(y) = 0$. If there is a job of size y then $s(y)$ is the speed of the machine in the optimal solution that this job is assigned to. Note that $s(y) \in [0, 1]$ is monotone increasing with $s(c) = 1$. We refer to the function s as the *speed profile*. Note that s is not a random variable. Let the *assignment sequence* $\mathbf{w} = (w, w_1, w_2, \ldots)$ denote the set of jobs assigned to the fastest machine by the algorithm where $w > w_1 > w_2 > \ldots$. Let $\max(\mathbf{w})$ denote the maximum element in the sequence $\mathbf{w}$, i.e. $\max(\mathbf{w}) = w$. In Lemma 3, we bound $OPT_1 - OPT_2$ by a function that depends *only* on $\mathbf{w}$, s, and c. Such a bound is possible because of the fact that any job can be assigned to any machine, i.e. the graph is a complete graph. The value we take for the aforementioned random variable Δ_1 turns out to be exactly this bound.

Lemma 1. $OPT_1 - OPT_2 \leq c - (c - w)s(w) + \sum_{k \geq 1} w_k \cdot s(w_k)$

Proof. Let I_1, I_2 be the instances corresponding to OPT_1 and OPT_2. I_2 is obtained from I_1 by removing the fastest machine and the set of jobs that are assigned to the fastest machine by the algorithm. Let us consider changing I_1 to I_2 in two steps: (1) Remove the fastest machine and the largest job w assigned by the algorithm to the fastest machine. (2) Remove the jobs $w_1, w_2, \ldots$. For each step, we will bound the change in the value of the optimal solution resulting in a feasible solution for I_2 and computing its value – this will be a lower bound for OPT_2.

First we analyze Step 1: OPT_1 assigns the largest job c to the fastest machine, contributing c to its value. The algorithm assigns w to the fastest machine instead of c. In OPT_1, w was assigned to a machine of speed $s(w)$. When we remove w and the fastest machine from I_1, one possible assignment to the resulting instance is obtained by placing c on the machine of speed $s(w)$. The value of the resulting solution is lower by exactly $(c + w \cdot s(w)) - c \cdot s(w) = c - (c - w)s(w)$.

Next, we analyze Step 2: Jobs $w_1, w_2, \ldots$ were assigned to machines of speeds $s(w_1), s(w_2), \ldots$ in OPT_1. When we remove jobs $w_1, w_2, \ldots$, one feasible assignment for the resulting instance is simply not to assign any jobs to the machines $s(w_1), s(w_2), \ldots$, and keep all other assignments unchanged. The value of the solution drops by exactly $\sum_{k \geq 1} w_k \cdot s(w_k)$.

Thus we exhibited a feasible solution to instance I_2 of value V where

$$OPT_1 - V = c - (c - w)s(w) + \sum_{k \geq 1} w_k \cdot s(w_k).$$

But $OPT_2 \geq V$. Hence, the lemma follows. □

We define the random variable Δ_1, a function of the assignment sequence $\mathbf{w}$ and the speed profile s, to be

$$\Delta_1(\mathbf{w}, s) = c - (c - w)s(w) + \sum_{k \geq 1} w_k \cdot s(w_k).$$

As defined, $\Delta_1(\mathbf{w}, s) \geq OPT_1 - OPT_2$. We note that even though OPT_1 and OPT_2 are functions of all the jobs in the instance, Δ_1 only depends on the subset of jobs assigned to the fastest machine by the algorithm. Our goal is to show $\mathbb{E}[A_1] = E[\max(\mathbf{w})] \geq \alpha\, \mathbb{E}[\Delta_1]$.

First, we argue that it suffices to restrict our analysis to a simple set of step function speed profiles s_t, $0 \leq t \leq 1$: For $t \in (0, 1]$, $s_t(y) = 1$ for $y \in [c^t, c]$ and $s_t(y) = 0$ for $y < c^t$. For $t = 0$, $s_0(y) = 1$ for all $y \leq c$. The proof is omitted.

Lemma 2. *Suppose that for* $t = 0$ *and for all* $t \in (0, 1]$ *such that there exists a job of weight* c^t*, we have*

$$\mathbb{E}[\max(\mathbf{w})] \geq \alpha\, \mathbb{E}[\Delta_1(\mathbf{w}, s_t)] \tag{3}$$

Then, $\mathbb{E}[\max(\mathbf{w})] \geq \alpha(\mathbb{E}[OPT_1] - \mathbb{E}[OPT_2])$.

Note that since we scaled job sizes, the thresholds (i.e interval boundaries) c^{k+x_1} should also be scaled by the same quantity (say γ). After scaling, let $x \in (0, 1]$ be such that c^x is the unique threshold from the set $\{\gamma c^{k+x_1}, k \text{ integer}\}$ in $(1, c]$. Since x_1 is uniformly distributed in $(0, 1]$, x is also uniformly distributed in $(0, 1]$. Having defined x thus, the interval boundaries picked by the algorithm for the fastest machine are c^{x+k} for integers k.

We prove (3) for $\alpha = \min\left(\frac{c-1}{c\ln(c)}, h(c)\right)$ in two separate lemmata, one for the case $t > 0$ (Lemma 3) and the other for the case $t = 0$ (Lemma 4). Recall that the expression for Δ_1 only depends on the subset of jobs assigned to the fastest machine. We call a job a *local maximum* if it is larger than all jobs preceding it. Since the algorithm assigns a new job to the fastest machine if and only if it falls in a larger interval than the current largest job, it follows that any job assigned to the fastest machine must be a local maximum.

Define $m_S(y)$ to be the minimum job in the sequence of all local maxima in the range $(y, cy]$, i.e., the first job larger than y and at most cy, if such a job exists and 0 otherwise. We use $m_S(y)$ in two ways. (1) We define $u_0 = m_S(1)$. Note that u_0 is not a random variable. We use u_0 in Lemma 3 to prove the desired statement for $t > 0$. Specifically, we use u_0 to compute (i) a lower bound for $\mathbb{E}[w]$ as a function of u_0 (and not of any other jobs) and (ii) an upper bound for $\mathbb{E}[\Delta_1]$ as a function of u_0. Combining (i) and (ii) we prove that the desired inequality holds for all u_0. (2) In Lemma 4 we bound $\mathbb{E}[\sum_{k\geq 1} w_k]$ by a sum of $m_S(y)$ over suitable values of y. This simplifies the analysis since the elements in the subsequence of *all* local maxima are *not* random variables, while the values in $\mathbf{w}$ are random variables.

We first prove some simple properties of u_0 that we will use:

Claim. (1) $u_0 \leq w$ and (2) $u_0 \geq w_1$.

Proof. $u_0 \leq w$ as u_0 is the minimum element in the sequence of all local maxima in $(1, c]$ and w is the element from the interval $(1, c]$ picked by the algorithm.

w_1 is the minimum element in the sequence of local maxima in the range $(c^{x-k-1}, c^{x-k}]$ for $x \in (0, 1]$ and k a non-negative integer. Either $u_0 \geq c^{x-k} \geq w_1$, or u_0 also falls into $(c^{x-k-1}, c^{x-k}]$ and $u_0 \geq w_1$ follows from the fact that w_1 is the smallest local maximum in this range, while u_0 is an arbitrary local maximum in this range. □

The next lemmata conclude our algorithm analysis. Proofs are omitted.

Lemma 3. *For $c \geq e$, $t \in (0, 1]$ such that there exists a job of weight c^t, and $\alpha = \min\left(\frac{c-1}{c\ln(c)}, h(c)\right)$, we have*

$$\alpha\, \mathbb{E}[\Delta_1(\mathbf{w}, s_t)] \leq \mathbb{E}[\max(\mathbf{w})]$$

Lemma 4. *For $s_0(x) \equiv 1$ and $\alpha = h(c)$, we have* $\mathbb{E}[\max(\mathbf{w})] \geq \alpha\, \mathbb{E}[\Delta_1(\mathbf{w}, s_0)]$.

4 Upper Bound for Deterministic Algorithms

To prove an upper bound of a, we construct an instance such that any deterministic algorithm has competitive ratio at most a for some prefix of the request sequence. The instance has one *fast* machine of speed $r > 1$ and n *slow* machines of speed 1. The request sequence has non-decreasing job sizes satisfying a certain oscillatory recurrence relation. The full proof is in the full version.

Theorem 3. *The competitive ratio of any deterministic algorithm is at most* $(\sqrt{5}-1)/2+\epsilon \approx 0.618034+\epsilon$ *for any* $\epsilon > 0$.

5 Upper Bound for Randomized Algorithms

To establish the bound for randomized algorithms, we use Yao's principle and show an upper bound on the expected competitive ratio of any deterministic algorithm on a distribution of instances. The construction uses one fast machine of speed 1 and n slow machines of speed 1/4. The request sequence has non-decreasing sizes 2^i. The prefix of this sequence ending with size 2^i is presented to the algorithm with probability $c/2^i$, where c is a normalizing constant. We show that the best algorithm for this sequence achieves at most $cn+1$ while the optimal algorithm achieves roughly $5nc/4$. The proof is in the full version.

Theorem 4. *The competitive ratio of any randomized algorithm against an oblivious adversary is at most* $0.8+\epsilon$ *for any* $\epsilon > 0$.

Acknowledgments. MC was supported by NSF awards CCF 0832797, AF 0916218 and a Google research award. MH's support: The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement no. 340506 and the Austrian Science Fund (FWF) grant P23499-N23. HN was supported by NSF awards CCF 0832797, AF 0916218, a Google research award, and a Gordon Wu fellowship.

References

1. Aggarwal, G., Goel, G., Karande, C., Mehta, A.: Online vertex-weighted bipartite matching and single-bid budgeted allocations. In: SODA, pp. 1253–1264 (2011)
2. Agrawal, S., Wang, Z., Ye, Y.: A dynamic near-optimal algorithm for online linear programming. CoRR, abs/0911.2974 (2009)
3. Badanidiyuru Varadaraja, A.: Buyback problem-approximate matroid intersection with cancellation costs. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part I. LNCS, vol. 6755, pp. 379–390. Springer, Heidelberg (2011)
4. Ashwinkumar, B.V., Kleinberg, R.: Randomized online algorithms for the buyback problem. In: Leonardi, S. (ed.) WINE 2009. LNCS, vol. 5929, pp. 529–536. Springer, Heidelberg (2009)
5. Aspnes, J., Azar, Y., Fiat, A., Plotkin, S., Waarts, O.: On-line routing of virtual circuits with applications to load balancing and machine scheduling. Journal of the ACM (JACM) 44(3), 486–504 (1997)
6. Azar, Y.: On-line load balancing. Online Algorithms, 178–195 (1998)
7. Babaioff, M., Hartline, J.D., Kleinberg, R.D.: Selling ad campaigns: online algorithms with cancellations. In: EC, pp. 61–70 (2009)
8. Beck, A., Newman, D.: Yet more on the linear search problem. Israel journal of mathematics 8(4), 419–429 (1970)
9. Berman, P., Charikar, M., Karpinski, M.: On-line load balancing for related machines. Journal of Algorithms 35(1), 108–121 (2000)
10. Birnbaum, B.E., Mathieu, C.: On-line bipartite matching made simple. SIGACT News 39(1), 80–87 (2008)

11. Buchbinder, N., Jain, K., Naor, J(S.): Online primal-dual algorithms for maximizing ad-auctions revenue. In: Arge, L., Hoffmann, M., Welzl, E. (eds.) ESA 2007. LNCS, vol. 4698, pp. 253–264. Springer, Heidelberg (2007)
12. Chakrabarti, S., Phillips, C., Schulz, A., Shmoys, D., Stein, C., Wein, J.: Improved scheduling algorithms for minsum criteria. In: Meyer, F., Monien, B. (eds.) Automata, Languages and Programming. LNCS, vol. 1099, pp. 646–657. Springer, Heidelberg (1996)
13. Constantin, F., Feldman, J., Muthukrishnan, S., Pál, M.: An online mechanism for ad slot reservations with cancellations. In: SODA, pp. 1265–1274 (2009)
14. Devanur, N.R., Hayes, T.P.: The adwords problem: Online keyword matching with budgeted bidders under random permutations. In: EC, pp. 71–78 (2009)
15. Devanur, N.R., Jain, K., Kleinberg, R.: Randomized primal-dual analysis of ranking for online bipartite matching. In: SODA (2013)
16. Epstein, L., Levin, A., Segev, D., Weimann, O.: Improved bounds for online preemptive matching. CoRR, abs/1207.1788 (2012)
17. Epstein, L., Sgall, J.: A lower bound for on-line scheduling on uniformly related machines. Operations Research Letters 26(1), 17–22 (2000)
18. Feldman, J., Korula, N., Mirrokni, V., Muthukrishnan, S., Pál, M.: Online ad assignment with free disposal. In: Leonardi, S. (ed.) WINE 2009. LNCS, vol. 5929, pp. 374–385. Springer, Heidelberg (2009)
19. Feldman, J., Mehta, A., Mirrokni, V., Muthukrishnan, S.: Online stochastic matching: Beating 1-1/e. In: 50th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2009, pp. 117–126. IEEE (2009)
20. Gal, S.: Search games. Mathematics in science and engeneering, vol. 149 (1980)
21. Goel, G., Mehta, A.: Online budgeted matching in random input models with applications to adwords. In: SODA, pp. 982–991 (2008)
22. Goel, G., Tripathi, P.: Matching with our eyes closed. In: FOCS, pp. 718–727 (2012)
23. Goemans, M., Kleinberg, J.: An improved approximation ratio for the minimum latency problem. Mathematical Programming 82(1), 111–124 (1998)
24. Haeupler, B., Mirrokni, V.S., Zadimoghaddam, M.: Online stochastic weighted matching: Improved approximation algorithms. In: Chen, N., Elkind, E., Koutsoupias, E. (eds.) Internet and Network Economics. LNCS, vol. 7090, pp. 170–181. Springer, Heidelberg (2011)
25. Karande, C., Mehta, A., Tripathi, P.: Online bipartite matching with unknown distributions. In: STOC, pp. 587–596 (2011)
26. Karp, R.M., Vazirani, U.V., Vazirani, V.V.: An optimal algorithm for on-line bipartite matching. In: STOC, pp. 352–358 (1990)
27. Mahdian, M., Yan, Q.: Online bipartite matching with random arrivals: an approach based on strongly factor-revealing LPs. In: STOC, pp. 597–606 (2011)
28. Manshadi, V., Gharan, S., Saberi, A.: Online stochastic matching: Online actions based on offline statistics. In: Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 1285–1294. SIAM (2011)
29. McGregor, A.: Finding graph matchings in data streams. In: Chekuri, C., Jansen, K., Rolim, J.D.P., Trevisan, L. (eds.) APPROX 2005 and RANDOM 2005. LNCS, vol. 3624, pp. 170–181. Springer, Heidelberg (2005)
30. Mehta, A., Panigrahi, D.: Online matching with stochastic rewards. In: FOCS, pp. 728–737 (2012)
31. Mehta, A., Saberi, A., Vazirani, U.V., Vazirani, V.V.: Adwords and generalized on-line matching. In: FOCS, pp. 264–273 (2005)
32. Poloczek, M., Szegedy, M.: Randomized greedy algorithms for the maximum matching problem with new analysis. In: FOCS, pp. 708–717 (2012)

A Faster Algorithm for Computing Straight Skeletons

Siu-Wing Cheng[1], Liam Mencel[2,⋆], and Antoine Vigneron[2]

[1] The Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong
scheng@cse.ust.hk

[2] King Abdullah University of Science and Technology
Thuwal 23955-6900, Saudi Arabia
{antoine.vigneron,liam.mencel}@kaust.edu.sa

Abstract. We present a new algorithm for computing the straight skeleton of a polygon. For a polygon with n vertices, among which r are reflex vertices, we give a deterministic algorithm that reduces the straight skeleton computation to a motorcycle graph computation in $O(n(\log n)\log r)$ time. It improves on the previously best known algorithm for this reduction, which is randomized, and runs in expected $O(n\sqrt{h+1}\log^2 n)$ time for a polygon with h holes. Using known motorcycle graph algorithms, our result yields improved time bounds for computing straight skeletons. In particular, we can compute the straight skeleton of a non-degenerate polygon in $O(n(\log n)\log r + r^{4/3+\varepsilon})$ time for any $\varepsilon > 0$. On degenerate input, our time bound increases to $O(n(\log n)\log r + r^{17/11+\varepsilon})$.

1 Introduction

The straight skeleton of a polygon is defined as the trace of the vertices when the polygon shrinks, each edge moving at the same speed inwards in a perpendicular direction to its orientation. (See Fig. 1.) It differs from the medial axis [7] in that it is a straight line graph embedded in the original polygon, while the medial axis may have parabolic edges. The notion was introduced by Aichholzer et al. [1] in 1995, who gave the earliest algorithm for computing the straight skeleton. However, the concept has been recognized as early as 1877 by von Peschka [20], in his interpretation as projection of roof surfaces.

The straight skeleton has numerous applications in computer graphics. It allows to compute offset polygons [14], which is a standard operation in CAD. Other applications include architectural modelling [19], biomedical image processing [8], city model reconstruction [10], computational origami [11,12,13] and polyhedral surface reconstruction [2,9,15]. Improving the efficiency of straight skeleton algorithms increases the speed of related tools in geometric computing.

The first algorithm by Aichholzer et al. [1] runs in $O(n^2 \log n)$ time, and simulates the shrinking process discretely. Eppstein and Erickson [14] developed the

⋆ Liam Mencel was supported by KAUST base funding. Research supported by the Research Grants Council, Hong Kong, China (project no. 611711).

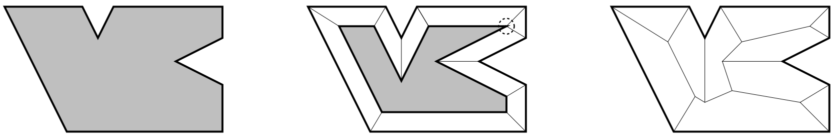

(a) The input polygon $\mathcal{P}$ (b) An offset of $\mathcal{P}$ (c) Straight skeleton $\mathcal{S}$

Fig. 1. The straight skeleton is obtained by shrinking the input polygon $\mathcal{P}$

first sub-quadratic algorithm, which runs in $O(n^{17/11+\varepsilon})$ time. In their work, they proposed motorcycle graphs as a means of encapsulating the main difficulty in computing straight skeletons. Cheng and Vigneron [6] expanded on this notion by reducing the straight skeleton problem in non-degenerate cases to a motorcycle graph computation and a lower envelope computation. This reduction was later extended to degenerate cases by Held and Huber [17]. Cheng and Vigneron gave an algorithm for the lower envelope computation on a non-degenerate polygon with h holes, which runs in $O(n\sqrt{h+1}\log^2 n)$ expected time. They also provided a method for solving the motorcycle graph problem in $O(n\sqrt{n}\log n)$ time. Putting the two together gives an algorithm which solves the straight skeleton problem in $O(n\sqrt{h+1}\log^2 n + r\sqrt{r}\log r)$ expected time, where r is the number of reflex vertices.

Comparison with previous work. Recently, Vigneron and Yan [21] found a faster, $O(n^{4/3+\varepsilon})$-time algorithm for computing motorcycle graphs. It thus removed one bottleneck in straight skeleton computation. In this paper we remove the second bottleneck: We give a faster reduction to the motorcycle graph problem. Our algorithm performs this reduction in deterministic $O(n(\log n)\log r)$ time, improving on the previously best known algorithm, which is randomized and runs in expected $O(n\sqrt{h+1}\log^2 n)$ time [6]. Recently, Bowers independently discovered an $O(n\log n)$-time, deterministic algorithm to perform this reduction in the case of simple polygons, using a very different approach [4].

Using known algorithms for computing motorcycle graphs, our reduction yields faster algorithms for computing the straight skeleton. In particular, using the algorithm by Vigneron and Yan [21], we can compute the straight skeleton of a non-degenerate polygon in $O(n(\log n)\log r + r^{4/3+\varepsilon})$ time for any $\varepsilon > 0$. On degenerate input, we use Eppstein and Erickson's algorithm, and our time bound increases to $O(n(\log n)\log r + r^{17/11+\varepsilon})$. For simple polygons whose coordinates are $O(\log n)$-bit rational numbers, we can compute the straight skeleton in $O(n\log^2 n)$ time using the motorcycle graph algorithm by Vigneron and Yan [21] (even in degenerate cases). Table 1 summarizes the previously known results and compares with our new algorithm.

Our approach. We use the known reduction to a lower envelope of slabs in 3D [6,17]: First a motorcycle graph induced by the input polygon is computed, and then this graph is used to define a set of slabs in 3D. The lower envelope of

Table 1. O^* denotes the expected time bound of a randomized algorithm, and O is for deterministic algorithms. To make the comparison easier, we replaced the number of holes h with r, as $h = O(r)$.

	Previously best known	This paper
Arbitrary polygon	$O(n^{8/11+\varepsilon}r^{9/11+\varepsilon})$ [14]	$O(n(\log n)\log r + r^{17/11+\varepsilon})$
Non-degenerate polygon	$O^*(n\sqrt{r}\log^2 n)$ [6]	$O(n(\log n)\log r + r^{4/3+\varepsilon})$
Simple pol., arbitrary	$O^*(n\log^2 n + r^{17/11+\varepsilon})$ [6,14]	$O(n(\log n)\log r + r^{17/11+\varepsilon})$
Simple pol., $O(\log n)$ bits	$O^*(n\log^2 n)$ [6,21]	$O(n\log^2 n)$

these slabs is a terrain, whose edges vertically project to the straight skeleton on the horizontal plane. (See Section 2.)

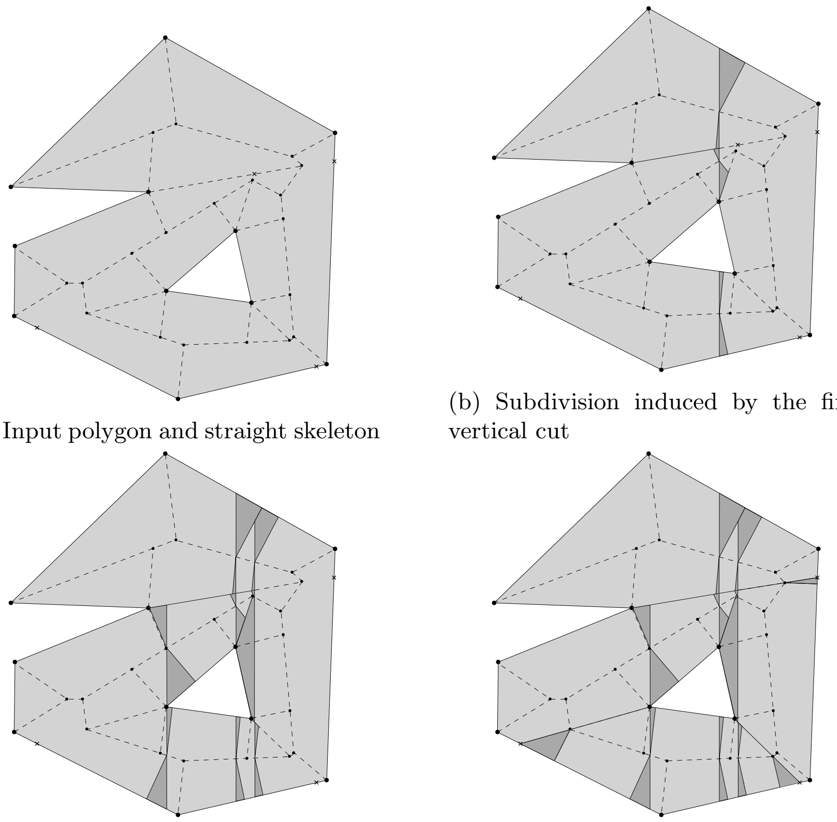
The difficulty is that these slabs may cross, and in general their lower envelope is a non-convex terrain, so known algorithms for computing lower envelopes of triangles would be too slow for our purpose. Our approach is thus to remove non-convex features: We compute a subdivision of the input polygon into convex cells such that, above each cell of this subdivision, the terrain is convex. Then the portion of the terrain above each cell can be computed efficiently, as it reduces to computing a lower envelope of planes in 3D. The subdivision is computed recursively, using a divide and conquer approach, in two stages.

During the first stage (Section 3), we partition using vertical lines, that is, lines parallel to the y-axis. At each step, we pick the vertical line ℓ through the median motorcycle vertex in the current cell. We first cut the cell using ℓ, and we compute the restriction of the terrain to the space above ℓ, which forms a polyline. It can be computed in near-linear time, as it reduces to computing a lower envelope of line segments in the vertical plane through ℓ. Then we cut the cell using the steepest descent paths from the vertices of this polyline. (See Fig. 2b.) We recurse until the current cell does not contain any vertex of the motorcycle graph. (See Fig. 2c.)

The first step ensures that the cells of the subdivision are convex. However, valleys (non-convex edges) may still enter the interior of the cells. So our second stage (Section 4) recursively partitions cells using lines that split the set of valleys of the current cell, instead of vertical lines. (See Fig. 2d.) As the first stage results in a partition where the restriction of the motorcycle graph to any cell is outerplanar, we can perform this subdivision efficiently by divide and conquer.

Each time we partition a cell, we know which slabs contribute to the child cells, as we know the terrain along the vertical plane through the cutting line. In addition, we will argue via careful analysis that our divide and conquer approach avoids slabs being used in too many iterations, and hence the algorithm completes in $O(n(\log n)\log r)$ time.

Due to space limitation, some proofs are missing from this extended abstract. A more detailed description of our algorithm, as well as the missing proofs, can be found in the full version of this paper [5]. We state here our main result:



(a) Input polygon and straight skeleton

(b) Subdivision induced by the first vertical cut

(c) Result of the vertical subdivision

(d) Final subdivision

Fig. 2. Example of subdivision computed by our algorithm

Theorem 1. *Given a polygon $\mathcal{P}$ with n vertices, r of which being reflex vertices, and given the motorcycle graph induced by $\mathcal{P}$, we can compute the straight skeleton of $\mathcal{P}$ in $O(n(\log n)\log r)$ time.*

Our algorithm does not handle weighted straight skeletons [14] (where edges move at different speeds during the shrink process), because the reduction to a lower envelope of slabs does not hold in this case.

2 Notations and Preliminaries

The input polygon is denoted by $\mathcal{P}$. A *reflex vertex* of a polygon is a vertex at which the internal angle is more than π. $\mathcal{P}$ has n vertices, among which r are reflex vertices. We work in $\mathbb{R}^3$ with $\mathcal{P}$ lying flat in the xy-plane. The z-axis becomes analogous to the time dimension. We say that a line, or a line segment, is *vertical*, if it is parallel to the y-axis, and we say that a plane is vertical if it is orthogonal to the xy-plane. The boundary of a set A is denote by ∂A. We denote by $\overline{pq}$ the line segment with endpoints p, q.

Terrain. At any time, the horizontal plane $z = t$ contains a snapshot of $\mathcal{P}$ after shrinking for t units of time. While the shrinking polygon moves vertically at unit speed, faces are formed as the trace of the edges, and these faces make an angle $\pi/4$ with the xy-plane. The surface formed by the traces of the edges is the *terrain* $\mathcal{T}$. (See Fig. 3 a.) The traces of the vertices of $\mathcal{P}$ form the set of edges of $\mathcal{T}$. An edge e of $\mathcal{T}$ is *convex* if there is a plane through e that is above the two faces bounding e. The edges of $\mathcal{T}$ corresponding to the traces of the reflex vertices will be referred to as *valleys*. Valleys are the only non-convex edges on $\mathcal{T}$. The other edges, which are convex, are called *ridges*. The *straight skeleton* $\mathcal{S}$ is the graph obtained by projecting the edges and vertices of $\mathcal{T}$ orthogonally onto the xy-plane. We also call valleys and ridges the edges of $\mathcal{S}$ that are obtained by projecting valleys and ridges of $\mathcal{T}$ onto the xy-plane.

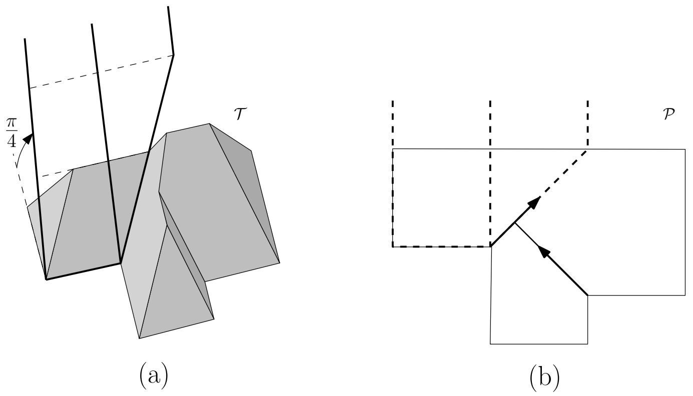


Fig. 3. Illustration of the two different types of slabs. (a) The terrain $\mathcal{T}$, an edge slab and motorcycle slab. This terrain has two valleys, adjacent to the two reflex vertices of the polygon. (b) The motorcycle graph associated with $\mathcal{P}$ and the boundaries of the edge slab and the motorcycle slab viewed from above.

Motorcycle graph. Our algorithm for computing the straight skeleton assumes that a motorcycle graph induced by $\mathcal{P}$ is precomputed [6]. This graph is defined as follows. A motorcycle is a point moving at a fixed velocity. We place a motorcycle at each reflex vertex of $\mathcal{P}$. The velocity of a motorcycle is the same as the velocity of the corresponding reflex vertex when $\mathcal{P}$ is shrunk, so its direction is the bisector of the interior angle, and its speed is $1/\sin(\theta/2)$, where θ is the exterior angle at the reflex vertex. (See Fig. 4a.)

The motorcycles begin moving simultaneously. They each leave behind a track as they move. When a motorcycle collides with either another motorcycle's track or the boundary of $\mathcal{P}$, the colliding motorcycle halts permanently. (In degenerate

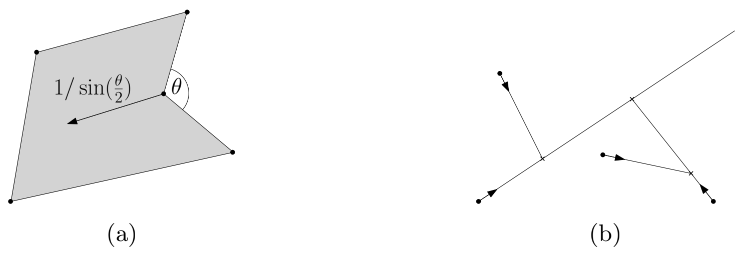


Fig. 4. Motorcycle graph

cases, a motorcycle may also collide head-on with another motorcycle, but for now we rule out this case.) After all motorcycles stop, the tracks form a planar graph called the *motorcycle graph induced by* $\mathcal{P}$. (see Fig. 4b.)

General position assumptions. To simplify the description and the analysis of our algorithm, we assume that the polygon is in general position. No edge of $\mathcal{P}$ or $\mathcal{S}$ is vertical. No two motorcycles collide with each other in the motorcycle graph, and thus each valley is adjacent to some reflex vertex. Each vertex in the straight skeleton graph has degree 1 or 3. Our results, however, generalize to degenerate polygons.

Lifting map. The *lifted* version $\hat{p}$ of a point $p \in \mathcal{P}$ is the point on $\mathcal{T}$ that is vertically above p. In other words, $\hat{p}$ is the point of $\mathcal{T}$ that projects orthogonally to p on the xy-plane. We may also apply this transformation to a line segment s in the xy-plane, then $\hat{s}$ is a polyline in $\mathcal{T}$. We will abuse notation and denote by $\hat{\mathcal{G}}$ a lifted version of $\mathcal{G}$ where the height of a point is the time at which the corresponding motorcycle reaches it. Then the lifted version $\hat{e}$ of an edge e of $\mathcal{G}$ does not lie entirely on $\mathcal{T}$, but it contains the corresponding valley, and the remaining part of $\hat{e}$ lies above $\mathcal{T}$ [6]. (See Fig. 3a.)

Given a point $\hat{p}$ that lies in the interior of a face f of $\mathcal{T}$, there is a unique steepest descent path from $\hat{p}$ to the boundary of $\mathcal{P}$. This path consists either of a straight line segment orthogonal to the base edge e of f, or it consists of a segment going straight to a valley, and then follows this valley. (In degenerate cases, the path may follow several valleys consecutively.) If $\hat{p}$ is on a ridge, then two such descent paths from p exist, and if $\hat{p}$ is a convex vertex, then there are three such paths. (See Fig. 5c.)

Reduction to a lower envelope. Following Eppstein and Erickson [14], Cheng and Vigneron [6], and Held and Huber [17], we use a construction of the straight skeleton based on the lower envelope of a set of three-dimensional slabs. Each edge e of $\mathcal{P}$ defines an *edge slab*, which is a 2-dimensional half-strip at an angle of $\pi/4$ to the xy-plane, bounded below by e and along the sides by rays perpendicular to e. (See Fig. 3.) We say that e is the *source* of this edge slab.

For each reflex vertex $v = e \cap e'$, where e and e' are edges of $\mathcal{P}$, we define two *motorcycle slabs* making angles of $\pi/4$ to the xy-plane. One motorcycle slab is

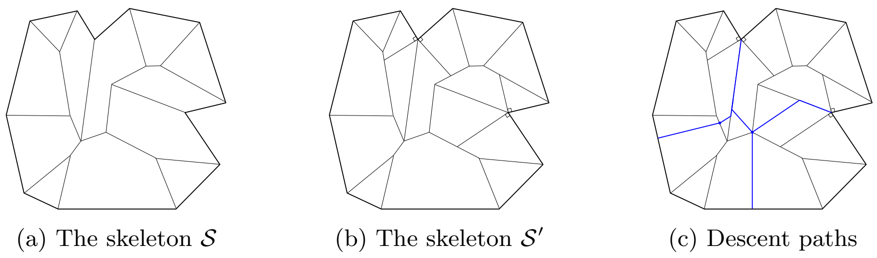

(a) The skeleton $\mathcal{S}$ (b) The skeleton $\mathcal{S}'$ (c) Descent paths

Fig. 5. The polygon $\mathcal{P}$, its skeletons, and descent paths

bounded below by the edge of $\hat{\mathcal{G}}$ incident to v and is bounded on the sides by two rays from each end of this edge in the ascent direction of e. The other motorcycle slab is defined similarly with e replaced by e'. The *source* of a motorcycle slab is the corresponding edge of $\hat{\mathcal{G}}$. Cheng and Vigneron [6] proved the following result, which was extended to degenerate cases by Huber and Held [16]:

Theorem 2. *The terrain $\mathcal{T}$ is the restriction of the lower envelope of the edge slabs and the motorcycle slabs to the space vertically above the polygon.*

Our algorithm constructs a graph $\mathcal{S}'$, which is obtained from $\mathcal{S}$ by adding two edges at each reflex vertex v of $\mathcal{P}$ going inwards and orthogonally to each edge of $\mathcal{P}$ incident to v. (See Fig. 5b.) We also include the edges of $\mathcal{P}$ into $\mathcal{S}'$. It means that each face f of $\mathcal{S}'$ corresponds to exactly one slab. More precisely, a face is the vertical projection of $\mathcal{T} \cap \sigma$ to the xy-plane for some slab σ. By contrast, in the original straight skeleton $\mathcal{S}$, a face incident to a reflex vertex corresponds to one edge slab and one motorcycle slab.

3 Computing the Vertical Subdivision

In this section, we describe and we analyze the first stage of our algorithm, where the input polygon $\mathcal{P}$ is recursively partitioned using vertical cuts. The corresponding procedure is called Divide-Vertical. It results in a subdivision of the input polygon $\mathcal{P}$, such that any cell of this subdivision has the following property: It does not contain any vertex of $\mathcal{G}$ in its interior, or it is contained in the union of two faces of $\mathcal{S}'$.

3.1 Subdivision Induced by a Vertical Cut

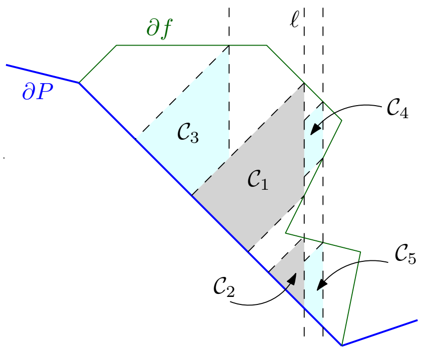
At any step of the algorithm, we maintain a planar subdivision $\mathcal{K}(\mathcal{P})$, which is a partition of the input polygon $\mathcal{P}$ into polygonal cells. Each cell is a polygon, hence it is connected. A cell $\mathcal{C}$ in the current subdivision $\mathcal{K}(\mathcal{P})$ may be further subdivided as follows.

Let ℓ be a vertical line through a vertex of $\mathcal{G}$. We assume that ℓ intersects $\mathcal{C}$, and hence $\mathcal{C} \cap \ell$ consists of several line segments $s_1, \ldots, s_q$. These line segments are introduced as new boundary edges in $\mathcal{K}(\mathcal{P})$; they are called the *vertical edges* of $\mathcal{K}(\mathcal{P})$. They may be further subdivided during the course of the algorithm, and we still call the resulting edges vertical edges.

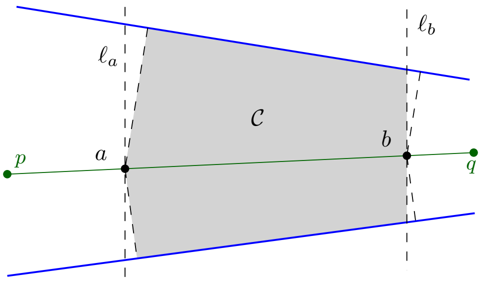
We then insert non-vertical edges along steepest descent paths, as follows. Note that we are able to efficiently compute the intersection $\mathcal{S}' \cap \ell$ without knowing $\mathcal{S}'$. To do this, we make use of an algorithm by Hershberger [18] and compute the lower envelope of the slabs restricted to the vertical plane through ℓ, using $O(n \log n)$ time. The points at which this envelope changes angle are precisely the points on $\mathcal{T}$ which project onto $\mathcal{S}' \cap \ell$. Each intersection point $p \in s_j \cap \mathcal{S}'$ has a lifted version $\hat{p}$ on $\mathcal{T}$. By our non-degeneracy assumptions, there are at most three steepest descent paths to $\partial\mathcal{C}$ from $\hat{p}$. The vertical projections of these paths onto $\mathcal{C}$ are also inserted as new edges in $\mathcal{K}(\mathcal{P})$. The resulting partition of $\mathcal{C}$ is the *subdivision induced by* ℓ. (See Fig. 2.)

We denote by $\mathcal{C}_1, \mathcal{C}_2, \ldots$ the cells of $\mathcal{K}(\mathcal{P})$ that are constructed during the course of the algorithm. Let ℓ_i^- and ℓ_i^+ denote the vertical lines through the leftmost and rightmost point of $\mathcal{C}_i$, respectively. When we perform one step of the subdivision, each new cell lies entirely to the left or to the right of the splitting line, and thus by induction, any vertical edge of a cell $\mathcal{C}_i$ either lies in ℓ_i^- or ℓ_i^+.

An *empty cell* is a cell of $\mathcal{K}(\mathcal{P})$ whose interior does not overlap with $\mathcal{S}'$. (See Fig. 6a.) Thus an empty cell is entirely contained in a face of $\mathcal{S}'$. Another type of cell, called a *wedge*, will play an important role in the analysis of our algorithm. Let $\overline{pq}$ be a ridge of $\mathcal{S}'$, and let a, b be two points in the interior of $\overline{pq}$. Let ℓ_a and ℓ_b be the vertical lines through a and b, respectively. Consider the subdivision of $\mathcal{P}$ obtained by inserting vertical boundaries along ℓ_a and ℓ_b, and the four descent paths from a and b. (See Fig. 6b.) The cell of this subdivision containing $\overline{ab}$ is called the wedge corresponding to $\overline{ab}$.



(a) The cells $\mathcal{C}_1, \ldots, \mathcal{C}_5$ are empty. The first cut is performed along ℓ



(b) The wedge $\mathcal{C}$ corresponding to $\overline{ab}$

Fig. 6. Empty cells and a wedge

3.2 Data Structure

During the course of the algorithm, we maintain the polygon $\mathcal{P}$ and its subdivision $\mathcal{K}(\mathcal{P})$ in a doubly-connected edge list [3]. So each cell $\mathcal{C}_i$ is represented by a circular list of edges, or several if it has holes. In the following, we show how we augment these chains so that they record incidences between the boundary of $\mathcal{C}_i$ and the faces of $\mathcal{S}'$.

For each cell $\mathcal{C}_i$, let $\mathcal{S}'_i$ be the subdivision of $\mathcal{C}_i$ induced by $\mathcal{S}'$. So the faces of $\mathcal{S}'_i$ are the connected components of $\mathcal{C}_i \setminus \mathcal{S}'$. Let Q denote a circular list of edges that form one component of $\partial\mathcal{C}_i$. We subdivide each vertical edge of Q at each intersection point with an edge of $\mathcal{S}'$. Now each edge e of Q bounds exactly one face f_j of $\mathcal{S}'_i$. We store a pointer from e to the slab σ_j corresponding to f_j. In addition, for each vertex of Q which is a reflex vertex of $\mathcal{P}$, we store pointers to the two corresponding motorcycle slabs. We call this data structure a *face list*. So we store one face list for each connected component of $\partial\mathcal{C}_i$.

We say that a vertex v of the motorcycle graph $\mathcal{G}$ *conflicts* with a cell $\mathcal{C}_i$ of $\mathcal{K}(\mathcal{P})$ if either v lies in the interior of $\mathcal{C}_i$, or v is a reflex vertex of $\partial\mathcal{C}_i$. We also store the list of all the vertices conflicting with each cell $\mathcal{C}_i$. This list V_i is called the *vertex conflict list* of $\mathcal{C}_i$. The size of this list is denoted by v_i. In summary, our data structure consists of:

- A doubly-connected edge list storing $\mathcal{K}(\mathcal{P})$.
- The face lists and the vertex conflict list V_i of each cell $\mathcal{C}_i$.

We say that an edge e of $\mathcal{S}'$ conflicts with the cell $\mathcal{C}_i$ if it intersects the interior of $\mathcal{C}_i$. So any edge of $\mathcal{S}'_i$ that is not on $\partial\mathcal{C}_i$ is of the form $e \cap \mathcal{C}_i$ for some edge e of $\mathcal{S}'$ conflicting with $\mathcal{C}_i$. We denote by c_i the number of edges conflicting with $\mathcal{C}_i$. During the course of the algorithm, we do not necessarily know all the edges conflicting with a cell $\mathcal{C}_i$, and we don't even know c_i, but this quantity will be useful for analyzing the running time. In particular, it allows to bound the size of the data structure for $\mathcal{C}_i$. (The proof is omitted due to space limitation.)

Lemma 1. *If $\mathcal{C}_i$ is non-empty, then the total size of the face lists of $\mathcal{C}_i$ is $O(c_i)$. In particular, it implies that $\partial\mathcal{C}_i$ has $O(c_i)$ edges, and $\mathcal{C}_i$ overlaps $O(c_i)$ faces of $\mathcal{S}'$. On the other hand, if $\mathcal{C}_i$ is empty, then the total size is $O(1)$, and thus $\partial\mathcal{C}_i$ has $O(1)$ edges.*

3.3 Algorithm

Our algorithm partitions $\mathcal{P}$ recursively, using vertical cuts, as in Sect. 3.1. A cell $\mathcal{C}_i$ is subdivided along a vertical cut ℓ through its median conflicting vertex, so the vertex conflict lists of the new cells will be at most half the size of the conflict lists of $\mathcal{C}_i$. When the vertex conflict list of $\mathcal{C}_i$ is empty, we call the procedure Divide-Valley. If $\mathcal{C}_i$ is empty or is a wedge, then we stop subdividing $\mathcal{C}_i$, and it becomes a *leaf cell*.

In the induced subdivision, the descent paths cannot cross, and by construction they do not cross the vertical boundary edges. Each edge of $\mathcal{S}'_i$ may create at most three such descent paths, so we create $O(c_i)$ such new descent paths. There are also $O(c_i)$ new vertical edges, so we can update the doubly-connected edge list in time $O(c_i \log c_i)$ by plane sweep. Using an additional $O(v_i \log c_i)$ time, we can update the vertex conflict lists during this plane sweep. The face lists can be updated in overall $O(c_i)$ time. It follows that:

Lemma 2. *We can compute the subdivision of a non-empty cell $\mathcal{C}_i$ induced by a line through its median conflicting vertex, and update our data structure accordingly, in $O((c_i + v_i) \log c_i)$ time.*

3.4 Analysis

Due to space limitation, we only give a sketch of proof for the running time of our algorithm. By Lemma 2, we only need to bound the total size of the conflict lists during the course of the algorithm. As there are only $2r$ motorcycle vertices, and each cell contains at most half as many as its parent, the total size of the vertex conflict lists is $O(r \log r)$.

We also show that the sum of the sizes of the edge conflict lists is $O(n \log r)$. We split into two cases. First, consider the cells containing vertices of $\mathcal{S}'$. At each subdivision, a vertex of $\mathcal{S}'$ in the cell being subdivided moves to a cell whose vertex conflict list has at most half the size of its parent's, so each vertex of $\mathcal{S}'$ is contained in $O(\log r)$ cells throughout the algorithm. Hence we create $O(n \log r)$ such cells. We then consider cells that overlap $\mathcal{S}'$, but none of its vertices. We argue that these cells must be wedges, and that each edge of $\mathcal{S}'$ yields $O(\log r)$ wedges.

Lemma 3. *The vertical subdivision procedure completes in $O(n(\log n) \log r)$ time. The cells of the resulting subdivision are either empty cells, wedges, or do not contain any motorcycle vertex in their interior. They are simply connected, and the only reflex vertices on their boundaries are along valleys.*

4 Cutting Between Valleys

In this section, we describe the second stage of the algorithm. Let $\mathcal{C}_i$ be a cell of $\mathcal{K}(\mathcal{P})$ constructed by DIVIDE-VERTICAL on which we call DIVIDE-VALLEY. The first stage of our algorithm ensures that $\mathcal{C}_i$ is convex and does not contain any reflex vertex in its interior. Let R_i denote the set of valleys that conflict with $\mathcal{C}_i$, and let r_i denote its cardinality. The *extended valley* e' corresponding to a valley $e \in R_i$ is the segment obtained by extending e until it meets the boundary $\partial\mathcal{C}_i$ of the cell. As $\mathcal{C}_i$ does not contain any motorcycle vertex in its interior, the extended valleys of $\mathcal{C}_i$ do not cross. So the extended valleys form an outerplanar graph with outer face $\partial\mathcal{C}_i$. (See Fig. 7.)

If $\mathcal{C}_i$ conflicts with at least one valley, we first construct a *balanced cut*, which is a chord s of $\partial\mathcal{C}_i$ such that there are at most $2r_i/3$ extended valleys on each side of s. (See Fig. 7, middle.) The existence and the algorithm for computing s are explained in the full version of this paper [5]. This balanced cut plays exactly the

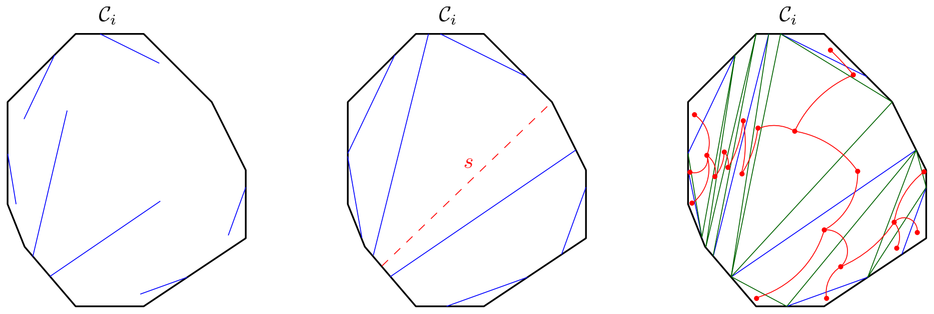


Fig. 7. (Left) The cell $\mathcal{C}_i$ and the conflicting valleys. (Middle) The extended valleys, and a balanced cut. (Right) The triangulation and its dual graph.

same role as the vertical edges $s_1, \dots, s_q$ along the cutting line that were used in DIVIDE-VERTICAL. So we insert s as a new boundary segment, we compute its lifted version $\hat{s}$, and at each crossing between s and $\mathcal{S}'$, intersects the descent paths as new boundary edges.

We repeat this process recursively, and we stop recursing whenever a cell does not conflict with any valley. All the structural results in Sect. 3 still hold, except that now a cell is sandwiched between two balanced cuts, which can have arbitrary orientation, instead of the lines ℓ_i^- and ℓ_i^+.

So now we assume that we reach a leaf $\mathcal{C}_i$, which does not conflict with any valley. This cell $\mathcal{C}_i$ must be convex. As valleys are the only reflex edges of $\mathcal{T}$, its restriction $\hat{\mathcal{C}}_i$ above $\mathcal{C}_i$ is convex. Hence, it is the lower envelope of the supporting planes of its faces. These faces are obtained in $O(c_i)$ time from the face lists, and the lower envelope can be computed in $O(c_i \log c_i)$ time algorithm using any optimal 3D convex hull algorithm. We project $\hat{\mathcal{C}}_i$ onto the xy-plane and we obtain the restriction $\mathcal{S}'_i$ of $\mathcal{S}'$ to $\mathcal{C}_i$.

The analysis is similar as for the first stage, except that now the valleys play the role of the vertices of the motorcycle graphs. Our balanced cuts ensure that after each subdivision, the number of conflicting valleys drops by a factor at least 3/2, so the depth of recursion is still $O(\log r)$. Theorem 1 follows.

Acknowledgments. We thank the anonymous referees for their helpful comments.

References

1. Aichholzer, O., Alberts, D., Aurenhammer, F., Gärtner, B.: A novel type of skeleton for polygons. Journal of Universal Computer Science 1(12), 752–761 (1995)
2. Barequet, G., Goodrich, M., Levi-Steiner, A., Steiner, D.: Straight-skeleton based contour interpolation. In: Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 119–127 (2003)

3. Berg, M.D., Cheong, O., Kreveld, M.V., Overmars, M.: Computational Geometry: Algorithms and Applications. Springer (2008)
4. Bowers, J.: Computing the straight skeleton of a simple polygon from its motorcycle graph in deterministic O(n log n) time. CoRR abs/1405.6260 (2014)
5. Cheng, S.W., Mencel, L., Vigneron, A.: A faster algorithm for computing straight skeletons. CoRR abs/1405.4691 (2014)
6. Cheng, S.W., Vigneron, A.: Motorcycle graphs and straight skeletons. Algorithmica 47(2), 159–182 (2007)
7. Chin, F., Snoeyink, J., Wang, C.A.: Finding the medial axis of a simple polygon in linear time. Discrete and Computational Geometry 21(3), 405–420 (1999)
8. Cloppet, F., Oliva, J., Stamon, G.: Angular bisector network, a simplified generalized voronoi diagram: Application to processing complex intersections in biomedical images. IEEE Transactions on Pattern Analysis and Machine Intelligence 22(1), 120–128 (2000)
9. Coquillart, S., Oliva, J., Perrin, M.: 3d reconstruction of complex polyhedral shapes from contours using a simplified generalized voronoi diagram. Computer Graphics Forum 15(3), 397–408 (1996)
10. Day, A., Laycock, R.: Automatically generating large urban environments based on the footprint data of buildings. In: Proceedings of the 8th ACM Symposium on Solid Modeling and Applications, pp. 346–351 (2003)
11. Demaine, E.D., Demaine, M.L., Lubiw, A.: Folding and cutting paper. In: Akiyama, J., Kano, M., Urabe, M. (eds.) JCDCG 1998. LNCS, vol. 1763, pp. 104–118. Springer, Heidelberg (2000)
12. Demaine, E.D., Demaine, M.L., Lubiw, A.: Folding and one straight cut suffice. In: Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 891–892 (1999)
13. Demaine, E.D., Demaine, M.L., Mitchell, J.S.B.: Folding flat silhouettes and wrapping polyhedral packages: New results in computational origami. In: Proceedings of the 15th Annual ACM Symposium on Computational Geometry, pp. 105–114 (1999)
14. Eppstein, D., Erickson, J.: Raising roofs, crashing cycles, and playing pool: Applications of a data structure for finding pairwise interactions. Discrete and Computational Geometry 22(4), 569–592 (1999)
15. Felkel, P., Obdržálek, Š.: Straight skeleton implementation. In: Proceedings of the 14th Spring Conference on Computer Graphics, pp. 210–218 (1998)
16. Held, M., Huber, S.: Theoretical and practical results on straight skeletons of planar straight-line graphs. In: Proceedings of the 27th Symposium on Computational Geometry, pp. 171–178 (2011)
17. Held, M., Huber, S.: A fast straight-skeleton algorithm based on generalized motorcycle graphs. International Journal of Computational Geometry and Applications 22(5), 471–498 (2012)
18. Hershberger, J.: Finding the upper envelope of n line segments in O(n log n) time. Information Processing Letters 33(4), 169–174 (1989)
19. Kelly, T., Wonka, P.: Interactive architectural modeling with procedural extrusions. ACM Transactions on Graphics 30(2), 14:1–14:15 (2011)
20. von Peschka, G.: Kotirte Ebenen: Kotirte Projektionen und deren Anwendung. Buschak and Irrgang, Vorträge (1877)
21. Vigneron, A., Yan, L.: A faster algorithm for computing motorcycle graphs. In: Proceedings of the 29th Symposium on Computational Geometry, pp. 17–26 (2013)

Optimal Time-Space Tradeoff for the 2D Convex-Hull Problem

Omar Darwish[1] and Amr Elmasry[2]

[1] Max-Planck Institute for Informatics, Saarbrücken, Germany
[2] Department of Computer Engineering and Systems, Alexandria University, Egypt

Abstract. We revisit the read-only random-access model, in which the input array is read-only and a limited amount of workspace is allowed. Given a set of N two-dimensional points in a read-only input array and $\Theta(S)$ bits of extra workspace (where $S \geq \lg N$), we present an algorithm that runs in $O(N^2/S + N \lg S)$ time for constructing the convex hull formed by the given points. Following a lower bound for sorting, our algorithm is asymptotically optimal with respect to the read-only random-access model. Of independent interest, we introduce a space-efficient data structure that we call the augmented memory-adjustable navigation pile. We expect this data structure to be a useful tool when designing other space-efficient algorithms.

1 Introduction

Upon appearance of new intelligent terminals such as iPads and iPhones, we now have new different environments for computing. Even though memory became drastically cheaper than before, input data sizes are growing more rapidly. Hence, algorithms designed for using a limited workspace are really requested. Algorithm design with memory constraints has been studied for many years under the name of log-space algorithms [13]. Although a great number of papers have been published in this direction, main focus has been put on purely theoretical computational complexity issues. Our starting point is somewhat different.

Memory constraints for log-space algorithms are too severe for practical use; what is desirable is to design a faster algorithm for a given amount of workspace. An important aspect that is extensively considered is the time-space tradeoff. Munro and Paterson introduced the *multi-pass streaming* model [12] (called the *tape-input* model in [7]). In this model, they assume that the input is stored on a read-only sequentially-accessible media. They basically accounted for the number of passes an algorithm makes over the input as a measure. The model that we consider in this paper is the *read-only random-access* model (called the *register-input* model in [7]). In this model, the input is assumed to be stored on a read-only randomly-accessible media, and arithmetic operations on operands that fit in a computer word are assumed to take constant time each. For this model, it is more appropriate to account for the number of operations the algorithm performs. For both models, we assume that the output is just reported to an output media, and that the algorithm is allowed to use a limited amount of extra workspace. A survey of time-space tradeoffs is given by Borodin [4]. An introduction to the area is given by Savage [17].

Pagter and Rauhe gave an asymptotically-optimal algorithm for sorting N elements [14] in the read-only random-access model, which runs in $O(N^2/S + N \lg S)$ time using workspace of $\Theta(S)$ bits (where $S \geq \lg N$). A modified simpler version of this sorting algorithm is given in [1]. Beame [3] has established a lower bound of $\Omega(N^2)$ for the time-space product for the sorting problem in the stronger branching-program model of computation. The asymptotic complexity of sorting is thus settled for this model.

Chan and Chen [5] gave an algorithm that runs in $O((N^2/S) \cdot \lg N + N \lg S)$ time using workspace of $\Theta(S)$ bits (where $S \geq \lg N$) to construct the convex hull of N two-dimensional points stored on a read-only media. They assumed that the coordinates of the points fit in $O(\lg N)$ bits each, and that arithmetic operations can be performed on $O(\lg N)$-bit words in constant time each. The output is a list of the points on the upper/lower hull in clockwise order. The Chan-Chen algorithm is optimal with respect to the multi-pass streaming model [5], but not with respect to the read-only random-access model.

In this paper we revisit the read-only random-access model. We improve the Chan-Chen bound by introducing a convex-hull algorithm that runs in $O(N^2/S + N \lg S)$ time. Beame's lower bound for sorting [3] naturally applies to the convex-hull problem. Since the running time of our convex-hull algorithm matches the lower bound for sorting, we thus conclude that our algorithm is asymptotically optimal. Obviously, using the random-access capabilities is the reason for the possible improvement. A related distinction between the two models was given in [6] concerning the selection problem.

We expect our techniques to be as well useful when handling other problems in a space-efficient manner. Our basic idea is to partition the input according to an ordering criteria into subsets of adjustable sizes, and perform the algorithm in rounds. In each round, we produce one subset in sequence using our space-efficient priority-queue-like structure. We maintain two filters (x-coordinate values) that enclose the members of the subset. The whole input array may be used to decide the outcome of each round, which is produced once settled.

On a related matter, other space-efficient algorithms for constructing convex hulls exist for the cases when:

– the input points are already sorted by their x-coordinate values [5],
– the running time is in terms of the output size (i.e. output sensitive) [5],
– the input points are in three dimensions [5], and when
– the input is a simple polygon [2].

In Section 2 we review a couple of background tools needed by our algorithm. The main structure that we rely on is the memory-adjustable navigation pile that was partially introduced in [1]. In Section 3 we augment the memory-adjustable navigation pile with extra information to allow more flexibility for handling only a selected subset of the input at a time. In Section 4 we describe the details of our space-efficient convex-hull algorithm. The paper is concluded in Section 5.

2 Background

In this section we describe the main ideas of the Chan-Chen convex-hull algorithm [5]. We also review two of the basic tools that we use in our algorithm: the memory-adjustable navigation piles and the rank-select data structures.

2.1 The Chan-Chen Algorithm

Assume that $\Theta(S)$ bits of workspace are available and that the input N points are stored on a read-only media. Let $S_a = \Theta(S/\lg N)$. The number of points that can possibly be stored in the working storage is $O(S_a)$. The algorithm performs $\lceil N/S_a \rceil$ passes, where in each pass it handles the S_a points with the next smallest x-coordinate values; these points form a vertical slab σ. In each pass, the algorithm starts with a known hull vertex v and computes the part of the upper/lower hull among the points of σ from point v up to and including the hull edge crossing the right wall of σ.

For finding the S_a points with the next smallest x-coordinate values among the remaining points P, the algorithm uses a space of $2S_a$ entries and maintains in the first half the S_a points with the smallest x-coordinate values among the points in P examined so far. Each time, the second half is refilled with another S_a points from P, the median of the $2S_a$ x-coordinate values is found, and the $2S_a$ points are partitioned around this median. This is repeated until all the points of P are examined, leaving the S_a points with the smallest x-coordinate values. The upper/lower convex chain of these points is then constructed using any of the known $O(S_a \lg S_a)$ convex-hull algorithms [15]. The Chan-Chen algorithm eliminates the points that are not on the hull by traversing the tentative chain in reverse order, the points with the larger x-coordinate values first. This procedure is done through finding the hull edge crossing the right wall of σ by performing a pass over the remaining points (those to the right of σ). Suppose a point p from the remaining points is currently being inspected, by imitating Graham-scan algorithm [15], the point p is tentatively added to the chain if it is above the tentative hull edge found so far crossing the right wall of σ. Also, adding p to the chain might require removing some points from the chain in Graham-scan fashion. After the end of the pass, the leftover points on the chain are indeed on the convex hull and are accordingly reported.

Finding the next S_a points with the smallest values requires $O(N)$ time, building a convex chain for S_a points requires $O(S_a \lg S_a)$ time, and pruning the chain from points not on the hull requires $O(N)$ time. Since there are $O(N/S_a)$ passes, the algorithm runs in $O(N/S_a \cdot (N + S_a \lg S_a)) = O((N^2/S) \cdot \lg N + N \lg S)$ time. More details about the Chan-Chen algorithm are given in [5].

2.2 Memory-Adjustable Navigation Piles

A *(minimum) navigation pile* [9] is a data structure that is a compact representation of a tournament tree [10] that uses $\Theta(N)$ bits. Like a priority queue, it supports the operations: *find-min*, *insert*, and *extract*. Alternatively, a *maximum navigation pile* can be defined to support the operation *find-max* instead of *find-min*. In addition to the N elements in the read-only input, the memory-adjustable navigation pile described in [1] uses a workspace of $\Theta(S)$ bits, where $S \geq \lg N$. Based on the following assumptions, it supports *find-min* and *insert* in $O(1)$ worst-case time and *extract* in $O(N/S + \lg S)$ worst-case time:

1. The elements are extracted in a monotonic (increasing) fashion. Consequently, a boundary value is maintained and only larger elements are alive. (We label an element as being *alive* if it inserted but has not been extracted from the priority-queue structure.)
2. The elements are inserted into the data structure sequentially from the input array, but insertions can be intermixed with extractions.

Structure. The input array is divided into S contiguous buckets each containing $\lceil N/S \rceil$ elements, except for the last bucket that may contain less. We assume that S is a power of 2, otherwise we can set it to $2^{\lceil \lg S \rceil}$. A complete binary tree is built above these buckets, where each leaf covers two buckets. A branch node covers the buckets that are covered by the leaves in the subtree rooted at this branch node. In other words, assuming the leaves have height 1, a node at height $h \in \{1, 2, \ldots, \lg S\}$ covers 2^h buckets. For every node at height h, to specify the index of the bucket containing the smallest alive element among the buckets covered by this node, h bits are stored. From the perspective of the nodes at height h, the buckets are virtually divided each into 2^h partitions of contiguous elements that are called *quantiles*. The quantiles of a bucket associated with a node at height h contain $\lceil N/(S \cdot 2^h) \rceil$ elements each, except for the last one that may contain less. Another h bits are kept in every node at height h, indicating the index of the quantile containing the smallest alive element among the quantiles of the bucket that contains this element. The $2h$ bits stored at a node of height h, together with the position of the node, encode the index of a quantile within the input array. We call this quantile the *active quantile* corresponding to the node. If $2h > \lceil \lg N \rceil$, only $\lceil \lg N \rceil$ bits are stored in that node. Summing up the number of bits for all nodes, the total space complexity used is $\sum_{h=1}^{\lg S} (S/2^h) \cdot \min(2h, \lceil \lg N \rceil) = \Theta(S)$ bits. More details can be found in [1].

Operations. To support the *find-min* operation in $O(1)$ time, we keep a separate pointer to the minimum alive element in the navigation pile. This minimum pointer may be updated with every *insert* and *extract* operation.

Concerning the *extract* operation, we get the bucket containing the extracted element using its array index. We find the new minimum of the element's bucket by scanning the bucket, then update the branch nodes covering this bucket along the path towards the root. Every branch node will refer to the smaller of the two elements referenced by its two child nodes. This process will require scanning these nodes' quantiles. During the process, the bucket and quantile bits may be updated in every node along this path. First, we need to know how to access the quantile of a branch node in constant time. All navigation bits are stored in a bit vector in breadth-first order. Hence, we can easily calculate the position where the navigation bits of a certain node are stored. We get the index of the desired bucket using the first h navigation bits, then use the second h bits for accessing the index of the desired quantile of this bucket in constant time. After getting to this position, we can access then scan the desired quantile. For a node of height h, the size of the quantile is at most $\lceil N/(S \cdot 2^h) \rceil$. By summing on this formula over the updating path, the time complexity of *extract* is $O(N/S + \lg S)$.

We skip the details of how to implement *insert* in $O(1)$ time, as we do not need the constant-time bound for our construction. We refer the reader to [1].

We show next how to possibly get rid of the two assumptions: Concerning the first assumption—that the extractions are monotonic—the assumption is imposed so that one can easily check whether a given element is alive or not. To get rid of this assumption, one solution is to keep a bit vector of size N (one bit per element) to indicate whether the corresponding element is alive or not. However, this would increase the space complexity to $\Theta(N)$ bits instead of $\Theta(S)$ bits. As long as we allow $\Theta(N)$ bits, we can thereby get rid of the second assumption—that elements are inserted into the navigation pile sequentially—allowing insertions to be from arbitrary entries of the input array. In this case, the time complexity of the *insert* operation will be $O(N/S + \lg S)$ instead of $O(1)$, as we then have to fix the navigation bits for the nodes covering the inserted element's bucket up to the root (this update is needed only if the new element turns out to be the bucket's new minimum).

2.3 Bit Vectors with *rank* and *select* Support

Given a bit vector V, consider the following operations:

- $V.access(i)$: Returns $V[i]$, the bit at index i.
- $V.rank(i)$: Returns the number of 1-bits among the bits $V[0], V[1], \ldots, V[i]$.
- $V.select(j)$: Returns the index of the j-th 1-bit, i.e., if $V.select(j) = i$, this means that $V[i] = 1$ and $V.rank(i) = j$.

The operations $V.rank0(i)$ and $V.select0(j)$ are similarly defined considering the 0-bits instead of the 1-bits.

What we need is a data structure that supports the aforementioned operations on a bit vector of size N. Our requirements here are that each operation should have $O(1)$ worst-case time, the extra workspace used should be $O(N)$ bits, and building the data structure should be done in $O(N)$ time.

The problem of supporting the previous operations has been addressed in several references. Most of the solutions presented for this problem rely on the idea of dividing the bit vector into blocks then calculating the value of *rank* and *select* for some specific positions, this would help finally in calculating the values for other positions quickly. All solutions, in addition, rely on look-up tables. The reader can refer to [8,11,16] for a couple of solutions to this problem.

3 Augmenting the Navigation Pile

As stated in the previous section, getting rid of the assumption of monotonic extractions, we would allocate one bit per array entry indicating whether the corresponding element is alive or not. To optimize the extra workspace used, our solution is to work with a subset of the input constituting at most S elements at a time. In more details, within each round we shall have two filter elements, and only elements whose values are between these filters are to be inserted or extracted from the navigation pile. We refer to these S elements as the *candidate* elements. Note that the candidates need not be contiguous in the input array. Throughout the round, we use a vector of S bits, one bit per candidate, to indicate whether each of these candidates is still alive or not. Subsequently, we need to map the indices of the array elements to indices in the range $[1 \cdot\cdot S]$. To be able to do that, the memory-adjustable navigation pile needs to be augmented with additional information. We refer to the memory-adjustable navigation pile explained in the previous section as the *unaugmented navigation pile* to distinguish it from the augmented version described below. We shall first introduce the additional structures augmented to the navigation pile, then explain how to update and utilize these structures by the operations. A schematic view illustrating the structure of an augmented navigation pile is given in Figure 1.

Additional Structures. We augment the piles with the following information.

- *alive*. An S-bit vector used to indicate whether each candidate is currently alive or not. The order of the candidates in this vector is identical to their order in the input array. The *alive* vector is dynamically updated by every *insert* and *extract* operation.
- *start*. An S-bit vector that corresponds to the same elements, in the same order, as *alive*. This vector is used to indicate whether each candidate is the first, among other candidates, of an active quantile or not. Recall that a quantile is active if it has the minimum element among those covered by a node. (Note that a candidate may simultaneously be the first of more than one active quantile.) The *start* vector is also dynamic, as it might change by every *insert* and *extract* operation. Every bucket will possibly map to a part of *alive* and *start*, which obviously has at most $\lceil N/S \rceil$ entries. We refer to the part of *start* corresponding to the u-th bucket as $\mathit{start.part}(u)$.
- *count*. A static bit vector that stores the number of candidates contained in each bucket. We encode these counts in unary, using a 0-bit to mark the border between every two consecutive buckets. Since we are dealing with at most S candidates, the vector contains at most S ones; and since we have exactly S buckets, it contains $S - 1$ zeros. The *count* vector should efficiently support *rank* and *select* queries. The vector and the accompanying rank-select structures thus consume $\Theta(S)$ bits. The objective is to efficiently locate the first entry of a given bucket in *alive*. Assume the first bucket has index 0. Let $u > 0$ be the index of the bucket whose first entry in *alive* is to be located. Then, $t = \mathit{count.select}0(u)$ is the index of the last 0-bit preceding

the u-th bucket in *count*. It follows that $t-u+1$ is the number of candidates lying in the buckets $0, 1, \ldots, u-1$, which precede the first entry of the u-th bucket in *alive* and *start*. The size (number of entries) of $start.part(u)$ is $count.select0(u+1) - count.select0(u) - 1$.

– For every node at height h, referring to bucket u and quantile q, $start.part(u)$ is divided into 2^h subparts (or less, if its size is less than 2^h). Another h bits will be stored in every node at height h to indicate in which subpart the first candidate of its active quantile lies. We refer to this subpart as $start.part(u, q)$. Since the size of $start.part(u)$ is at most $\lceil N/S \rceil$, the size of $start.part(u, q)$ is upper-bounded by, $\lceil N/(S \cdot 2^h) \rceil$, the size of q.
– For every branch node at height h, referring to bucket u and quantile q, an additional $\lceil \lg h \rceil$ bits will be used. These bits indicate how many candidates among the first entries of active quantiles exist in the same subpart $start.part(u, q)$ before the first entry for a candidate from quantile q, i.e., the number of ones in $start.part(u, q)$ preceding the one representing the first entry for a candidate from q. Let us refer to this number as $start.before(u, q)$. The reason we need only $\lceil \lg h \rceil$ bits to store this information for each node at height h is that only quantiles tied to nodes (whose heights are less than h) on one and only one path from a leaf node to the given node can have their starting entries before the first entry of q in $start.part(u, q)$.

It directly follows, using simple calculations, that the space complexity of augmenting the navigation pile is still $\Theta(S)$ bits.

To keep the time complexity for *extract* and *insert* in $O(N/S + \lg S)$, we need to know in an efficient way if a given candidate that belongs to an active quantile is alive or not. Starting with the index of such a candidate in the input array, we want to find, without altering the desired time bounds, its index in the *alive* vector. Given a node of the navigation pile ν, referring to bucket u and quantile q, the index of the first entry of an element of q in *alive* is to be located. Using the *count* vector and the bits in ν, we can easily locate $start.part(u, q)$, where the entry we are searching for lies. We also get the value $r = start.before(u, q)$ from the bits in ν. As previously stated, the size of $start.part(u, q)$ is at most the size of q. While navigating through ν we shall be scanning quantile q anyhow. A scan of $start.part(u, q)$ would then not alter the worst-case asymptotic time complexity. We scan $start.part(u, q)$ to find the r-th one bit, the index where we find this bit is the index of the first candidate of q in *alive*.

Naturally, we always access the elements of a quantile sequentially. To locate the corresponding elements of a quantile in *alive*, we start at the first entry of the quantile in *alive* as illustrated in the previous paragraph. While scanning the quantile, we repeatedly check the elements one after another. If the next element is a candidate (lying in the range of the filters), we increment the current index to the next entry in *alive* to correspond to this candidate. The same procedure can be applied on buckets. We first get the first entry of the bucket in *alive* using the *count* vector, and then move sequentially on the bucket and on *alive*, incrementing the *alive* index whenever we encounter a candidate in the bucket.

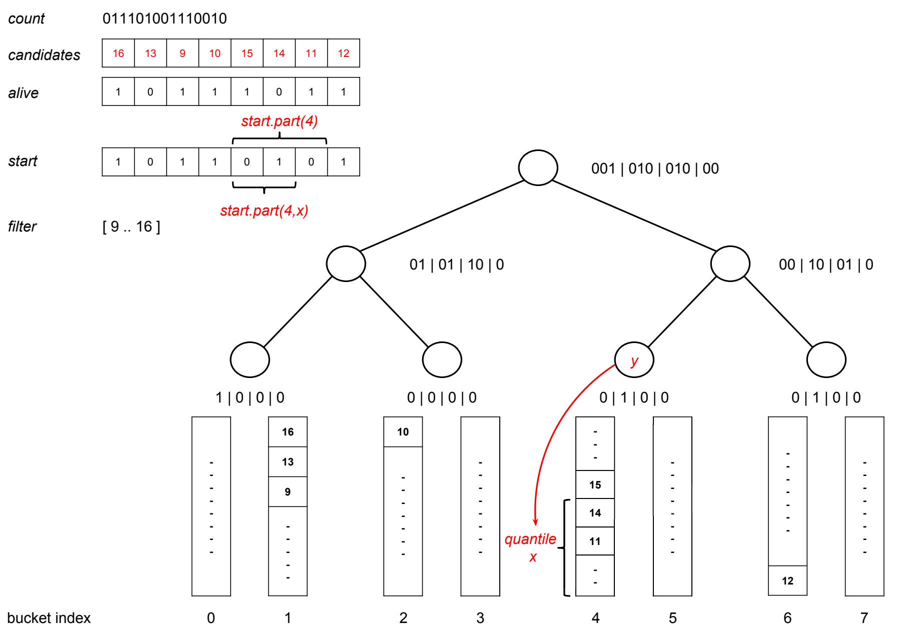


Fig. 1. A snapshot for an augmented navigation pile that has $N = 64$ and $S = 8$. Only elements within the range of the filters are shown. The snapshot is taken considering the given alive vector. The nodes' bits are presented in left-to-right order as follows: bucket index, quantile index, start.part index, and start.before. As an example, we can see that node y refers to the candidate 11 as its alive minimum. Hence, its bucket index is 0 indicating bucket 4. Bucket 4 is partitioned into two quantiles in the view of y, where the candidate 11 lies in the second quantile x. So, the quantile index for y is 1. $start.part(4)$ in $start$ is dedicated to bucket 4. At node y, $start.part(4)$ is partitioned into two parts. Since the candidate 14 represents the first entry of quantile x in $start$, and as this candidate is in the first part of $start.part(4)$, the start.part index of y is set to 0. The part that contains this first candidate of x is referred to as $start.part(4, x)$.

Operations. We next explain how the *insert* and *extract* operations are performed in our augmented navigation pile in $O(N/S + \lg S)$ time per operation.

We first find the bucket in which the element to be inserted/extracted lies; this can be done with simple calculations in constant time once we have the array index of the element. Let the index of this bucket be u. Using the *count* vector, we get the first entry of this bucket in *alive* as explained earlier, and move sequentially on bucket u and *alive*. We can then get the indices of the candidates lying in this bucket within *alive*, and consequently know whether each of these candidates is currently alive or not. After knowing the index of the element to be inserted/extracted in *alive*, we should set the corresponding bit to one/zero. Also, while scanning the bucket, we would know if this element is the minimum element in the bucket or not. If the minimum alive element in bucket u has changed due to this operation, the following updates need to be done.

Let us call the path from the leaf covering bucket u to the root the *updating path*. As in an unaugmented navigation pile, the information in the nodes of the updating path is to be fixed bottom up. (The update stops once we reach a node on the updating path that covers the same minimum element after as before the update.) The update will work as in the unaugmented navigation pile, where we scan the quantiles referenced by the nodes lying on or hanging from the updating path. However, here we also want to know whether each of the candidates in these quantiles is alive or not. Before accessing a quantile, we first get its first entry in *alive*; this can be done as stated previously. Then, we simultaneously scan both the quantile and the corresponding subpart in *alive*.

Next, we show how to update the *start* vector. We explain one simple way to perform this update, but there are other alternatives. Note that we handle the nodes along the updating path in a bottom up manner. Suppose we are to handle a node y, knowing that its child on the updating path has just been handled. If the first index of a candidate in the quantile referenced by y is different from the first index of a candidate in each of the two quantiles referenced by y's two children, we reset the bit for the first entry of the quantile of y in *start* to zero. Note that we do not reset that bit to zero if it is the first entry of the quantile of a child of y in *start*. Alternatively, this bit may be temporarily reset to zero in the previous step and then again set to one within the upcoming step. Assume that the child of y that refers to the quantile that has the smaller element between the two children of y refers to quantile q'. Now the quantile referenced by y should be either the first half or the second half of q'. If it is the first half, then the corresponding entry in *start* must have been already set to one before. Else, we move sequentially on q' and *start* till we reach the first candidate in the second half of q', and set its corresponding entry in *start* to one. The above procedure is repeated for every node on the updating path.

For the nodes along the updating path, we show next how the additional bits in our augmented navigation pile will be updated. Consider a node y that refers to bucket u and quantile q after the update. While handling y, as explained earlier, we are able to know the first entry in *start* for a candidate in quantile q as well as the size of $start.part(u, q)$. Knowing these values, it is easy to update the bits indicating $start.part(u, q)$ in node y. Also, after getting these bits, we loop on $start.part(u, q)$ to count the number of ones in this subpart preceding the first entry for a candidate from q in *start*, and store this count in y.

It is obvious that the update will be done on at most $\lg S$ nodes on the updating path. Also, looping on the quantiles and the corresponding parts of *start* for the nodes on the updating path would sum up to $O(\sum_{i=1}^{\lg S} \lceil N/(S \cdot 2^i) \rceil)$. So the time complexity for the update is in $O(N/S + \lg S)$, as claimed.

The following lemma summarizes the functionality of our data structure.

Lemma 1. *Given a read-only array of N elements, and two filters that enclose at most S of the elements of the input array between their values. After spending $O(N)$ time of preprocessing to build the augmented structure, and using $\Theta(S)$ bits of workspace, insert and extract operations can be applied to any element of the array whose value is between the filters in $O(N/S + \lg S)$ time per operation.*

4 Our Convex-Hull Algorithm

The algorithm uses three navigation piles: a maximum augmented navigation pile that we call *maxPile*, and two minimum unaugmented navigation piles that we call *minPile*1 and *minPile*2. (The algorithm can be implemented using only two navigation piles, but it is easier for the implementation and explanation to use three navigation piles.) Without loss of generality, we assume that the orientation (either maximum or minimum) of the navigation piles is with respect to the points' x-coordinates. The algorithm works as follows:

1. Insert all the N points in both *minPile*1 and *minPile*2.
2. Let the point with the smallest x-coordinate value (leftmost) be v.
3. While v is not the rightmost point do:
   - (a) Extract the minimum S points one by one from *minPile*1 (or until the pile is empty), and keep track of the first and last points (the points with the smallest and largest x-coordinate values) among the S points. These two values will be used as filters and called f_1 and f_2 for *maxPile*. The S points will be the points considered in this iteration.
   - (b) Reinitialize the maximum navigation pile *maxPile* by rebuilding it using the S points whose x-coordinate values are between the filters f_1 and f_2, without actually inserting the points. Reinitialize the *count* vector by scanning the N points bucket by bucket counting the elements whose x-coordinate values are between the filters. Then build the rank-select data structure for the *count* vector.
   - (c) Construct the convex chain for the current S points. This can be done using a space-efficient implementation of Graham-scan algorithm [15], which deploys *maxPile* and *minPile*2 as follows.
     - i. Extract the minimum two points from *minPile*2 and insert both of them in *maxPile*.
     - ii. Repeat $S-2$ times (or until *minPile*2 is empty):
       - A. Extract the next minimum point from *minPile*2, call it p.
       - B. While there are at least two points in *maxPile* and the maximum two among them do not make a right turn with the point p do: repeatedly extract the maximum point from *maxPile*.
       - C. Insert the point p in *maxPile*.

     The alive points in *maxPile* form the convex chain for the S points.
   - (d) Loop on the points from the input array whose x-coordinate values are greater than f_2, or until one point remains in *maxPile*. (The objective is to eliminate the points that are not actually on the hull from the chain formed by the S points in *maxPile*.)
     - i. Let p' be the first point met whose x-coordinate is greater than f_2. Extract points from *maxPile* until the maximum two points make a right turn with p', or until one point remains in *maxPile*.
     - ii. For every point p'' whose x-coordinate value is greater than f_2 do:
       - A. Inspect the current maximum point in *maxPile* and point p', and call the straight line formed by these two points ℓ.

B. If p'' lies above ℓ, start the usual procedure of removing points from *maxPile* until one point remains in *maxPile* or until the maximum and next-maximum points in it make a right turn with the point p'', and set p' to p''.

(e) If p' is null (the current S points are the last ones in the sorted input), extract the maximum point from *maxPile* and set v to this point.
(f) Otherwise, set v to p', then extract and neglect points from *minPile*1 and *minPile*2 until we reach v as the current minimum for both.
(g) Extract all alive points from *maxPile* and report them as points on the convex hull, but in reverse order.

4. Report v as the last point on the convex hull.

By inspecting our algorithm and the Chan-Chen algorithm, we can see that similar algorithmic ideas are used. The main difference is that we are using navigation piles in order to keep the points and deal with them.

We need to prove that our convex-hull algorithm achieves a time complexity of $O(N^2/S + N \lg S)$ and a space complexity of $\Theta(S)$ bits. As for the space complexity, we deploy three navigation piles that are proved to be using only $\Theta(S)$ bits. As for the time complexity, step (1) requires $O(N)$ time to build two unaugmented navigation piles. Obviously, steps (2) and (4) can be done in $O(1)$ time. Now, we are going to analyze step (3). There are $\lceil N/S \rceil$ iterations performed in the while loop of step (3). Concerning the steps in the while loop:

– In step (a), extracting S points requires $O(N + S \lg S)$ time.
– Step (b) needs $O(N)$ time to construct the *count* vector and the data structures for answering *rank* and *select* queries.
– Constructing the upper hull of S points in step (c) is done in $O(N + S \lg S)$ time. Note that S points are extracted from *minPile*2. Each of these points will be inserted and may be extracted later from *maxPile*, but a point can be inserted once and extracted once from *maxPile*, for a total of at most S insertions and S extractions.
– The time complexity of step (d) is $O(N + S \lg S)$ too. We need $O(N)$ to loop on the whole input sequence. Also, we may be extracting points from *maxPile*. Given that the number of alive points in *maxPile* is at most S, then the extractions need at most $O(N + S \lg S)$ time.
– Step (e) requires at most one extraction that requires $O(N/S + \lg S)$ time.
– Step (f) does not affect the time complexity of the algorithm. Here we extract points from *minPile*1 and *minPile*2. Since these piles initially contain the N points and no more insertions are done into them, throughout the algorithm all extractions from these piles require $O(N^2/S + N \lg S)$ time.
– Given that the number of points in *maxPile* is at most S, step (g) will be done in $O(N + S \lg S)$ time.

Except for step (3f), the worst-case time complexity for each iteration of step (3) is $O(N + S \lg S)$. Multiplying this by the $\lceil N/S \rceil$ while loop iterations, the time complexity of our convex-hull algorithm is $O(N^2/S + N \lg S)$ as claimed.

5 Conclusion

We gave an algorithm for constructing the convex hull formed by a given set of points in the plane. The time-space product for our algorithm is asymptotically optimal. The basic tool we used is a data structure that we call the augmented memory-adjustable navigation pile, an important construction in its own right. We expect this data structure to be a useful ingredient in new space-efficient algorithms for problems that employ sorting within their engine.

References

1. Asano, T., Elmasry, A., Katajainen, J.: Priority queues and sorting for read-only data. In: Chan, T.-H.H., Lau, L.C., Trevisan, L. (eds.) TAMC 2013. LNCS, vol. 7876, pp. 32–41. Springer, Heidelberg (2013)
2. Barba, L., Korman, M., Langerman, S., Silveira, R.I., Sadakane, K.: Space-time trade-offs for stack-based algorithms. In: 30th International Symposium on Theoretical Aspects of Computer Science, Schloss-Dagstuhl Leibniz International Proceedings in Informatics, pp. 281–292 (2013)
3. Beame, P.: A general sequential time-space tradeoff for finding unique elements. SIAM Journal on Computing 20, 270–277 (1991)
4. Borodin, A.: Time space tradeoffs (getting closer to the barrier?). In: Ng, K.W., Balasubramanian, N.V., Raghavan, P., Chin, F.Y.L. (eds.) ISAAC 1993. LNCS, vol. 762, pp. 209–220. Springer, Heidelberg (1993)
5. Chan, T.M., Chen, E.Y.: Multi-pass geometric algorithms. Discrete and Computational Geometry 37(1), 79–102 (2007)
6. Elmasry, A., Juhl, D.D., Katajainen, J., Satti, S.R.: Selection from read-only memory with limited workspace. In: Du, D.-Z., Zhang, G. (eds.) COCOON 2013. LNCS, vol. 7936, pp. 147–157. Springer, Heidelberg (2013)
7. Frederickson, G.N.: Upper and lower bounds for time-space trade-offs in sorting and selection. Journal of Computer and System Sciences 34, 19–26 (1987)
8. Jacobson, G.: Space-efficient static trees and graphs. In: 30th IEEE Annual Symposium on Foundations of Computer Science, pp. 549–554 (1989)
9. Katajainen, J., Vitale, F.: Navigation piles with applications to sorting, priority queues, and priority deques. Nordic J. of Computing 10(3), 238–262 (2003)
10. Knuth, D.E.: The Art of Computer Programming: Sorting and Searching, vol, 2nd edn., vol. 3. Addison-Wesley (1998)
11. Munro, J.I.: Tables. In: Chandru, V., Vinay, V. (eds.) FSTTCS 1996. LNCS, vol. 1180, pp. 37–42. Springer, Heidelberg (1996)
12. Munro, J.I., Paterson, M.S.: Selection and sorting with limited storage. Theoretical Computer Science 12, 315–323 (1980)
13. Papadimitriou, C.H.: Computational Complexity. Addison-Wesley (1994)
14. Pagter, J., Rauhe, T.: Optimal time-space trade-offs for sorting. In: 39th IEEE Annual Symposium on Foundations of Computer Science, pp. 264–268 (1998)
15. Preparata, F.P., Shamos, M.I.: Computational Geometry: An introduction. Springer (1985)
16. Raman, R., Raman, V., Satti, S.R.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Transactions on Algorithms 3(4) (2007)
17. Savage, J.E.: Models of Computation. Addison-Wesley (1998)

Cache-Oblivious Persistence[⋆]

Pooya Davoodi[1,⋆⋆], Jeremy T. Fineman[2,⋆⋆⋆], John Iacono[1,†], and Özgür Özkan[1]

[1] New York University
[2] Georgetown University

Abstract. Partial persistence is a general transformation that takes a data structure and allows queries to be executed on any past state of the structure. The cache-oblivious model is the leading model of a modern multi-level memory hierarchy. We present the first general transformation for making cache-oblivious model data structures partially persistent.

1 Introduction

Our result is a general transformation to make a data structure partially persistent in the cache-oblivious model. We first review both the persistence paradigm and the cache-oblivious model before presenting our result.

Persistence. *Persistence* is a fundamental data structuring paradigm whereby operations are allowed not only on the current state of the data structure but also on past states. Being able to efficiently work with the past has become a basic feature of many real world systems, including consumer-oriented software such as Apple's Time Machine, as well as being a fundamental tool for algorithm development. There are several types of persistence which vary in power as to how new versions can be created from existing ones. One can also view persistence as a transformation that extends the operations of an underlying data structure ADT. Data structures, generally speaking, have two types of operations, queries and updates. A query is an operation that does not change the structure, while updates do[1]. A *version* is a snapshot of the data structure at the completion of an update, and updates generate new versions. In all variants of persistence queries are allowed in any version, past or present. The simplest form of persistence is *partial persistence* whereby updates can only be performed on the most recently generated version, called the *present*. The ensuing collection of versions models a linear view of time whereby one can execute operations only in the present and have a read-only view back into the past. A more complicated form of persistence is known as *full* persistence and allows new versions to be generated by performing an

⋆ The full version is available on the arXiv [7].

⋆⋆ Research supported by NSF grant CCF-1018370 and BSF grant 2010437.

⋆⋆⋆ Research supported by NSF grants CCF-1314633 and CCF-1218188.

† Research supported by NSF grant CCF-1018370.

1 Note that, for example, in self-adjusting structures (e.g. [11, 14, 15]) operations considered to be queries are not queries by this definition since rotations are performed to move the searched item to the root, thus changing the structure.

update operation on any of the previous versions; the new version is branched off from the old one and the relationship among the versions form a tree. The most complicated form of persistence is when the underlying data structure allows the combination of two data structures into one. Such operations are called *meld*-type operations, and if meld-type operations are allowed to be performed on any two versions of the data structure, the result is said to be *confluently persistent*. In confluent persistence, the versions form a DAG. In [8], the retroactive model was introduced, which differed from persistence by allowing insertion and deletion of operations in the past, with the results propagated to the present; this is substantially harder than forking off new versions.

Much of the work so far on persistence has been done in the pointer model. The landmark result was when in [9, 13] it was shown that any pointer based structure with constant in-degree can be made fully persistent with no additional asymptotic cost in time and space; the space used was simply the total number of changes to the structure. The first attempt at supporting confluent operations was [10], but the query runtimes could degrade significantly as a result of their transformation. This was largely because by merging a structure with itself, one can obtain an implicit representation of an exponential sized structure in linear steps. In [6] the situation was improved by showing that if a piece of data can only occur once in each structure, then the process of making a structure confluently persistent becomes reasonable.

The cache-oblivious model. The *Disk-Access Model (DAM)* is the classic model of a two-level memory hierarchy; there is a computer with internal memory size M, and an external memory (typically called for historical reasons the *disk*), and data can be moved back and forth between internal memory and disk in blocks of a given size B at unit cost. The underlying premise is that since disk is so much slower than internal memory, counting the number of disk-block transfers, while ignoring all else, is a good model of runtime. A classic data structure in the DAM model is the B-tree [1].

However as the modern computer has evolved, it has become not just a two-level hierarchy but a multi-level hierarchy, ranging from the registers to the main storage, with several levels of cache in between. Each level has a smaller amount of faster memory than the previous one. The most successful attempt to cope with this hierarchy to date has been employing *cache-oblivious algorithms* [12], which are not parameterized by M and B. These algorithms are still analyzed in a two-level memory hierarchy like the DAM, but the values of M and B are used only in the analysis and not known to the algorithm. By doing the analysis for an arbitrary two-level memory hierarchy, it applies automatically to all the levels of a multi-level memory hierarchy.

Cache-oblivious algorithms are analyzed in the ideal-cache model [12], which is the same as the DAM model but for one addition: the system automatically and optimally decides which block to evict from the internal memory when loading a new block. We use the term *cache-oblivious model* to encapsulate both the ideal-cache model and the restriction that the algorithm be cache oblivious. Some DAM-model algorithms are trivially cache oblivious, for example scanning an array of size N has cost $\mathcal{O}(N/B+1)$ in both models. Other structures, such as the B-tree, are parametrized on the value of B and thus cache-oblivious alternatives to the B-tree require completely different methods (e.g. [3, 4]).

Previous persistence results for memory models. In [5] a general method to make a pointer-based DAM model algorithm fully persistent (assuming constant in-degree) was presented where updates incur a $\mathcal{O}(\log B)$ slowdown compared to the ephemeral version. In [2] a partially persistent dictionary for the cache-oblivious model was presented, with a runtime of $\mathcal{O}(\log_B(V+N))$ for all queries and updates, where N is the size of the dictionary and V is the number of updates. Both of these results are based on adapting the methods of the original persistence result for the memory model at hand, and are still tied to the view of a structure as being pointer-based. In contrast, we make no assumptions about a pointer-based structure and do not use any of the techniques from [13].

Persistence in the cache-oblivious model. We present a general transformation to make any cache-oblivious data structure partially persistent; this is the first such general result.

In a cache-oblivious algorithm, from the point of view of the algorithm, the only thing that is manipulated is the lowest level of memory (the disk). The algorithm simply acts as if there is one memory array which is initially empty, and the only thing the algorithm can do is look at and change memory cells. As the *ideal cache assumption* states that the optimal offline paging can be approximated within a constant factor online, the algorithm does not directly do any memory management; it just looks at and changes the memory array. Working directly with the memory array as a whole is a necessary feature of efficient cache-oblivious algorithms, and the overriding design paradigm is to ensure data locality at all levels of granularity. We define the space used by a cache-oblivious algorithm, U, to be the highest address in memory touched so far. The runtime of any operation on a particular state of a data structure is expressed as a function of B and M (only), which are unknown to the algorithm and in fact are different for different levels of the memory hierarchy. In partial persistence, another parameter of interest is the number V of updates performed. This is simply defined as the total number of memory cells changed by the algorithm. So, the goal is to keep a history of all previous states of the data structure, and support queries in the past for any cache-oblivious structure while minimizing any additional space and time needed.

Our results. The most straightforward way to present our results is in the case when the ephemeral data structure's runtime does not assume more than a constant sized memory, and does not benefit from keeping anything in memory between operations. These assumptions often correspond to query-based structures, where blocks that happen to be in memory from previous queries are not of use, and are often reflected in the fact that there is no M in the runtime of the structure. In this case our results are as follows: queries in the present take no additional asymptotic cost; queries in the past and updates incur a $\mathcal{O}(\log_B U)$-factor time overhead over the cost if they were executed with a polynomially smaller block size. In particular, a runtime of $t_q(B)$ in the ephemeral structure will become $\mathcal{O}(t_q(\Theta(B^{(1-\epsilon)/\log 3}))\log_B U)$ for any constant $\epsilon > 0$. Thus, as a simple example, a structure of size $\Theta(N)$ and runtime $\mathcal{O}(\log_B N)$ will support queries in the present in time $\mathcal{O}(\log_B N)$, persistent queries in the past and updates in time $\mathcal{O}(\log_B^2 N)$.

Structures where memory plays a role are more involved to analyze. There are two cases. One is where memory is needed only within an operation, but the data structure's runtime is not dependent on anything being left in memory from one operation to the next operation. In this case, as in the above, queries in the present will be run as asymptotically fast as in the ephemeral using a memory of size a constant factor smaller. For queries in the past and updates, our transformation will have the effect of using the runtime of the ephemeral where the memory size is reduced by a $\mathcal{O}(B^{1-(1-\epsilon)/\log 3}\log_B U)$ and a $\mathcal{O}(B^{1-(1-\epsilon)/\log 3})$ factor, respectively. Thus, in this case, a query of time $t_q(M,B)$ in the ephemeral structure will take time $\mathcal{O}(t_q(\Theta(M),\Theta(B)))$ for a query in the present, $\mathcal{O}(t_q(\Theta(M/B^{1-(1-\epsilon)/\log 3}\log_B U),\Theta(B^{(1-\epsilon)/\log 3}))\log_B U)$ for a query in the past, and $\mathcal{O}(t_q(\Theta(M/B^{1-(1-\epsilon)/\log 3}),\Theta(B^{(1-\epsilon)/\log 3}))\log_B U)$ for an update.

When the runtime of a structure depends on keeping something in memory between operations, the runtimes of the previous paragraph hold for updates and queries in the present. But, for queries in the past, this is not possible. It is impossible to store the memory of all previous versions in memory. For queries in the past, suppose that executing a sequence of queries in the ephemeral takes time $t_q(M,B)$ if memory was initially empty. By treating the sequence as one meta query, the results of the previous paragraph apply, and our structure will execute the sequence in $\mathcal{O}(t_q(\Theta(M/B^{1-(1-\epsilon)/\log 3}\log_B U),\Theta(B^{(1-\epsilon)/\log 3}))\log_B U)$ time. Having $t_q(M,B)$ represent the runtime in the ephemeral assuming memory starts off empty could cause $t_q(M,B)$ to be higher than without this requirement. Short sequences of operations that have zero or sub-constant cost will have the largest impact, while long or expensive sequences will not.

Data structures often have runtimes that are a function of a single variable, commonly denoted by N, representing the size of the structure. This letter makes no appearance in our construction or analysis, however, as it would be too restrictive—bounds that include multiple variables (e.g. graph algorithms) or more complicated competitive or instance-based analysis can all be accommodated in the framework of our result. For the same reason we treat space separately and can freely work with structures with non-linear space or whose space usage is not neatly expressed as a function of one variable.

All of our update times are amortized, as doubling tricks are used in the construction. The amortization cannot be removed easily with standard lazy rebuilding methods, as in the cache-oblivious model one cannot run an algorithm for a certain amount of time as this would imply knowledge of B.

A lower bound on the space usage is the total number V of memory changes that the structure must remember. If the number of updates is sufficiently long compared to ephemeral space usage (in particular, if $V = \Omega(U^{\log 3})$), then our construction has a space usage of $\mathcal{O}(V \log U)$.

Overview of methods. The basic design principle of a cache-oblivious data structure is to try to ensure as much locality as possible in executing operations. So, our structure tries to maintain locality. It does so directly by ensuring that the data of any consecutive range of memory of size w of any past version will be stored in a constant number of consecutive ranges of memory of size $\mathcal{O}(w^{\log 3})$. Our structure is based on viewing changes to memory as points in two dimensions, with one dimension being the memory address, and the other the time the change in memory was made. With this view, our structure is a decomposition of 2-D space-time, with a particular embedding into memory, and routines to update and query.

2 Preliminaries

First we define precisely what a cache-oblivious data structure is, and how its runtime is formulated. A cache-oblivious data structure manipulates a contiguous array of cells in memory by performing read and write operations on the cells of this array, which we denote by A. The array A represents the lowest level of the memory hierarchy. By the ideal cache assumption [12], cache-oblivious algorithms need not (and in fact must not) manage transfers between levels of memory; they simply work with the lowest level.

Ephemeral primitives. The ephemeral data structure can be viewed as simply wishing to interact with lowest level of memory using reads and writes:

- Read(i): returns $A[i]$.
- Write(i, x): sets $A[i] = x$.

Ephemeral cache-oblivious runtime and space usage. Given a sequence of primitive operations, we can define the runtime to execute the sequence in the cache-oblivious model. The runtime of a single primitive operation depends on the sequence of primitives executed before it, and is a function of M and B that returns either zero if the desired element is in a block in memory or one if it is not. This is computed as follows: for a given B, view A as being partitioned into contiguous blocks of size B. Then, compute the most recent M/B blocks that have had a primitive operation performed in them. If the current operation is in one of those M/B blocks, its cost is zero; if it is not, its cost is one.

Space usage is simply defined as the largest memory address touched so far, which we require to be upper bounded by a linear function in the number of Write operations.

Persistence. In order to support partial persistence, the notion of a version number is needed. We let V denote the total number of versions so far, which is defined to be the number of Write operations executed so far. Let A_v denote the state of memory array A after the vth Write operation. Then, supporting partial persistence is simply a matter of supporting one additional primitive, in addition to Read and Write:

- Persistent-Read(v, i): returns $A_v[i]$

3 Data Structure

We view the problem in two dimensions, where an integer grid represents space-time. We say there is a point labeled x at (i, v) if at time v a Write(i, x) was executed, setting $A[i] = x$. Thus the x axis represents space and the y axis represents time; we assume the coordinates increase up and to the right. Let P refer to the set of all points associated with all Write operations. In such a view P is sufficient to answer all Read and Persistent-Read queries. A Read(i) simply returns the label of the highest point in

P with x-coordinate i, and $\mathsf{Persistent\text{-}Read}(v, i)$ returns the label of the highest point in P at or directly below (i, v). We refer to the point (i, v) to be the value associated with memory location i at time v, that is, $A_v[i]$. We denote by V the number of Write operations performed, which is the index of the most recent version.

All of the points lie in a rectangular region bounded by horizontal lines representing time zero at the bottom and the most recent version at the top, and vertical lines representing array location zero (on the left) and the maximum space usage so far (on the right). At a high level, we store a decomposition of this rectangular region into vertically-separated square regions. For each of these square-shaped regions, we use a balanced hierarchical decomposition of the square that stores the points and supports needed queries, which we call the ST-tree. As new points are only added to the top of the structure, only the top square's ST-tree needs to support insertion. As such, the non-top squares' ST-trees are stored in a more space-efficient manner and as new squares are created the old ones are compressed.

3.1 Space-Time Trees

We define the *Space-Time Tree* or ST-tree which is the primary data structure we use to store the point set P and support fast queries on P. This tree is composed of nodes, each of which has an associated space-time rectangle (which we simply call the rectangle of the node). The tree has certain properties:

- Each node at height h in the tree (a leaf is defined to be at height 0) corresponds to a rectangle of width 2^h. This implies all leaves are at the same depth.
- Internal nodes have two or three children. An internal node's rectangle is partitioned by its children's rectangles.
- A leaf is *full* if and only if it contains a point in P. An internal node is full if and only if it has two full children. If an internal node has three children, one must be full.
- Some rectangles may be three sided and open in the upward direction (future time); these are called *open*, while four-sided rectangles are called *closed*. All open rectangles are required to be non-full.

The above conditions imply that a node's rectangle is partitioned by the rectangles of half the width belonging to its children which we call left, right, and if there is a third one, upper. Each node stores:

- Pointers to the three children (left, right and upper) and parent.
- The coordinates of its rectangle.

Additionally, each leaf node (height 0 and thus width 1) stores the following

- The at most 1 point in P that intersects the rectangle of the leaf.
- The results of a Persistent-Read corresponding to the point at the bottom of the rectangle.

Lemma 1. *If a node at height h is full, then its rectangle intersects at least 2^h points in P.*

Proof. Follows directly from the fact that full nodes have at least two full children, full leaves intersect one point in P contained in their rectangle, and the rectangles of all leaves are disjoint and at the same level. □

3.2 Global Data Structure

- The variable U will be stored and will represent the space usage of the data structure rounded up to the next power of two.
- The data structure will store an array of $\lceil \frac{V}{U} \rceil$ ST-trees. The last one is called the *top* tree, and the others are called *bottom* trees. The tree's root's rectangles will partition the grid $[1..U] \times [1..\infty]$. Bottom tree's roots correspond to squares of size U; the j-th bottom root corresponds to square $[1..U] \times [((j-1)U+1)..jU]$. The top tree's rectangle is the remaining three-sided rectangle $[1..U] \times [((\lceil \frac{V}{U} \rceil - 1)U + 1)..\infty]$.
- An array storing a log of all V Write operations which will be used for rebuilding.
- A current memory array, call it C, which is of size U. The entry $C[i]$ contains the value of $A[i]$ at the present, and a pointer to the leaf containing the highest point in P in column i; which also contains the value of $A[i]$ at the present.
- A pointer p to the leaf corresponding to the most recent Persistent-Read.

3.3 How the ST-Trees are Stored

The roots of all ST-trees correspond to rectangles of width U and thus have height $\log U$, and structurally are subgraphs of the complete 3-ary tree of height $\log U$.

The top tree is stored in memory in a brute force way as a complete 3-ary tree of height $\log U$ using a biased Van Emde Boas layout [12]. This layout depends on a constant $0 < \epsilon < 1$ and can be viewed as partitioning a tree of height h into a top tree of height ϵh and $3^{\epsilon h}$ bottom trees of height $h(1-\epsilon)$. Each of the nodes of these trees is then placed recursively into memory. This will waste some space as nodes and subtrees that do not exist in the tree will have space reserved for them in memory. Thus the top ST-tree uses space $U^{\log 3}$.

There will be a level of the van Emde Boas layout that includes the leaves and has size in the range $3^{\log_3 B} = B$ to $3^{(1-\epsilon)\log_3 B} = B^{1-\epsilon}$ nodes. Any path of length k in an ST-tree will be stored in $\mathcal{O}(1 + \frac{k}{\log B})$ blocks. Additionally, we have the following lemma:

Lemma 2. *Any induced binary tree of height h will be stored in $\mathcal{O}(1 + \frac{2^h}{B^{(1-\epsilon)/\log 3}})$ blocks.*

Proof. There is a height h' in the range $\log_3 B$ to $\log_3 B^{1-\epsilon}$ whereby all induced subtrees of nodes at that height will fit into a block. A binary tree of height h will have $2^{h-h'}$ trees of height h', and $2^{h-h'} - 1$ nodes of height greater than h'. Each tree of height h' is stored in memory in B consecutive locations and therefore intersects at most two blocks, thus, even if each node above height h' is in a different block, the total number of blocks the subtree is stored in is $\mathcal{O}(2^{h-h'}) = \mathcal{O}(2^{h-\log_3 B^{1-\epsilon}}) = \mathcal{O}\left(\frac{2^h}{2^{(1-\epsilon)\log_3 B}}\right) = \mathcal{O}\left(\frac{2^h}{B^{(1-\epsilon)\log_3 2}}\right) = \mathcal{O}\left(\frac{2^h}{B^{(1-\epsilon)/\log 3}}\right)$. □

The bottom trees are stored in a more compressed manner; the nodes appear in the same order as if they were in the top tree, but instead of storing all nodes in a complete 3-ary tree, only those nodes that are actually in the bottom tree are stored. Thus the size of the bottom tree is simply the number of nodes in the tree. The facts presented for the top trees and Lemma 2 also hold for the compressed representation.

3.4 Executing Operations

Read(i). We simply return $C[i]$.

Persistent-Read(v, i). The answer is in the tree leaf containing (i, v), the point associated with this operation. We find this leaf by moving p in the obvious way: move the pointer p to parent nodes until you reach a rectangle containing (i, v) or the root of a tree. If you reach a root, set p the root of the appropriate tree, the $\lfloor v/U \rfloor$th one. Then move the pointer p down until you hit a leaf. The answer is in the leaf.

Write(i, x). We call a node a top node if its rectangle is open. We will maintain the invariant that no top node is full. Since Write modifies P by adding a point above all others, this guarantees that the new point will intersect only non-full nodes.

The insertion begins by checking for two special cases. The first is if the memory location i is larger than U, which is an upper bound on the highest memory location written so far. In this case, we apply the standard doubling trick and U is doubled, all the trees are destroyed and re-built by re-executing all Writes which are stored in the log as mentioned in Section 3.2. The current memory array C is also doubled in size.

The other special case is when once every U operations the point associated with the Write is on the $(U + 1)$-th row of the rectangle of the top tree and a new top tree is needed. In this case, the representation of the existing top tree is compressed and copied as a new bottom tree, removing any unused memory locations, and the top tree is reinitialized as a binary tree where the leafs contain values stored in the corresponding cells of C.

Then the main part of the Write operation proceeds as follows.

- Sets $C[i] = x$
- Increments V
- Follows the pointer to the leaf containing (i, V). Note that this is a top node. Add the data to this leaf and mark it full. This means that the point set P now contains the new Write, as it must. However, it is a top node and is full, which violates the previously stated invariant. We then use the following general reconfiguration to preserve that fact that top nodes can not be full.

Reconfiguration. When a node becomes full we mark it as such and proceed according to one of the following two cases. 1) The parent of the node already has three children, in which case the parent also becomes full. We recurse with the parent. 2) The parent of the node has only two children, which implies the parent is still not full. At this point, we close all open rectangles in the subtree of the current node and add a third child to the parent. (These procedures are explained below in more detail.) The rectangle of the new child is on top of the rectangle of the current node.

In other words, when a leaf node becomes full this leads to that leaf node and 0 or more of its ancestors becoming full. We recurse until the highest such ancestor node p and close every open rectangle in its subtree, and add as the third child to the parent of p a sibling node whose open rectangle is on top of p's newly closed rectangle.

Closing rectangles. We close the open rectangles in the subtree of a node by traversing the 2 children of each node that correspond to open rectangles and changing the top side of each open rectangle from ∞ to V.

Adding the third child. In order to add a third child to a node at height $h+1$, we create a complete binary tree of height h whose root becomes the third child. Note that the leafs of this tree contain the answers to the Persistent-Read queries at the corresponding points. We copy this information from C.

Lemma 3. *The amortized number of times a node at height $h+1$ gains a third child following an insertion into its subtree is $\mathcal{O}(\frac{1}{2^h})$.*

Proof. We use a simple potential function where adding a third child on top of a node p at height h has cost 1, and all top nodes at height $l \leq h$ in the subtree of p have a potential of $\frac{1}{2^{h-l}}$ for each full child they have.

Observe that each step during the handling of an insert, a top node with two full children becomes no longer a top node and it is marked as full. Thus, since the potential difference at the level of p matches the cost, the amortized cost at any other level is zero except for at the leaf level where the amortized cost is $\frac{1}{2^h}$. $\square$

4 Analysis

4.1 Space Usage

Lemma 4. *The space usage of a bottom tree is $\mathcal{O}(U \log U)$.*

Proof. The proof is deferred to the full version [7]. $\square$

Lemma 5. *The space usage of the entire structure is $\mathcal{O}(U^{\log 3} + V \log U)$.*

Proof. The top tree uses space $\mathcal{O}(U^{\log 3})$. From Lemma 4, the $\mathcal{O}(V/U)$ bottom ST-trees use space $\mathcal{O}(U \log U)$ each. The log of Write operations is size $\mathcal{O}(V)$. The current memory array is size $\mathcal{O}(U)$. Other global information takes size $\mathcal{O}(1)$. $\square$

4.2 Comments on Memory Allocation

According to the ideal cache assumption, in the cache-oblivious model the runtime is within a constant factor of that which has the optimal movement of blocks in and out of cache. Thus, we can assume a particular active memory management strategy and a runtime of using this strategy is an upper bound of the runtime in the cache-oblivious model. In particular, we can base our assumed memory management strategy on the one an ephemeral structure would use on the same sequence of operations, with particular memory and block sizes.

Globally, we assume the memory M is split into three equal parts, with each part dedicated to the execution of each of the three primitive operations.

4.3 Analysis of Read

Theorem 6. *Suppose a sequence X of* Read *operations takes time $T(M, B)$ to execute ephemerally in the cache-oblivious model on a machine with memory M and block size B and an initially empty memory. Then in our structure, the runtime is $\mathcal{O}(T(M/3, B))$.*

Proof. Since executing Read operations is simply done using the current memory buffer, and there is a memory of size $M/3$ allocated for Reads, this gives the result. □

4.4 Analysis of Persistent-Read

Lemma 7. *Consider the leaves of the ST-tree corresponding to a subarray of ephemeral memory of size w at a fixed time v ($A_v[i \ldots i + w - 1]$ for some i). These leaves and all of their ancestors in the ST-tree are contained in $\mathcal{O}\left(\frac{w}{B^{(1-\epsilon)/\log 3}} + \log_B U\right)$ blocks*

Proof. Let L be the set of all points corresponding to the memory locations $A_v[i \ldots i + w - 1]$. There are two disjoint rectangles corresponding to nodes in the ST-tree with width at most $2w$ such that the two rectangles contain and partition all elements of L. Let x_l and x_r be the nodes corresponding to these two rectangles. Those leaves in subtrees of x_l and x_r corresponding to points in L are the leaves of two binary trees with roots x_l and x_r. Since the height of x_l and x_r are at most $1 + \log w$, by Lemma 2 any binary tree rooted at them will occupy at most $\mathcal{O}\left(1 + \frac{w}{B^{(1-\epsilon)/\log 3}}\right)$ blocks. The nodes on the path between x_l and x_r are stored in $\mathcal{O}(\log_B U)$ blocks. □

Theorem 8. *Suppose a sequence X of* Persistent-Read *operations executed on the same version takes time $T(M, B)$ to execute ephemerally in the cache-oblivious model on a machine with memory M and block size B and an initially empty memory. Then in our structure, the runtime is*

$$\mathcal{O}\left(T\left(\frac{\frac{1}{3}M}{B^{1-\frac{1-\epsilon}{\log 3}}(1 + \log_B U)}, B^{\frac{1-\epsilon}{\log 3}}\right)(1 + \log_B U)\right)$$

Proof. Consider executing X ephemerally with memory $\frac{M}{3(1+\log_B U)B^{1-\frac{1-\epsilon}{\log 3}}}$ and block size $B^{\frac{1-\epsilon}{\log 3}}$. It keeps in memory $\frac{M}{3(1+\log_B U)B^{1-\frac{1-\epsilon}{\log 3}}} \cdot \frac{1}{B^{\frac{1-\epsilon}{\log 3}}} = \frac{M}{3(1+\log_B U)B}$ ephemeral blocks. Each ephemeral block is stored in $\mathcal{O}\left(B^{\frac{1-\epsilon}{\log 3}}/B^{\frac{1-\epsilon}{\log 3}}\right) = \mathcal{O}(1)$ blocks; including the ancestors of the block this becomes $\mathcal{O}(1 + \log_B U)$. Now, by Lemma 7 the memory blocks needed to keep the leaves representing $\frac{M}{3(1+\log_B U)B}$ consecutive memory locations of length B in the persistent structure and their ancestors is $\frac{M}{3(1+\log_B U)B} \cdot (1 + \log_B U) = \frac{M}{3B}$. Thus they can all be stored in the third of memory allocated to Persistent-Read operations ($\frac{M}{3B}$ blocks), and thus moving p to any location in the memory in the ephemeral structure will involve walking it entirely though locations in memory in the persistent structure, and thus will have no cost. Now suppose the ephemeral structure moves one block into memory at unit cost. This could require moving $\mathcal{O}\left(B^{\frac{1-\epsilon}{\log 3}}/B^{\frac{1-\epsilon}{\log 3}} + \log_B U\right) = \mathcal{O}(1 + \log_B U)$ blocks to move the associated leaves and their ancestors in the persistent structure into memory by Lemma 7. Thus, this is the slowdown factor the persistent structure will incur relative to the ephemeral. □

4.5 Analysis of Write

We charge the cost of rebuilding to Write operations. This only increases their cost by a multiplicative constant factor since we double U after $\Omega(U)$ Writes.

Every U Writes, we make the existing top tree a bottom tree and create a new top tree. Three steps involved in this process are closing the rectangles of the top tree, compression of the top tree to a bottom tree, and initializing a new top tree and populating the leafs.

The following lemmas bound the cost of these steps.

Lemma 9. *Copying performed to populate the leafs of a newly created subtree of height h, or closing the open rectangles in the subtree of a node at height h costs $\mathcal{O}\left(\frac{2^h}{B^{(1-\epsilon)/\log 3}}\right)$ block transfers.*

Proof. We only close rectangles that are open and we copy nodes from the array C to leafs of the new subtree which only has open rectangles. If a node is closed all of its children are closed and do not need to be traversed. Given any node, since at most 2 out of its potentially 3 children can be open, the tree we traverse to close rectangles or copy nodes is a binary tree. By Lemma 2, each of these tasks costs $\mathcal{O}\left(\frac{2^h}{B^{(1-\epsilon)/\log 3}}\right)$ block transfers. □

Lemma 10. *Compression of the top tree into a bottom tree costs $\mathcal{O}\left(\frac{U \log U}{B^{(1-\epsilon)/\log 3}}\right)$ block transfers.*

Proof. The top tree is initially a complete binary tree and is stored in $\mathcal{O}\left(\frac{U}{B^{(1-\epsilon)/\log 3}}\right)$ blocks by Lemma 2. We have to account for the additional blocks used to store the elements inserted and nodes created in the top tree after it was created.

By Lemma 3, $\mathcal{O}(\frac{U}{2^k})$ nodes are added to the top tree at height k. When a node at height k is added to the top tree, it is the root of a complete binary search tree and its subtree is stored in $\mathcal{O}\left(2^k/B^{(1-\epsilon)/\log 3}\right)$ blocks by Lemma 2. This implies that added nodes take an additional $\sum_{j=1}^{\log U} \frac{U}{2^j} \cdot \frac{2^j}{B^{(1-\epsilon)/\log 3}} = \mathcal{O}\left(\frac{U \log U}{B^{(1-\epsilon)/\log 3}}\right)$ blocks. □

Theorem 11. *Suppose a sequence of k Write operations takes time $T(M, B)$ to execute ephemerally in the cache-oblivious model on a machine with memory M and block size B and an initially empty memory. Then in our structure, the runtime is*

$$\mathcal{O}\left(T\left(\frac{\frac{1}{3}M}{B^{1-\frac{1-\epsilon}{\log 3}}}, B^{\frac{1-\epsilon}{\log 3}}\right) + k\frac{\log U}{B^{\frac{1-\epsilon}{\log 3}}}\right)$$

Proof. To find the item, the analysis is similar to that of Persistent-Read, except we can use the pointers in C to directly go to the item. This removes the $\log_B U$ terms.

We need to bound the cost of operations performed to maintain the ST-tree. Recall that after each insertion, we go up the tree until we find an ancestor p such that its parent is not full. Then, we create a new sibling node q whose leafs are populated with the copies of values stored in the corresponding top leaf nodes of p, and close the rectangles in the subtree of p. Letting the height of p's parent be h, we refer to this sequence of operations as the expansion of a node at height h.

The creation of the new top tree occurs once every U Writes, and thus the amortized cost per Write is $\mathcal{O}(\log U/B^{(1-\epsilon)/\log 3})$ by Lemma 10. Since the expansion of a node at height h happens because there have been at least 2^h insertions since the node was created, by Lemma 9, the amortized cost of a Write operation is at most $\frac{\log U}{B^{\frac{1-\epsilon}{\log 3}}} + \sum_{j=1}^{\log U} \frac{1}{2^j} \cdot \frac{2^j}{B^{\frac{1-\epsilon}{\log 3}}} = \mathcal{O}\left(\frac{\log U}{B^{\frac{1-\epsilon}{\log 3}}}\right)$. □

We also consider the case when all Write operations are executed in unique memory cells. This is a reasonable assumption when we have update operations involving numerous Write operations where for each memory cell we only need to remember the last Write executed during that update operation.

Theorem 12. *Suppose a sequence of* Write *operations performed on unique cells of A takes time $T(M, B)$ to execute ephemerally in the cache-oblivious model on a machine with memory M and block size B and an initially empty memory. Then in our structure, the runtime is*

$$\mathcal{O}\left(T\left(\frac{\frac{1}{3}M}{B^{1-\frac{1-\epsilon}{\log 3}}\log B}, \frac{B^{\frac{1-\epsilon}{\log 3}}}{\log B}\right)\log_B U\right)$$

Proof. Observe that given M', M'', B', B'' if $B' \leq B''$ and $M'/B' = M''/B''$, then $T(M', B') \geq T(M'', B'')$. Thus, since no two Write operations overlap, we have in the worst case that $k \leq T\left(\frac{\frac{1}{3}M}{B^{1-\frac{1-\epsilon}{\log 3}}\log B}, \frac{B^{\frac{1-\epsilon}{\log 3}}}{\log B}\right) \cdot \frac{B^{\frac{1-\epsilon}{\log 3}}}{\log B}$. The result then follows by substituting k in Theorem 11. □

References

1. Bayer, R., McCreight, E.M.: Organization and Maintenance of Large Ordered Indices. Acta Inf. 1, 173–189 (1972)
2. Bender, M.A., Cole, R., Raman, R.: Exponential Structures for Efficient Cache-Oblivious Algorithms. In: Widmayer, P., Triguero, F., Morales, R., Hennessy, M., Eidenbenz, S., Conejo, R. (eds.) ICALP 2002. LNCS, vol. 2380, p. 195. Springer, Heidelberg (2002)
3. Michael, A., Bender, E.D.: Demaine, and Martin Farach-Colton. Cache-Oblivious B-Trees. SIAM J. Comput. 35(2), 341–358 (2005)
4. Michael, A.: Bender, Ziyang Duan, John Iacono, and Jing Wu. A locality-preserving cache-oblivious dynamic dictionary 53(2), 115–136 (2004)
5. Brodal, G.S., Tsakalidis, K., Sioutas, S., Tsichlas, K.: Fully persistent B-trees. In: Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 602–614 (2012)
6. Collette, S., Iacono, J., Langerman, S.: Confluent persistence revisited. In: Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 593–601 (2012)

7. Davoodi, P., Fineman, J.T., Iacono, J., Özkan, Ö.: Cache-oblivious persistence. CoRR, abs/1402.5492 (2014)
8. Demaine, E.D., Iacono, J., Langerman, S.: Retroactive data structures. ACM Transactions on Algorithms 3(2) (2007)
9. Driscoll, J.R., Sarnak, N., Sleator, D.D., Tarjan, R.E.: Making data structures persistent. J. Comput. Syst. Sci. 38(1), 86–124 (1989)
10. Fiat, A., Kaplan, H.: Making data structures confluently persistent. J. Algorithms 48(1), 16–58 (2003)
11. Fredman, M.L., Sedgewick, R., Sleator, D.D., Tarjan, R.E.: The Pairing Heap: A New Form of Self-Adjusting Heap. Algorithmica 1(1), 111–129 (1986)
12. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. ACM Transactions on Algorithms 8(1), 4 (2012)
13. Sarnak, N., Tarjan, R.E.: Planar Point Location Using Persistent Search Trees. Commun. ACM 29(7), 669–679 (1986)
14. Sleator, D.D., Tarjan, R.E.: Self-Adjusting Binary Search Trees. J. ACM 32(3), 652–686 (1985)
15. Tarjan, R.E.: Efficiency of a Good But Not Linear Set Union Algorithm. J. ACM 22(2), 215–225 (1975)

Lightweight Approximate Selection

Brian C. Dean, Rommel Jalasutram, and Chad Waters

School of Computing, Clemson University, USA

Abstract. Given a relative rank $r \in (0, 1)$ (e.g., $r = 1/2$ refers to the median), we show how to efficiently sample with high probability an element with rank very close to r from any probability distribution that supports efficient sampling (e.g., elements stored in an array). A primary feature of our methods is their elegance and ease of implementation – they can be coded in less space than is occupied by this abstract, and their lightweight footprint makes them ideally suited for highly resource-constrained computing environments. We demonstrate through empirical testing that these methods perform well in practice, and provide a complete theoretical analysis for our methods that offers valuable insight into the performance of a natural class of approximate selection algorithms based on hierarchical random sampling.

1 Introduction

Selection of order statistics has always been a fundamental problem in algorithmic computer science. With recent trends in computing focusing on massive data sets, there has been renewed interest in this problem from the perspective of "in place" and streaming models that allow for only a small amount of auxiliary memory. Moreover, since exact order statistics can be much more challenging to compute in this setting, approximate order statistics are often sought. Specifically, for relative rank $r \in [0, 1]$, an *r-quantile* of a distribution D is given by evaluating the inverse cumulative density function of D at r (e.g., $r = 1/2$ for the median). For an n-element dataset, an r-quantile is the element appearing at index rn when the dataset is sorted. An ε-approximate r-quantile is an r'-quantile for $r' \in [r - \varepsilon, r + \varepsilon]$.

In this paper, we provide a detailed study fully describing both the theoretical and empirical performance of a natural class of approximate selection algorithms. Although non-trivial to analyze mathematically, these algorithms are intuitive and *extremely* easy to implement (requiring less code than even the simple textbook "quickselect" algorithm for exact selection). Furthermore, their lightweight memory footprint makes them ideal for highly-resource-constrained environments (e.g., mobile sensing devices).

Our methods perform approximate selection from any probability distribution D from which we can easily sample. For example, D could be represented explicitly by an array, where we sample from D by randomly sampling from the array. A standard method for approximate selection in this context is to perform exact selection on a small random subset of elements from D, using Chernoff bounds to analyze the quality of approximation. However, our methods are far simpler to code than this, and use much less memory (or equivalently, deliver a much better approximation guarantee using the same amount of memory). We show how to compute an n^{-c}-approximate r-quantile with high probability for any constant $r \in (0, 1)$ in $O(n)$ time and $O(\log n)$ space, where c is a constant that depends on the specific implementation of our approach as well as r; for $r = 1/2$, the most basic manifestation of our approach gives $c \approx 0.369$. The longer we run our algorithm (i.e., the more samples, n, we choose to examine), the better our approximation guarantee becomes.

Our approach is not designed primarily as a "streaming" algorithm per se, although it would still perform well on a randomly-ordered data stream, and we show in the final section through empirical testing that it performs well on a number of large non-random data streams found in practice. A full-fledged streaming algorithm for approximate selection would need to contend with complicating issues like adversarial stream order or unknown stream length, which do cause problems for our methods as they do for many other streaming approaches; we briefly comment on ways to mitigate some of these issues at the end of the paper, at the expense of added complexity. However, our approach is best viewed as a "sampling" algorithm rather than a "streaming" algorithm.

Approximate selection has a number of useful applications in practice, such as estimation of statistical properties of data streams (e.g., packets through a router), database query optimization [SAC$^+$79], computation of association rules for data mining [SA96], load balancing [DNS91], approximating quantities that are reducible to median computation (e.g., circular earthmover distance between a pair of vectors [BLS12]), and construction of equi-depth histograms, which themselves have numerous uses [Ioa03]. From a higher-level algorithmic perspective, fast approximate selection methods can lead to more effective "divide-and-conquer" algorithms and data structures, when used to compute partitioning points between subproblems. They also play a key role in boosting the success probability of randomized algorithms.

1.1 Prior Results

In one of the seminal papers on streaming algorithms, Munro and Paterson [MP78] show that $\Omega(n)$ auxiliary space is necessary for any algorithm that computes the exact median of an adversarially-ordered data stream in a single pass (much less space is necessary with multiple passes). These bounds motivate the study of approximate selection in hopes of achieving substantially less memory usage.

A number of authors have considered approximate selection with one pass over a data stream: Jain and Chlamtac [JC85] and Agrawal and Swami [AS95] provided early results without provable bounds on approximation quality. Provable bounds with increasingly smaller space requirements were given by Alsabti et al. [ARS97], Manku et al. [MRL98], and Greenwald and Khanna [GK01], who ultimately showed that $O(\frac{1}{\varepsilon}\log(\varepsilon n))$ space suffices to compute an ε-approximate r-quantile in a single pass over any n-element data stream. This bound is nearly tight, since Guha and McGregor [GM09] showed that $\Omega(1/\varepsilon)$ space is necessary to find an ε-approximate median in an adversarial stream in one pass. For strong approximation guarantees of the form $\varepsilon = n^{-c}$ with constant c, polylogarithmic space bounds (commonly sought in streaming applications) are therefore still not possible with one pass over an adversarial data stream. Several authors (e.g., [CKMS05,CKMS06]) have derived similar bounds for the related *biased* quantiles problem, where we seek tighter approximation bounds for quantiles closer to the min and max (for example, an ε-approximate r-quantile for $r < \varepsilon$ could be any element of relative rank at most r, which is undesirable).

Polylogarithmic space is possible if we impose some type of randomness assumption on the input, either that we are dealing with a randomly-ordered stream, or (as we assume in this paper) that we can produce a sequence of independent samples drawn from some probability distribution D. For one pass over a randomly-ordered stream that is "mildly" adversarial, Guha and McGregor [GM09] compute an $n^{-1/2+\varepsilon}$-approximate r-quantile with high probability in $O(2^{1/\varepsilon}\text{polylog}\, n)$ space. Standard tail bound analysis shows that this approximation guarantee is the best one can hope to achieve in the model where we are drawing n random samples from a probability distribution. Compared to this result, our methods are much simpler and achieve a stronger space bound of $O(1.254^{1/\varepsilon} \log n)$; however, the two results are not directly comparable since Guha and McGregor consider a more adversarial input model.

If we consider the somewhat different model of selection from a randomly-ordered stream, McGregor and Valiant [MV12] recently obtained an approximation guarantee of $n^{-2/3+\varepsilon}$ with $O(1)$ space, which is optimal if we want only polylogarithmic space usage, since $\Omega(n^{3\varepsilon/2})$ space is needed to achieve an approximation guarantee of $n^{-2/3-\varepsilon}$ [GM09]. However, while impressive from an asymptotic viewpoint, their algorithm is unfortunately far from practical – making reasonable assumptions about hidden constants in the algorithm, it would require inputs of size $n \geq 2^{100}$ for its guarantees to hold[1].

The remainder of this paper is organized as follows. We first discuss our basic algorithm for computing an approximate median of a probability distribution D via hierarchical sampling. Next, we show how a simple modification – using *biased* hierarchical sampling – allows us to generalize this approach to compute an approximate r-quantile for any constant $r \in (0, 1)$. Finally, we discuss implementation considerations that might arise in certain computational environments (e.g., parallel environments, streaming with unknown stream length), and provide the results of empirical testing to show that our methods can deliver good results even when run on non-random data streams in practice.

[1] The algorithm described in [MV12] breaks the length-n input stream into blocks, the first of which has size $(3/4)4^t n^{2/3}$ for $t = \frac{1}{6}\log_2 n$. This first block is then further divided into a polylogarithmic number of sub-blocks each of size $O(3^t n^{2/3}\text{polylog}\, n)$. Assuming that this expression is at least $3^t n^{2/3} \log_2 n$, we need $n \geq 2^{100}$ or else this expression is larger than the size of the first block, preventing subdivision.

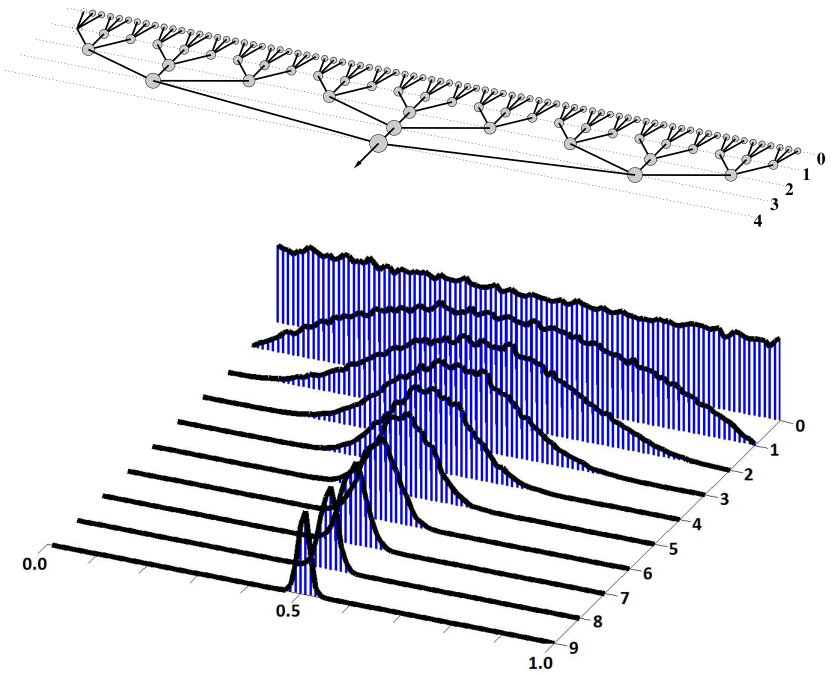


Fig. 1. On top, hierarchical selection along a ternary tree. The original samples from our distribution appear in level 0, each node computes the median of its three children, and the final estimate of the median is emitted from the root. Below, succession of histograms showing how multiple layers of median-of-three operations converge to an approximate median.

2 Approximating the Median

Like many other approaches for approximate selection, our work is based on hierarchical sampling, an approach pioneered initially by Floyd and Rivest [FR75]. The main difference with our approach is that we only sample groups of three elements, instead of larger sets (we show in a few pages how to extend the same approach to odd sets larger than three as well).

Figure 1 illustrates the mechanics of our method in terms of a ternary tree: each leaf (level 0) node computes a random sample from D, and each level $k > 0$ node computes the median of its three children at level $k - 1$. The answer computed by the root is our final estimate of the median of D. To visualize how this converges, the back row (level 0) of the lower part of Figure 1 shows a histogram of the relative ranks of 100,000 random samples taken from D (uniformly distributed over $[0, 1]$, as expected). The next row forward (level 1) shows a histogram of samples that are each taken by choosing the median of three random samples from level 0. We continue in this fashion, generating each sample at level k by taking the median of three random samples from level $k-1$. The result converges quickly to the median of D.

```
function A($x$)
{
    if $x \leq 1$, return sample($D$)
    return med3(A($x/3$), A($x/3$), A($x/3$))
}
```

```
for($i = 1$; $i \leq n$ or stacksize $> 1$; $i$++) {
    push sample($D$)
    for($j = i$; $j$ mod $3 = 0$; $j = j/3$)
        push med3(pop(), pop(), pop())
}
```

Fig. 2. Pseudocode for recursive (left) and iterative (right) algorithms for approximate median computation. The recursive version is invoked by calling $A(n)$, where $O(n)$ is the desired running time.

Pseudocode for a recursive variant of our algorithm appears in Figure 2 on the left, invoked by calling $A(n)$, where $O(n)$ is the desired running time of the algorithm. The larger we set n, the more accurate the final result. The function $\text{med3}(a, b, c)$ computes the median of its three arguments; for example, we might implement it simply as

$$\text{med3}(a, b, c) = a + b + c - \min(a, b, c) - \max(a, b, c).$$

Although the recursive code for our algorithm is extremely short, our preferred implementation in practice is the iterative variant on the right in Figure 2, since it avoids function call overhead. The iterative variant simulates the recursive variant by performing a post-order traversal of our tree. It also runs in $O(n)$ time for a specified value of n of our choosing, and its output is the sole element remaining on the stack at termination. If we are processing a data stream whose size is unknown or not necessarily a power of 3, we would terminate the outer "for" loop upon reaching the end of the stream and return the element at the bottom of the stack as our answer. The total memory usage for both variants of our algorithm is clearly only $O(\log n)$.

2.1 Analysis

Let $F_j(x)$ denote the cumulative distribution function for the relative rank of a level-j sample (i.e., the probability that the relative rank of a level-j sample is at most x). Since the median of three elements is at most x if and only if all three are at most x or two of the three are at most x, we have

$$\begin{aligned} F_0(x) &= x \\ F_1(x) &= x^3 + 3x^2(1 - x) \\ &\vdots \\ F_j(x) &= [F_{j-1}(x)]^3 + 3\,[F_{j-1}(x)]^2\,[1 - F_{j-1}(x)]\,. \end{aligned}$$

In order to obtain a high probability guarantee that a level-j sample is an n^{-c}-approximation to the median, due to symmetry we must have

$$F_j\,(1/2 - 1/n^c) \leq 1/n^k$$

for any constant $k \geq 1$ of our choosing. Using the approximation $f(x-y) \approx f(x) - f'(x)y$, we wish to solve

$$F_j\left(1/2\right) - F_j'\left(1/2\right)/n^c \approx 1/n^k,$$

Since $F_j(1/2) = 1/2$ for all j, this gives

$$F_j'(1/2) \approx n^c(1/2 - 1/n^k) \approx n^c/2,$$

which becomes an equality in the limit as $n \to \infty$. Finally,

$$F_j'(x) = 6F_{j-1}(x)\left[1 - F_{j-1}(x)\right]F_{j-1}'(x),$$

so $F_j'(1/2) = (3/2)F_{j-1}'(1/2)$, and hence $F_j'(1/2) = (3/2)^j$. Plugging in $j = \log_3 n$ if we run our algorithm on n total samples, we have

$$n^{\log_3 \frac{3}{2}} = (3/2)^{\log_3 n} = F_j'(1/2) \approx n^c/2,$$

so as $n \to \infty$, our approximation guarantee is n^{-c} with $c = \log_3(3/2) \approx 0.369$.

By running the algorithm longer, we get more accurate results. As $n \to \infty$, the bound we get by running on $O(n^s)$ samples is $s\log_3(3/2) \approx 0.369s$, obtained by plugging in $j = \log_3 n^s$ above. Hence, we can compute the true median of a length-n array (with high probability, as $n \to \infty$) in $O(n^{\log_{3/2} 3}) \approx O(n^{2.71})$ time. A much faster way to build on our approach to get the true median with $O(\log\log n)$ expected passes over the data stream is to use divide and conquer (e.g., see [MR96]): run our algorithm twice to compute two estimates L and H, repeating until we verify by counting that the true median lies in $[L, H]$; we then recurse on just the elements in this smaller range.

2.2 Higher-Order Generalizations

One can improve the asymptotic approximation guarantee of our approach at the expense of implementation complexity by aggregating along a k-ary tree for larger odd values of $k > 3$. Our analysis above generalizes to these variants, giving an approximation guarantee of

$$\log_k \frac{F_j'(1/2)}{F_{j-1}'(1/2)},$$

where $F_j(x)$ is now unfortunately somewhat more complicated. However, we can still compute this value using a more combinatorial approach. As $\delta \to 0$, observe that $\delta F_j'(1/2)$ is the probability that the rank of the median of k level-$(j-1)$ samples lies in $[1/2 - \delta/2, 1/2 + \delta/2]$. We can compute this probability by:

- Picking a set L of $(k-1)/2$ elements from our k level-$(j-1)$ samples that must all end up being less than $1/2 - \delta/2$ in rank,

- Picking a set H of $(k-1)/2$ elements (disjoint from L) from our k level-$(j-1)$ samples that must all end up being more than $1/2+\delta/2$ in rank, and
- Selecting the final level-$(j-1)$ sample so it has rank $[1/2-\delta/2, 1/2+\delta/2]$.

The number of ways to pick L and H is equal to twice the number of edges in the well-studied *odd graph* $O_{(k+1)/2}$, with nodes corresponding to all subsets of size $(k-1)/2$ from a ground set of k, and $(k+1)\binom{k}{(k-1)/2}/4$ edges, each joining a pair of nodes representing disjoint subsets [Big79]. The probability that all elements in L are at most $1/2-\delta/2$ is $F_{j-1}(1/2-\delta/2)^{|L|}$, which is $(1/2)^{|L|}=(1/2)^{(k-1)/2}$ in the limit as $\delta \to 0$. Likewise, the limiting probability all elements in H are larger than $1/2+\delta/2$ is $(1/2)^{(k-1)/2}$. The probability our single remaining sample lands in $[1/2-\delta/2, 1/2+\delta/2]$ is given by $\delta F'_{j-1}(1/2)$. Multiplying all of these together, we find the asymptotic approximation guarantee for a k-ary tree is n^{-c}, where

$$c = \log_k \left[\frac{k+1}{2^k}\binom{k}{(k-1)/2}\right].$$

Figure 3 shows the asymptotic approximation factors obtained by using successively larger values of k. If we now apply the central binomial coefficient approximation

$$\binom{2x}{x} \geq \frac{4^x}{\sqrt{\pi(x+1/2)}}$$

we find that

$$\binom{k}{(k-1)/2} = \frac{2k}{k+1}\binom{k-1}{(k-1)/2} \geq \frac{2^k}{k+1}\sqrt{\frac{2k}{\pi}}.$$

Setting $\varepsilon = \log_k \sqrt{\pi/2}$, our asymptotic approximation guarantee can be written as

$$c \geq \log_k \sqrt{\frac{2k}{\pi}} = 1/2 - \varepsilon,$$

with space usage

$$O(k \log_k n) = O(\varepsilon \sqrt{\pi/2}^{1/\varepsilon} \log n) = O(1.254^{1/\varepsilon} \log n).$$

These bounds have the same form as the bounds from Guha and McGregor [GM09], albeit with slightly smaller constants (which is reasonable, since the result of [GM09] assumes a more complicated partially adversarial model).

One can also conceive of generalizations of our algorithm with even branching factors $k \geq 2$, but these turn out to be both more complicated and worse-performing than their odd-k counterparts (i.e., performance for $k=6$ is worse than for $k=5$), since an even-sized set has no well-defined median. We omit these from further discussion.

k	3	5	7	9	11	13	15	17
c	0.369	0.391	0.402	0.410	0.415	0.420	0.422	0.426

Fig. 3. Approximation guarantees for larger branching factors k. Each column shows the value of c in the approximation guarantee n^{-c} for a different value of k.

3 Approximating Arbitrary Quantiles

Returning to the basic algorithm with a branching factor of 3, one of the strengths of this method is that it generalizes in a remarkably easy fashion to handle selection at any constant relative rank $r \in (0, 1)$. The only change is that each node makes a biased random decision between its three children instead of always choosing the median. Modified pseudocode for the recursive variant is shown in Figure 4; the iterative variant generalizes similarly.

To analyze this method, we first show how to derive the formulas for α and β. Let us assume we are searching for a rank $r < 1/2$, so $\beta = 0$ and we are focusing only on α (a symmetric argument works for computing β with $r > 1/2$). Based on the biased random decision made by our algorithm, we can re-write the cumulative distribution for a level-j node as

$$\begin{aligned} F_j(x) &= [F_{j-1}(x)]^3 + 3\,[F_{j-1}(x)]^2\,[1 - F_{j-1}(x)] + 3\alpha F_{j-1}(x)\,[1 - F_{j-1}(x)]^2 \\ &= F_{j-1}(x)\left[(3\alpha - 2)(F_{j-1}(x))^2 + (3 - 6\alpha)F_{j-1}(x) + 3\alpha\right] \end{aligned}$$

In order to converge properly, we must have $F_j(x) < F_{j-1}(x)$ for all $x \in [0, r)$ and $1 - F_j(x) < 1 - F_{j-1}(x)$ (hence $F_j(x) > F_{j-1}(x)$) for all $x \in (r, 1]$. Since F is continuous, this means $F_j(r) = F_{j-1}(r)$, so $F_j(r) = F_0(r) = r$ for all j. Our convergence conditions can now be written as: $F_j(x) < F_{j-1}(x)$ for all x such that $F_{j-1}(x) \in [0, r)$, and $1 - F_j(x) < 1 - F_{j-1}(x)$ for all x such that $F_{j-1}(x) \in (r, 1]$. Hence, the quadratic function

$$p(z) = (3\alpha - 2)z^2 + (3 - 6\alpha)z + 3\alpha$$

must satisfy $p(z) < 1$ for $z \in [0, r)$ and $p(z) > 1$ for $z \in (r, 1]$. Since p is continuous, this means $p(r) = 1$. Evaluating $p(z)$ at $z = 0$, we find that $\alpha < 1/3$, so p is concave and satisfies $p(z) = 1$ at two points

$$z = \frac{6\alpha - 3 \pm \sqrt{(3 - 6\alpha)^2 - 4(3\alpha - 2)(3\alpha - 1)}}{6\alpha - 4} = \frac{6\alpha - 3 \pm 1}{6\alpha - 4} = 1, \frac{1 - 3\alpha}{2 - 3\alpha}.$$

Solving $r = \frac{1-3\alpha}{2-3\alpha}$ for α, we obtain the desired formula.

Now let us consider approximation guarantee, assuming again for simplicity that $r < 1/2$. We can use essentially the same approach as in Section 2.1. Starting from

$$F_j\left(r\right) - F_j'\left(r\right)/n^c \approx 1/n^k,$$

this gives

$$F_j'(r) \approx n^c(F_j(r) - 1/n^k) \lesssim n^c(r - 1/n^k) \approx rn^c.$$

```
$\alpha = \max\left(0, \frac{1-2r}{3-3r}\right)$, $\beta = \max\left(0, \frac{2r-1}{3r}\right)$
function A($x$)
{
    if $x \leq 1$, return sample($D$)
    if rand() $< \alpha$ then return min($A(x/3), A(x/3), A(x/3)$)
    if rand() $< \beta$ then return max($A(x/3), A(x/3), A(x/3)$)
    return med3($A(x/3), A(x/3), A(x/3)$)
}
```

Fig. 4. Finding an element of approximate relative rank r. The rand() function returns a random number in the range $[0, 1]$.

On the other hand,

$$\begin{aligned} F_j'(r) &= F_{j-1}'(r)\left[(9\alpha - 6)(F_j(r))^2 + (6 - 12\alpha)F_j(r) + 3\alpha\right] \\ &= F_{j-1}'(r)\left[(9\alpha - 6)r^2 + (6 - 12\alpha)r + 3\alpha\right] \\ &= F_{j-1}'(r)[1 + r]. \end{aligned}$$

Combining as before, we obtain a high probability asymptotic approximation bound of n^{-c} with $c = \log_3(1 + r)$.

We leave as an open question what should be the most natural way to generalize the variants of our algorithm with odd branching factors $k > 3$ so that they compute arbitrary quantiles.

4 Implementation Considerations

Our method is cache-friendly and also parallelizes well. If we divide the total number of samples n into $n = n_1 + \ldots + n_p$ across p processors and run the algorithm (say, the iterative version from Figure 2) on each processor independently, then the results can be easily aggregated after collecting the stacks output by each processor. The ternary representation of n_i tells us the number of level-j outputs on the final stack from process i, so aggregation is essentially equivalent to base-3 addition of the n_i's, with carries corresponding to popping 3 level-j outputs and pushing one level-$(j + 1)$ output.

For real-world inputs that are not sufficiently randomly-ordered, one may wish to apply a technique like the "backing sample" approach of Gibbons et al. [GMP02] in order to effectively randomize the input order. If our input is expected to potentially exhibit a long-term trend (e.g., a linear function plus noise, or in the worst case, a sorted sequence), then we must take special consideration to avoid poor performance. Our algorithm generally only accepts an input size n that is a power of three, so for example if our input is a sorted sequence of length $n = 3^k - 1$, then we will only process the first third of the data, leading to an approximation bound of $1/2 - 1/6 = 1/3$. To improve this, we can use the common idea in the streaming literature of running multiple instances of our algorithm that each make a different guess for the fractional part of $\log_3 n$. If we run t such instances, then instance i ($i = 0 \ldots t-1$) should flip a biased coin for each element of data and only process it with probability

$$p_i = \frac{1}{1 + 2i/t}.$$

It is easy to show that this reduces the approximation bound in the sorted case to $1/(3t)$. Recall that the lower bound from [GM09] states that this is essentially the best any one-pass selection algorithm can do with an adversarial stream, since one must use $\Omega(t)$ space to obtain this quality of approximation; we are only using a logarithmic factor more.

Finally, the variant of our algorithm designed to select quantiles of arbitrary rank $r \in (0, 1)$ suffers from poor convergence as r deviates significantly from $1/2$. To alleviate this and force faster convergence at the outset, suppose $r < 1/2$ and let $b = \lfloor 1/r \rfloor$. We divide our stream into blocks of size b, taking the minimum of each block, and the running our original algorithm on the n/b block mins using relative rank $r' = 1 - (1-r)^b$ (for the case where $r > 1/2$, we use $b = \lfloor 1/(1-r) \rfloor$, block maxes, and relative rank $r' = r^b$). The main idea here is that we are shifting the underlying distribution our algorithm samples from to one requiring a more central relative rank r'; it is straightforward to show that $r' \in [1/4, 3/4]$.

Only trivial modifications to our pseudocode are necessary to implement these changes. For example, let us round n and b to powers of 3. In the code shown in Figure 4, if $x \leq b$ we would just set $\alpha = 1$ (if $r < 1/2$) or $\beta = 1$ (if $r > 1/2$), and otherwise we would use r' in place of r.

5 Computational Experience

To test performance in practice, empirical testing was performed on both random and non-random datasets. As expected, performance on random datasets (the intended model for our algorithm) was strongest, as shown in Figure 5. Each random dataset approximation bound in the figure was obtained by averaging 10 independent executions with different random data. For $r = 0.5$, performance is quite good for large values of n, for example giving an approximation bound ε of roughly two hundredths of a percent for $n = 10^9$. For quantiles other than the median, we see how the blocking technique from the end of the previous section (marked (*) in the figure) leads to dramatic improvement in approximation quality.

To get a sense of how well our approach works when applied in a streaming context to a non-random stream (recall that is not the primarily-intended use case), we tested our algorithm on several large-scale time series datasets: traces.(1-7) are datasets of server metrics over time (e.g., cache usage, I/O load), and EEG is a time series dataset of 10 million samples worth of brain wave data on a single electrode. Performance on these datasets is quite a bit more varied than on truly random datasets, with approximation guarantees ranging from

Instance	ε
random, $n = 10^3$, $r = 0.5$	0.0273
random, $n = 10^6$, $r = 0.5$	0.00333
random, $n = 10^9$, $r = 0.5$	0.000205
traces.1, $n = 1,232,799,308$, $r = 0.5$	0.00442
traces.2, $n = 1,232,799,308$, $r = 0.5$	0.00878
traces.3, $n = 1,232,799,308$, $r = 0.5$	0.0149
traces.4, $n = 574,354,365$, $r = 0.5$	0.0165
traces.5, $n = 1,232,799,308$, $r = 0.5$	0.0133
traces.6, $n = 1,232,799,308$, $r = 0.5$	0.0162
traces.7, $n = 542,674,569$, $r = 0.5$	0.0354
EEG, $n = 10,000,000$, $r = 0.5$	0.0330
random, $n = 10^9$, $r = 0.25$	0.00306
random, $n = 10^9$, $r = 0.25$	0.00173 (*)
random, $n = 10^9$, $r = 0.1$	0.0235
random, $n = 10^9$, $r = 0.1$	0.000715 (*)
random, $n = 10^9$, $r = 0.05$	0.0365
random, $n = 10^9$, $r = 0.05$	0.000274 (*)

Fig. 5. Approximation bounds measured from empirical testing

slightly less than 1% to 3.6% on the worst example. Of course, performance on this data could be improved by using other straightforward heuristics. For example, by using the blocking technique from the previous section to sample the median of size-b blocks as a base case, we can easily bring the relative error of the EEG dataset down to a fraction of a percent for most small values of b. Hence, even when applied to non-random data in a streaming (versus sampling) context, our methods produce reasonable results. When we have the ability to sample, however, the empirical results above demonstrate very strong performance.

Acknowledgements. The authors wish to thank Ben Cousins for helpful discussions on the analysis of our methods, and Linh Ngo for help with the server trace datasets. The first two authors received support from NSF CAREER award CCF-0845593.

References

ARS97. Alsabti, K., Ranka, S., Singh, V.: A one-pass algorithm for accurately estimating quantiles for disk-resident data. In: VLDB, pp. 346–355 (1997)

AS95. Agrawal, R., Swami, A.: A one-pass space-efficient algorithm for finding quantiles. In: COMAD (1995)

Big79. Biggs, N.: Some odd graph theory. Annals of the New York Academy of Sciences 319(1), 71–81 (1979)

BLS12. Brody, J., Liang, H., Sun, X.: Space-efficient approximation scheme for circular earth mover distance. In: Fernández-Baca, D. (ed.) LATIN 2012. LNCS, vol. 7256, pp. 97–108. Springer, Heidelberg (2012)

CKMS05. Cormode, G., Korn, F., Muthukrishnan, S., Srivastava, D.: Space- and time-efficient deterministic algorithms for biased quantiles over data streams. In: IEEE International Conference on Data Engineering (2005)

CKMS06. Cormode, G., Korn, F., Muthukrishnan, S., Srivastava, D.: Space- and time-efficient deterministic algorithms for biased quantiles over data streams. In: PODS, pp. 263–272 (2006)

DNS91. DeWitt, D.J., Naughton, J.F., Schneider, D.A.: Parallel sorting on a shared-nothing architecture using probabilistic splitting. In: PDIS, pp. 280–291 (1991)

FR75. Floyd, R.W., Rivest, R.L.: Expected time bounds for selection. Commun. ACM 18(3), 165–172 (1975)

GK01. Greenwald, M., Khanna, S.: Space-efficient online computation of quantile summaries. In: SIGMOD, pp. 58–66 (2001)

GM09. Guha, S., McGregor, A.: Approximate quantiles and the order of the stream. SIAM J. Comput. 38(5), 2044–2059 (2009)

GMP02. Gibbons, P.B., Matias, Y., Poosala, V.: Fast incremental maintenance of approximate histograms. ACM Trans. Database Syst. 27(3), 261–298 (2002)

Ioa03. Ioannidis, Y.E.: The history of histograms (abridged). In: VLDB, pp. 19–30 (2003)

JC85. Jain, R., Chlamtac, I.: The P2 algorithm for dynamic calculation of quantiles and histograms without storing observations. Commun. ACM 28(10), 1076–1085 (1985)

MP78. Munro, I., Paterson, M.: Selection and sorting with limited storage. In: FOCS, pp. 253–258 (1978)

MR96. Munro, I., Raman, V.: Selection from read-only memory and sorting with minimum data movement. Theor. Comput. Sci. 165(2), 311–323 (1996)

MRL98. Manku, G.S., Rajagopalan, S., Lindsay, B.G.: Approximate medians and other quantiles in one pass and with limited memory. In: SIGMOD, pp. 426–435 (1998)

MV12. McGregor, A., Valiant, P.: The shifting sands algorithm. In: SODA, pp. 453–458 (2012)

SA96. Srikant, R., Agrawal, R.: Mining quantitative association rules in large relational tables. In: SIGMOD, pp. 1–12 (1996)

SAC$^+$79. Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access path selection in a relational database management system. In: SIGMOD, pp. 23–34 (1979)

Robust Distance Queries on Massive Networks

Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck

Microsoft Research, USA
{dadellin,goldberg,tpajor,renatow}@microsoft.com

Abstract. We present a versatile and scalable algorithm for computing exact distances on real-world networks with tens of millions of arcs in real time. Unlike existing approaches, preprocessing and queries are practical on a wide variety of inputs, such as social, communication, sensor, and road networks. We achieve this by providing a unified approach based on the concept of 2-hop labels, improving upon existing methods. In particular, we introduce a fast sampling-based algorithm to order vertices by importance, as well as effective compression techniques.

1 Introduction

Answering point-to-point distance queries in graphs is a fundamental building block for many applications [21] in social networks, search, computational biology, computer networks, and road networks. Dijkstra's algorithm [22] can answer such queries in almost linear time, but this can take several seconds on large graphs. This motivates two-phase algorithms, in which auxiliary data computed during *preprocessing* is used to accelerate on-line *queries*. Although there are practical exact algorithms for road networks [2, 8, 13] and some social and communication graphs [3–5, 17, 18, 23], none is robust on a wide range of inputs.

We propose an exact algorithm that is much more robust to network structure, scales to large networks, and improves (or at least is competitive with) existing specialized solutions. Our method is based on *hierarchical hub labeling* (HHL) [3], a special kind of 2-hop labeling [11]. HHL preprocessing first *orders* vertices by importance, then transforms this ordering into *labels* that enable fast exact shortest-path distance queries (either in RAM or in external memory [1, 17, 20]). Labels can be optionally compressed with no loss in correctness.

While there are fast algorithms to transform an ordering into the corresponding labeling [3, 4], finding a good ordering is challenging. Heuristics that are effective on road networks [3, 13] or on unweighted, undirected small-diameter networks [4] are not robust on other inputs. Compression strategies are similarly specialized to particular networks [4, 10]. We close both gaps by introducing efficient algorithms to find good orders (Section 3) and compress the resulting labels (Section 4) on a wide variety of inputs, including some which no other known method can handle. Our experiments (Section 5) show that our methods are robust, scaling to graphs with tens of millions of arcs. We answer queries to optimality within microseconds using significantly less auxiliary data than previous approaches, effectively widening the range of inputs that can be dealt with efficiently. Details omitted from this extended abstract can be found in the full version [9].

2 Background

The input to the distance query problem is a directed graph $G = (V, A)$ with a positive length function $\ell : A \to \mathbb{Z}_{>0}$. Let $n = |V|$ and $m = |A|$. We denote the length of a shortest path (or the *distance*) from vertex v to vertex w by $\text{dist}(v, w)$. A *distance query* takes a pair of vertices (s, t) as input and outputs $\text{dist}(s, t)$.

A *labeling algorithm* [19] preprocesses the graph to compute a *label* for every vertex such that an s–t query can be answered using only the labels of s and t. The *2-hop labeling* or *hub labeling* (HL) algorithm [11] is a special case with a two-part label $L(v)$ for every vertex v: a *forward label* $L_f(v)$ and a *backward label* $L_b(v)$. (For undirected graphs, each vertex stores a single label that acts as both forward and backward.) The forward label $L_f(v)$ is a sequence of pairs $(w, \text{dist}(v, w))$, with $w \in V$; similarly, $L_b(v)$ has pairs $(u, \text{dist}(u, v))$. Vertices w and u are said to be *hubs* of v. To simplify notation, we often interpret labels as sets of hubs; $v \in L_f(u)$ thus means label $L_f(u)$ contains a pair $(v, \text{dist}(u, v))$. The size $|L(v)|$ of a forward or backward label is its number of hubs. A *labeling* is the set of labels for all $v \in V$ and its size is $\sum_v (|L_f(v)| + |L_b(v)|)$. The *average label size* is the size of the labeling divided by $2n$. Labels must obey the *cover property*: for any s and t, the set $L_f(s) \cap L_b(t)$ must contain at least one hub v that is on the shortest s–t path. We do not assume that shortest paths are unique; to avoid confusion, we mostly refer to (ordered) *pairs* $[u, w]$ instead of paths u–w. We say that a vertex v *covers* (or *hits*) a pair $[u, w]$ if $\text{dist}(u, v) + \text{dist}(v, w) = \text{dist}(u, w)$, i.e., if at least one shortest u–w path contains v.

To find $\text{dist}(s, t)$, an HL query finds the hub $v \in L_f(s) \cap L_b(t)$ that minimizes $\text{dist}(s, v) + \text{dist}(v, t)$. If the entries in each label are sorted by hub ID, this takes linear time by a coordinated sweep over both labels, as in mergesort.

Our focus is on *hierarchical hub labelings* (HHL). Given a labeling, let $v \lesssim w$ if w is a hub of $L(v)$. Abraham et al. [3] define a hub labeling as hierarchical if $\lesssim$ is a partial order. (Intuitively, $v \lesssim w$ if w is "more important" than v.) Natural heuristics for finding labelings produce hierarchical ones [3,4,13,17].

Abraham et al. [3] show that one can compute the smallest HHL consistent with a given ordering *rank*$(\cdot)$ on the vertices in polynomial time. In this *canonical labeling*, vertex v belongs to $L_f(u)$ if and only if there exists w such that v is the highest-ranked vertex that hits $[u, w]$. Similarly, v belongs to $L_b(w)$ if and only if there exists u such that v is the highest-ranked vertex that hits $[u, w]$. Although canonical labelings were originally defined under the assumption that shortest paths are unique [3], the same definition holds when they are not [14]. The algorithms by Abraham et al. [2,3] to compute a labeling from a given order are polynomial, but impractical for most graph classes.

More recently, Akiba et al. [4] proposed the *Pruned Labeling* (PL) algorithm, which efficiently computes a labeling from a given vertex order. (We will use it as a subroutine.) Starting from empty labels, PL processes vertices from most to least important (higher to lower *rank*). The iteration that processes vertex v adds v to all relevant labels. To process v, it runs two pruned versions of Dijkstra's algorithm [22] from v. The first works on the forward graph (out of v) as follows. Before scanning a vertex w (with distance label $d(w)$ within Dijkstra's algorithm), it computes a v–w distance estimate q by performing an HL query with the current partial labels. (If the labels do not intersect, set $q = \infty$.) If $q \leq d(w)$, the $[v, w]$ pair is already covered by previous hubs and the algorithm prunes the search (ignores w). Otherwise (if $q > d(w)$), it adds $(v, \text{dist}(v, w))$ to $L_b(w)$ and scans w as usual. The second Dijkstra computation uses the reverse graph and is pruned similarly; it adds $(v, \text{dist}(w, v))$ to $L_f(w)$ for all scanned vertices w. Note that the number of Dijkstra scans equals the size of the labeling. Also, rather than assuming shortest paths are unique, PL breaks ties on-line in favor of more important (higher-ranked) vertices. Akiba et al. show that PL is correct and produces a minimal labeling (deleting any hub violates the cover property). It is easy to show that it is also hierarchical, and thus canonical.

3 Computing Orderings

Knowing how to efficiently compute a hierarchical labeling from an order, we now consider how to find orders that lead to small labelings. (Recall that any order produces correct labels.) One can sidestep this issue by using the order implied by vertex degrees [4, 17]; using degree as a proxy for importance works well for some unweighted and undirected small-diameter networks, but is not robust. *Contraction Hierarchies* (CH) [3] orders vertices bottom-up, using only local information that is carefully updated as decisions are made. This often leads to small labels [3], but can be costly because the number of updates may be superlinear and the updates themselves may be expensive.

For better results, Abraham et al. [3] propose a greedy top-down algorithm. It finds good labels for a wide range of graph classes, but is too expensive (in both time and space) for large instances. In this section, we recap this *basic algorithm* and then show that using on-line tie breaking leads to even better orders, but with greater preprocessing effort. Finally, we propose a sampling technique that makes the basic algorithm much faster while still finding good solutions.

Basic Algorithm. The basic algorithm [3] defines the order greedily: the i-th highest ranked vertex (hub) is the one that hits the most previously uncovered shortest paths (i.e., not covered by the $i-1$ hubs already picked). To implement this rule efficiently, the basic algorithm starts by building n full shortest path trees, one rooted at each vertex of the graph. The tree T_s rooted at s represents all uncovered shortest paths starting at s. This effectively makes shortest paths unique: the algorithm assumes that only vertices on the s–t path in T_s can hit the pair $[s, t]$. The number of descendants of v in T_s is thus the number of uncovered shortest paths that start at s and contain v. The total number of descendants of v over all trees, denoted by $\sigma(v)$, is the number of shortest paths that would be hit if v were picked as the next most important hub.

Each iteration of the algorithm picks as the next hub the vertex v^* for which $\sigma(v^*)$ is maximum. To prepare for the next iteration, it removes the subtree rooted at v^* from each tree (as the paths they represent are now covered by v^*) and updates the $\sigma(\cdot)$ values of all descendants and ancestors of v^*. This algorithm is *path-greedy*: it maximizes the number of new paths hit in each iteration. Abraham et al. also propose a *label-greedy* variant, which picks the vertex v^* that maximizes the ratio between $\sigma(v^*)$ and the number of labels to which v^* will be added; this leads to slightly better labels. Note that the basic algorithm breaks ties off-line (a-priori) while computing the initial trees; the resulting paths determine not only which hub to select next, but also to which labels this hub is added. Our experiments thus refer to it as OffPG (path-greedy) or OffLG (label-greedy). Both variants run in $O(mn \log n)$ time [3].

Better Tie-Breaking. When shortest paths are far from unique (as in some unweighted small-diameter networks), the basic algorithm underestimates the number of pairs hit by each hub it picks. Since it breaks ties a-priori, it produces bigger labels than needed. For better results, we propose a simple hybrid algorithm: first compute a vertex order using the basic algorithm, then use PL to find the labeling. Since PL breaks ties in favor of paths with the highest maximum vertex rank, this can lead to substantially smaller labels with little overhead. We refer to this algorithm as HybPG (path-greedy) or HybLG (label-greedy).

We also propose an algorithm that breaks ties *on-line* while selecting the order. Although impractical for large instances, it finds the smallest hierarchical labels we are aware of (on moderate-sized inputs). It follows the same approach as the basic algorithm, picking in each iteration the hub v^* (not picked before) that covers the most uncovered pairs $[u, w]$. The challenge is finding v^* efficiently in every iteration: since ties are broken on-line, we must implicitly maintain the numbers of descendants in all shortest path DAGs, which is harder than in trees.

Thus, during initialization, we compute an $n \times n$ distance table between all n vertices and create an $n \times n$ boolean matrix in which entry (u, w) indicates whether the pair $[u, w]$ is already covered by a previously selected hub. All entries $[u, w]$ with finite $\text{dist}(u, w)$ are initially false. For each vertex v, we maintain $\sigma(v)$, the number of new pairs that would be covered if v were selected as the next hub. Initially, this is the total number of pairs $[u, w]$ hit by v. For a fixed v, this value can be found in $O(n^2)$ time: check for each $[u, w]$ if $\text{dist}(u, v) + \text{dist}(v, w) = \text{dist}(u, w)$.

Each iteration of the algorithm is as follows. First, pick a vertex v for which $\sigma(v)$ is maximum (in $O(n)$ time). Then find the set Q of uncovered pairs hit by v (note that $|Q| = \sigma(v)$); this takes $O(n^2)$ time using the distance table and the boolean matrix. For each pair $[u, w] \in Q$, mark $[u, w]$ as covered and add v to both $L_f(u)$ and $L_b(w)$ (if not there already). Finally, update the other σ values: for each pair $[u, w] \in Q$ and vertex $x \in V$, decrease $\sigma(x)$ by one if x hits $[u, w]$. This step takes $O(|Q|n)$ time. Since any pair appears in some Q at most once during the algorithm, the combined size of all Q lists is $O(n^2)$. Altogether, the algorithm runs in $O(n^3)$ worst-case time and $\Theta(n^2)$ space. This *path-greedy* algorithm can be extended to be *label-greedy* with the same bounds. We call these variants OnPG and OnLG, respectively.

Finding Good Orderings Faster. All methods considered so far are impractical for large graphs, since they use $\Omega(n^2)$ space and time. We thus propose an improvement of the path-greedy hybrid algorithm (HybPG) that uses sampling to compute *estimates* $\tilde{\sigma}(\cdot)$ on the $\sigma(\cdot)$ values. The estimates need only be precise enough to distinguish important vertices (those for which $\sigma(\cdot)$ is large) from unimportant ones. We can tolerate fairly large errors on the estimate of unimportant vertices. Intuitively, if $\sigma(v) \gg \sigma(w)$, we want to have $\tilde{\sigma}(v) > \tilde{\sigma}(w)$.

A natural approach is to build $k \ll n$ trees from random roots and set $\tilde{\sigma}(v)$ to be the total number of descendants of v in all trees of the sample [7]. (Sampling paths uniformly would be ideal, but too costly.) Once a vertex v^* is picked (from a priority queue), we update counters as in the basic algorithm, with all descendants of v^* removed from the sampled trees. Unfortunately, when k is small (as required for good performance), such $\tilde{\sigma}(v)$ estimates are only accurate for very important vertices (with many descendants in most trees); as sampled trees get smaller, we have insufficient information to assess less important vertices.

We deal with this by generating more trees (from new roots) as the algorithm progresses. We grow them using Dijkstra's algorithm, but pruning vertices already covered by previously picked hubs (like in PL). Newly added trees thus only contain uncovered paths and get smaller as the algorithm progresses, keeping space and time under control. Since we need partial labels for pruning, we add v^* to all relevant labels (running one PL iteration from v^*) right after v^* is selected as next hub. We balance the work spent growing trees and constructing (adding hubs to) labels. Let c_t be the total number of arcs and hubs touched so far while building new trees (the k original trees are free); define c_l similarly, for operations during label construction. We generate trees from random new roots until either $c_t > c_l$ or the total number of vertices in existing trees exceeds $10kn$. To bound the space usage, we represent small trees as hash tables.

Although the total number of descendants in the sample is a natural estimator for the total over all n trees, its variance is very high. In particular, it overestimates the importance of vertices that are at (or near) the root of a sampled tree [12]. Replacing the sum (or average) by a more robust measure (such as the median) would remedy this, but is costly to maintain as trees (and counters) are updated. We achieve both robustness and speed as follows. Instead of keeping a single counter $\tilde{\sigma}(v)$ for each vertex v, we keep c counters $\tilde{\sigma}_1(v), \tilde{\sigma}_2(v), \ldots, \tilde{\sigma}_c(v)$, for some constant c. Counter $\tilde{\sigma}_i(v)$ is the total number of descendants of v over all trees t_j such that $i = (j \bmod c)$. (Here t_j is the j-th tree in the sample, not the tree rooted at j.) These counters are easy to maintain and allow us to eliminate outliers when evaluating v, for instance by discarding the counter i that maximizes $\sigma_i(v)$ and taking as estimator the average value of the remaining counters. (Intuitively, if v is close to the root of one tree, only one counter will be affected.)

In general, increasing c improves accuracy, but can be costly because the priority of a vertex depends on all its c counters. We found that using $c = 16$ and discarding the two highest counters gives good results with negligible overhead. In case of ties, we prefer vertices maximizing $\tilde{\sigma}(v) = \sum_{i=1}^{c} \tilde{\sigma}_i(v)$. Moreover, we ensure at least c trees are live during the execution. We call this ordering algorithm SamPG. We have no label-greedy variant of this algorithm, as it is unclear how to obtain good estimates on the number of labels a hub is added to.

4 Compression

Representing labels compactly is crucial for large graphs. We first show how to represent distances or IDs with fewer bits without sacrificing query times, then propose a more elaborate technique that exploits similarities across labels, trading higher compression for slower (but still exact and fast enough) queries.

Basic Compression. Recall that a label $L_f(u)$ can be seen as an array of pairs $(v, \text{dist}(u, v))$ sorted by hub ID v. In practice [2], it pays to first represent all hubs, then the corresponding distances (in the same order). Since distances are only read when hubs match, queries have fewer cache misses. We represent distances with as few bits (8, 16, or 32) as needed for the largest distance stored in any label. (For unweighted small-diameter networks, 8 bits are enough [4].)

Less trivially, one can use fewer bits to represent hub IDs. Abraham et al. [2] *rename* the hubs so that IDs 0 to 255 are assigned to the most important (higher-ranked) vertices, and use only 8 bits to represent them (and 32 bits otherwise). On road networks, space is reduced by around 10% (and queries become faster), since many hubs in each label are in this set. For greater effectiveness on more inputs, we propose two improvements: delta representation and advanced reordering.

Delta representation stores hub IDs in difference form. Let the hub IDs in a label be $h_1 < h_2 < h_3 < \dots$ We store h_1 explicitly, but for every $i > 1$ we store $\Delta_i = h_i - h_{i-1} - 1$. A label with hubs `(0 16 29 189 299 446 529)` is thus represented as `(0 15 12 159 109 146 82)`. Because queries always traverse labels in order, we can retrieve h_i as $\Delta_i + h_{i-1} + 1$. Since $\Delta_i < h_i$, this increases the range of entries that can be represented with fewer bits. (In the example above, 8 bits suffice for all entries.) To keep queries simple, we avoid variable-length encoding. Instead, we divide the label into two blocks: we start with 8 bits per entry, and switch to 32 bits when needed.

Our second technique is to rename vertices to increase the number of 8-bit hub entries. We could reorder hubs by rank (as in Abraham et al. [2]) or by frequency, with smaller IDs assigned to hubs that appear in more labels, but we can do even better (by about 10%) with *advanced reordering*. We assign ID 0 to the most frequent vertex and allocate additional IDs (up to $n - 1$) to one vertex at a time. For each vertex v that is yet unassigned, let $s(v)$ be the number of labels in which v could be represented with 8 bits if v were given the smallest available ID. Initially, $s(v)$ is the number of labels containing v, but its value may decrease as the algorithm progresses. Each iteration of our method picks the vertex v with maximum $s(v)$ value and assigns an ID to it. If multiple available IDs are equally good (i.e., realize $s(v)$), we assign v the *maximum* ID among those, saving smaller IDs for other vertices. In particular, the second most frequent vertex could have any ID between 1 and 256 and still be represented as 8 bits, so it gets ID 256.

The main challenge for advanced reordering is efficiently updating the $s(\cdot)$ values. Our lazy implementation keeps a priority queue with estimated $\tilde{s}(\cdot)$ values. Each iteration picks the maximum such element $\tilde{s}(v)$ and computes the actual $s(v)$ value. If the estimate is approximately correct, we assign an ID to v; otherwise, we reinsert v into the queue with $\tilde{s}(v) \leftarrow s(v)$.

Token-Based Compression. We now present a novel scheme to achieve even higher compression. It extends *hub label compression* (HLC) [10], which interprets each label as a tree and represents each unique subtree (which may occur in many labels) only once. We explain HLC first, then our improvements.

HLC represents the hubs of a forward label $L_f(u)$ as a tree rooted at u. For canonical hierarchical labels, the parent of $w \in L_f(u) \setminus \{u\}$ in the tree is the highest-ranked vertex $v \in L_f(u) \setminus \{w\}$ that hits $[u, w]$ (the tree representing $L_b(u)$ is defined analogously). The key insight is that the same subtree often appears in the labels of several different vertices. HLC represents each unique subtree as a *token* consisting of (1) a *root vertex* r; (2) the number k of *child tokens*; (3) a list of k pairs (i, d_i) indicating that the root of the child token with ID i is within distance d_i from r. A token with no children ($k = 0$) is a *trivial token*, and is represented implicitly. Each nontrivial unique token is stored only once. The data structure also maintains an index mapping each vertex v to its two *anchor tokens*, the roots of the trees representing $L_f(v)$ and $L_b(v)$.

An s–t query works in two phases. The first reconstructs the labels $L_f(s)$ and $L_b(t)$ by traversing the corresponding trees in BFS order and aggregating distances appropriately. The second phase finds the vertex $v \in L_f(s) \cap L_b(t)$ that minimizes $\text{dist}(s, v) + \text{dist}(v, t)$. Since the label entries produced by the first phase are not sorted by hub ID, the second phase uses hashing rather than merging [10].

Although HLC compresses road network labelings by an order of magnitude [10], it is much less effective on small-diameter inputs: high-degree vertices are costlier to represent and there are fewer exact matches between subtrees.

To make HLC effective on a wider range of inputs, we now propose *mask tokens*. A mask token t represents a unique subtree, but not directly: it contains the ID of another token t' (its *reference token*), as well as an incidence vector (bitmask) indicating which children of t' should be taken as children of t. Note that both t and t' must have the same root. This avoids the need to represent the same children multiple times. To exploit this further, we use *supertokens*. A supertoken has the same structure as a standard token (with a root and a list of children), but represents the union of several tokens, defined as the union of their children. For each vertex v, we create a supertoken representing the union of *all* standard tokens rooted at v. Subtrees that actually appear in the labeling can be represented as mask tokens using the supertoken as reference.

Since a mask that refers to a supertoken with k children needs k bits, space usage can be large. But most mask entries are zero (original tokens tend to have few children), motivating the use of *mask compression*. We propose a *two-level approach*. Conceptually, we split a k-bit mask into $b = \lceil k/8 \rceil$ *buckets*, each representing up to 8 consecutive bits. For example, a label with $k = 45$ has six 8-bit buckets: bucket 0 refers to bits 0 to 7, bucket 1 to bits 8 to 15, and so on. Only nonempty buckets are stored explicitly: an *index array* indicates which q buckets (with $1 \leq q \leq b$) are nonempty, and is followed by q 8-bit incidence arrays representing the nonempty buckets. The index takes $\lceil \lceil k/8 \rceil /8 \rceil$ bytes.

In general, there will be fewer nonempty buckets if the "1" entries in each bit mask are clustered. Since correctness does not depend on the order in which children appear in a supertoken, we can permute them to make the "1" entries more concentrated. Therefore, for each child x of v, we count the number $c_v(x)$ of standard tokens rooted at v in which x appears, then sort the children of the supertoken rooted at v in decreasing order of $c_v(x)$.

Token-based compression must transform labels into trees, which requires finding parents for all vertices in the label. Delling et al. [10] compute such parents in $O(nM^3)$ time, where M is the maximum label size. We use a much faster (and novel) $O(nM^2)$-time algorithm tailored to hierarchical labels. It augments PL to maintain tentative parent pointers as it goes, using the fact that, by the time a hub is added to a label, its final children are already present.

5 Experiments

We implemented all algorithms in C++ using Visual Studio 2013 with full optimization. All experiments were conducted on a machine with two Intel Xeon E5-2690 CPUs and 384 GiB of DDR3-1066 RAM, running Windows 2008R2 Server. Each CPU has 8 cores (2.90 GHz, 8×64 kiB L1, 8×256 kiB, and 20 MiB L3 cache), but all runs are sequential. We use at most 32 bits for distances.

We test *social networks* (Epinions, Slashdot, Flickr, Hollywood, WikiTalk), *computer networks* (Gnutella, Skitter, MetroSec), *web graphs* (NotreDame, Indo, Indochina, uk2002), *road networks* (ber-t, fla-t, eur-t, eur-d), and *3D triangular meshes* (buddha), available from snap.stanford.edu, webgraph.di.unimi.it, www.dis.uniroma1.it/challenge9, and socialnetworks.mpi-sws.org/datasets.html. We also test unweighted grid graphs with holes from VLSI applications (alue7065; steinlib.zib.de) and grids with obstacles built from *computer games* (FrozenSea, AR0503SR; movingai.com). For the latter we set edge lengths to 408 for axis-aligned moves and 577 for diagonal moves. (Note that $577/408 \approx \sqrt{2}$.) We also test synthetic inputs: square *grids* (grid$i$), *Delaunay triangulations* of random points on the unit square (del$i$), random geometric graphs, often used to model *sensor networks* (rgg$i$) [16], random *preferential attachment* graphs (rba$i$), and random *small-world* networks (rws$i$) [15], with $i = \log n$. Some instances are unweighted, while in others (with suffix -w) edge lengths correspond to Euclidean distances (scaled appropriately and rounded up).

Table 1. Key values for inputs, ordering quality of degree and SamPG, and performance of RXL and CRXL

instance						**degree**		**SamPG**		**RXL**		**CRXL**	
type	name	n	m/n	d	w	prep [s]	lab	prep [s]	lab	[MiB]	[µs]	[MiB]	[µs]
sensor	rgg20	1048576	13.1	○	○	2804	1135.7	977	220.0	806.5	2.0	167.3	23.4
	rgg20-w	1048576	13.1	○	●	52962	5502.7	3608	588.8	3154.3	4.9	436.4	76.1
roads	fla-t	1070376	2.5	○	●	1321	791.8	103	41.4	260.9	0.5	55.0	3.4
	eur-t	18010173	2.3	●	●	–	–	8364	82.4	17202.8	0.8	1589.3	13.3
	eur-d	18010173	2.3	●	●	–	–	18664	163.1	33059.5	1.5	2184.2	32.1
grid	alue7065	34046	3.2	○	○	1	98.2	3	55.9	6.1	0.5	2.8	3.5
	grid20	1048576	4.0	○	○	92	144.8	364	126.6	526.5	1.3	127.0	14.8
triang	buddha	543524	6.0	○	○	119	289.5	122	91.5	179.8	0.9	62.6	9.0
	buddha-w	543524	6.0	○	●	1424	1164.7	678	336.0	952.9	2.9	176.6	41.5
	del20	1048576	6.0	○	○	241	286.8	306	117.5	452.1	1.1	134.1	13.2
	del20-w	1048576	6.0	○	●	4606	1598.9	2449	575.3	3077.1	4.8	426.6	115.6
game	FrozenSea	754304	7.6	○	●	160	241.4	214	92.1	429.3	0.9	133.0	10.9
web	NotreDame	325729	4.5	●	○	4	21.1	17	11.3	25.9	0.1	19.5	0.4
	Indo	1382908	12.0	●	○	253	171.7	241	27.4	217.5	0.4	127.9	1.3
	Indochina	7414866	25.8	●	○	12028	539.8	14824	65.5	3916.5	0.7	1322.9	3.2
	uk2002	18520486	15.8	●	○	–	–	43090	278.5	34140.5	1.8	2533.1	25.2
comp	Gnutella	62586	2.4	●	○	37	240.9	60	157.1	39.4	0.9	17.8	7.4
	Skitter	1696415	13.1	○	○	1905	456.5	2813	273.5	1074.6	2.3	316.7	20.6
	MetrocSec	2250498	19.2	○	○	356	132.0	2276	116.5	592.8	0.8	207.7	3.6
social	Epinions	75888	6.7	●	○	12	94.2	50	91.3	29.2	0.6	13.3	3.6
	Slashdot	82168	10.6	●	○	40	188.3	140	190.7	65.3	1.5	31.2	7.4
	rws17	131072	6.0	○	○	5827	4264.4	9224	3597.7	901.2	27.5	1102.9	327.8
	rba20	1048576	12.0	○	○	8006	1485.6	26238	1541.6	4918.0	11.0	2517.6	131.8
	Hollywood	1139905	98.9	○	○	38412	2921.3	61411	2114.3	5934.3	13.9	2050.0	204.0
	Flickr	1861232	12.2	●	○	3353	423.3	10332	322.4	3093.8	2.5	603.8	17.2
	WikiTalk	2394385	2.1	●	○	281	68.0	999	60.2	625.8	0.5	127.3	2.1

Table 1 summarizes our main results. For each instance, we show its type, average number of vertices (n), average out-degree (m/n), and whether it is directed (d) and weighted (w). We then show the preprocessing time and average number of hubs per label if we run PL with vertices ordered by degree (with ties in the order broken at random) or if we run SamPG, our new ordering algorithm. We then show the space and average time for random queries for the two main label representations we propose: RXL (*Robust eXact Labeling*) uses delta compression and CRXL (*Compressed RXL*) uses two-level mask compression. Both use SamPG. The additional preprocessing time for RXL (over SamPG) is very small (delta compression is fast), but CRXL increases the preprocessing times by 20%–50% (due to parent pointer computation and token generation).

We confirm Akiba et al.'s observation that ordering by degree works well on some inputs. SamPG is much more robust, however, often finding much smaller labels (as in Indo, rgg20-w, buddha-w, or fla-t). Because both algorithms have superlinear dependence on label size, SamPG is much faster when it finds better labels. However, since SamPG spends about two-thirds of its time maintaining sampled trees, it is slower when label sizes are similar.

RXL can handle instances with up to tens of millions of arcs and supports queries in microseconds. Compared to RXL, CRXL reduces space usage by up to an order of magnitude (as in eur-t and uk2002). Query times increase mainly due to worse locality, but still take only microseconds. On uk2002, with almost 300 million arcs, it uses only 2.5 GiB and answers queries in 25 µs.

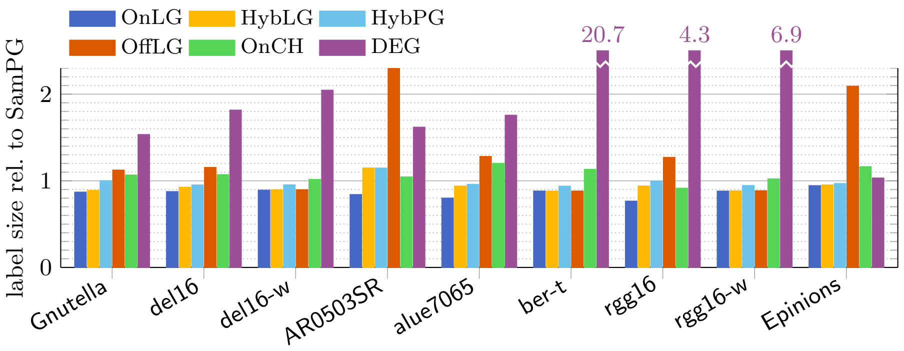
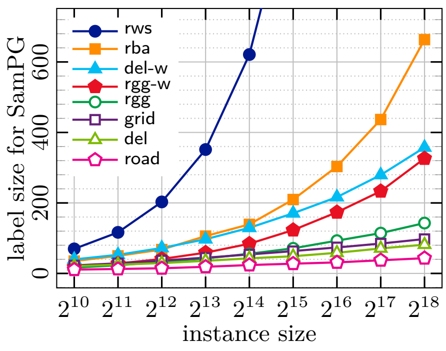


Fig. 1. Label sizes of various orderings relative to SamPG

Fig. 1 shows, for the ordering algorithms discussed in Section 3, their average label sizes *relative to SamPG*; shorter bars are better. As expected, degree is the least robust order. Differences between the other approaches are much smaller, but still significant. When ties are numerous, OffLG [3], the label-greedy algorithm that breaks ties in advance (off-line), is much worse than other methods. HybLG, which uses the same order but breaks ties on-line with PL when building the labels, is much better, as is its path-greedy variant (HybPG). Adding sampling to HybPG yields SamPG, with almost no loss in quality. In fact, SamPG can be better (as in the game graph AR0503SR), since tie-breaking is partially on-line, with new trees representing only uncovered pairs. Most importantly, SamPG is asymptotically faster: even on such small instances, the median time (not shown) is less than half a minute for SamPG, about half an hour for HybPG, HybLG, and OffLG, and days for OnLG. The median time for the CH-based order (OnCH) is only a minute, and it is twice as fast as SamPG on ber-t (Berlin). Although it is not robust, taking hours on Epinions and Gnutella,



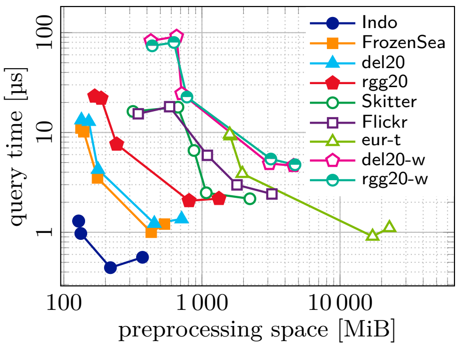


Fig. 2. Left: label sizes for SamPG. Right: space and time tradeoffs; from left to right, the curves are CRXL, $CRXL_1$, HLC, RXL, plain (CRXL and $CRXL_1$ may coincide).

Table 2. Average label size (superhubs for PLL, hubs for RXL), preprocessing time, space, and query times for various methods.

	label size		**preprocessing [s]**				**space [MiB]**				**query [µs]**			
instance	PLL	RXL	PLL	Tree	RXL	CRXL	PLL	Tree	RXL	CRXL	PLL	Tree	RXL	CRXL
Gnutella*	644×16	791	54	209	307	451	209	68	95.7	49.1	5.2	19.0	7.1	45.9
Epinions*	33×16	118	2	128	31	39	32	42	19.1	7.7	0.5	11.0	1.1	4.1
Slashdot*	68×16	219	6	343	85	110	48	83	37.4	17.8	0.8	12.0	1.7	8.0
NotreDame*	34×16	25	5	243	18	22	138	120	22.9	11.9	0.5	39.0	0.2	1.0
WikiTalk*	34×16	113	61	2459	1076	1278	1000	416	560.8	86.5	0.6	1.8	1.0	3.4
Skitter	123×64	273	359	–	2862	3511	2700	–	1074.6	316.7	2.3	–	2.3	20.6
Indo*	133×64	43	173	–	173	201	2300	–	158.6	90.2	1.6	–	0.5	1.8
MetroSec	19×64	116	108	–	2300	2573	2500	–	592.8	207.7	0.7	–	0.8	3.6
Flickr*	247×64	360	866	–	5888	7110	4000	–	1794.6	345.9	2.6	–	2.8	19.9
Hollywood	2098×64	2114	15164	–	61736	75539	12000	–	5934.3	2050.0	15.6	–	13.9	204.0
Indochina*	415×64	91	6068	–	8390	8973	22000	–	1978.8	876.8	4.1	–	0.9	3.9

it finds remarkably small labels, considering that it picks the order based only on local information.

Fig. 2 (left) shows the asymptotic behavior of SamPG on road (square-shaped subgraphs of eur-t) and various synthetic graph classes. Label sizes increase relatively fast for small-world (rws) graphs, and less so for preferential attachment (rba) problems. Higher-diameter inputs have much better behavior. The degree order is asymptotically worse than SamPG for Delaunay triangulations, random geometric graphs, and road networks.

Fig. 2 (right) analyzes the trade-off between space usage and query times for various compression techniques (cf. Section 4). We consider five different representations of the same (SamPG) labels; from left to right, these are *CRXL*, *CRXL*$_1$, *HLC*, *RXL*, and *plain*. The *plain* method represents all hub IDs as 32-bit integers and distances with as few bits as needed (8, 16, or 32) in each case. By incorporating delta compression for hub IDs, RXL uses as little as half as much space as the plain representation, and often has faster queries due to better locality. HLC is Delling et al.'s *hub label compression* [10], but using as few bits as needed (8, 16, or 32) for all distances; it has good compression ratio for road and other high-diameter networks, but is less effective for small-diameter graphs (such as Skitter). CRXL$_1$ and CRXL use supertokens and bitmasks; while CRXL$_1$ uses only one level, CRXL may use two. Both are most effective on small-diameter networks. The extra level often helps, but not always (as in Indo). Queries take a few microseconds, fast enough for most applications.

Table 2 compares RXL and CRXL to two state-of-the-art algorithms. PLL is a restricted variant of PL by Akiba et al. [4] tailored to unweighted and undirected networks. This extended PL algorithm joins each new hub v in the order with a small set $S(v)$ of neighboring vertices, then adds all vertices in the "superhub" $\{v\} \cup S(v)$ to all labels that would benefit from at least one vertex in the set. It stores $\text{dist}(u, v)$ explicitly, but for $w \in S(v)$ it stores $\text{dist}(u, w) - \text{dist}(u, v)$, which is in $\{-1, 0, 1\}$ on unweighted, undirected graphs. The resulting labeling is not hierarchical (any two vertices u, w in $S(v)$ will be in each other's labels), but uses less space and has faster preprocessing (all $|\{v\} \cup S(v)|$ searches run simultaneously). The second algorithm, *Tree Decomposition* [5] (Tree), is not label-based. We report preprocessing time (including SamPG for our methods), space, and average query time, as well as the average number of hubs for RXL and superhubs for PLL ($\times 16$ and $\times 64$ indicate superhub sizes). Tree and PLL were run (sequentially) on a 2.93 GHz Intel Xeon X5670 [4], a machine similar to ours. For consistency with previous work [4, 5], all inputs in Table 2 are undirected; those obtained from directed ones are marked by asterisks.

Superhubs are quite effective in accelerating PLL preprocessing, which is generally faster than for RXL (notably for MetroSec or WikiTalk). Even so, RXL (which does not use superhubs and is more general) has comparable query times and uses less space, sometimes by a large margin, as in Indo and Indochina. In fact, RXL often has fewer hubs than PLL has superhubs, indicating that SamPG indeed finds good orders. Tree is slower than RXL and sometimes uses much more space. CRXL requires less space than any other method.

We conclude that our approach is quite robust. By combining a new sampling-based order (leveraging both HHL [3] and PL [4]) and a novel label representation, RXL is competitive with any other technique, each specialized in different graph classes (such as road networks or social graphs).

References

1. Abraham, I., Delling, D., Fiat, A., Goldberg, A.V., Werneck, R.F.: HLDB: Location-based services in databases. In: GIS, pp. 339–348. ACM Press (2012)
2. Abraham, I., Delling, D., Goldberg, A.V., Werneck, R.F.: A hub-based labeling algorithm for shortest paths on road networks. In: Pardalos, P.M., Rebennack, S. (eds.) SEA 2011. LNCS, vol. 6630, pp. 230–241. Springer, Heidelberg (2011)
3. Abraham, I., Delling, D., Goldberg, A.V., Werneck, R.F.: Hierarchical hub labelings for shortest paths. In: Epstein, L., Ferragina, P. (eds.) ESA 2012. LNCS, vol. 7501, pp. 24–35. Springer, Heidelberg (2012)
4. Akiba, T., Iwata, Y., Yoshida, Y.: Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In: SIGMOD, pp. 349–360. ACM (2013)
5. Akiba, T., Sommer, C., Kawarabayashi, K.-I.: Shortest-path queries for complex networks: Exploiting low tree-width outside the core. In: EBDT, pp. 144–155 (2012)
6. Boldi, P., Rosa, M., Santini, M., Vigna, S.: Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In: WWW, pp. 587–596 (2011)
7. Brandes, U.: A faster algorithm for betweenness centrality. Journal of Mathematical Sociology 25(2), 163–177 (2001)
8. Delling, D., Goldberg, A.V., Pajor, T., Werneck, R.F.: Customizable route planning. In: Pardalos, P.M., Rebennack, S. (eds.) SEA 2011. LNCS, vol. 6630, pp. 376–387. Springer, Heidelberg (2011)
9. Delling, D., Goldberg, A.V., Pajor, T., Werneck, R.F.: Robust Exact Distance Queries on Massive Networks. MSR-TR-2014-12. Microsoft Research (2014)
10. Delling, D., Goldberg, A.V., Werneck, R.F.: Hub label compression. In: Bonifaci, V., Demetrescu, C., Marchetti-Spaccamela, A. (eds.) SEA 2013. LNCS, vol. 7933, pp. 18–29. Springer, Heidelberg (2013)
11. Gavoille, C., Peleg, D., Pérennes, S., Raz, R.: Distance Labeling in Graphs. Journal of Algorithms 53, 85–112 (2004)

12. Geisberger, R., Sanders, P., Schultes, D.: Better approximation of betweenness centrality. In: ALENEX, pp. 90–100 (2008)
13. Geisberger, R., Sanders, P., Schultes, D., Vetter, C.: Exact Routing in Large Road Networks Using Contraction Hierarchies. Trans. Science 46(3), 388–404 (2012)
14. Goldberg, A.V., Razenshteyn, I., Savchenko, R.: Separating hierarchical and general hub labelings. In: Chatterjee, K., Sgall, J. (eds.) MFCS 2013. LNCS, vol. 8087, pp. 469–479. Springer, Heidelberg (2013)
15. Hagberg, A.A., Schult, D.A., Swart, P.J.: Exploring network structure, dynamics, and function using NetworkX. In: SciPy, pp. 11–15 (2008)
16. Holtgrewe, M., Sanders, P., Schulz, C.: Engineering a scalable high quality graph partitioner. In: IPDPS, pp. 1–12. IEEE (2010)
17. Jiang, M., Fu, A.W.C., Wong, R.C.W., Cheng, J., Xu, Y.: Hop doubling label indexing for point-to-point distance querying on scale-free networks, coRR (2014)
18. Jin, R., Ruan, N., Xiang, Y., Lee, V.: A highway-centric labeling approach for answering distance queries on large sparse graphs. In: SIGMOD, pp. 445–456 (2012)
19. Peleg, D.: Proximity-preserving labeling schemes. Journal of Graph Theory 33(3), 167–176 (2000)
20. Schenkel, R., Theobald, A., Weikum, G.: HOPI: An efficient connection index for complex XML document collections. In: Bertino, E., Christodoulakis, S., Plexousakis, D., Christophides, V., Koubarakis, M., Böhm, K. (eds.) EDBT 2004. LNCS, vol. 2992, pp. 237–255. Springer, Heidelberg (2004)
21. Sommer, C.: Shortest-path queries in static networks. ACM Computing Surveys 46, 547–560 (2014)
22. Tarjan, R.: Data Structures and Network Algorithms. SIAM (1983)
23. Wei, F.: TEDI: Efficient shortest path query answering on graphs. In: SIGMOD, pp. 99–110. ACM (2010)

A Dynamic Data Structure for MSO Properties in Graphs with Bounded Tree-Depth[⋆]

Zdeněk Dvořák, Martin Kupec, and Vojtěch Tůma

Computer Science Institute, Charles University
Prague, Czech Republic
{rakdver,kupec,voyta}@iuuk.mff.cuni.cz

Abstract. Tree-depth is an important graph parameter which arose in the study of sparse graph classes. We present a dynamic data structure for representing a graph G with tree-depth at most D. The structure allows addition and removal of edges and vertices under assumption that the resulting graph still has tree-depth at most D, in time bounds depending only on D. A tree-depth decomposition of the graph is maintained explicitly.

This makes the data structure useful for dynamization of static algorithms for graphs with bounded tree-depth. As an example application, we give a dynamic data structure for MSO property testing.

1 Introduction

A *dynamic data structure* maintains information about some object during a sequence of changes and can answer queries about specific properties of the object. For example, we might want to maintain a graph during a sequence of edge additions and removals and be able to answer queries about its connectivity [1]. Of course, we care only about solutions that are significantly faster than recomputing the answer from scratch each time.

In addition to theoretical interest, dynamic data structures have important practical applications when processing rapidly changing input data, such as the web graph or graphs of transportation and social networks. Another application is in generation of random graphs by a random process which adds edges one by one and at each point needs to check whether some desired property is maintained. Further classical examples are the usage of a disjoint-find-union data structure in minimal spanning tree algorithms [2] or link-cut trees for network flow algorithms [3]. A more recent example along these lines is a data structure for subgraph counting [4] with applications in graph coloring and social networking.

In all the mentioned examples, a single specific graph property is maintained. In recent years, there has been a lot of interest in *meta-algorithms*, i.e., algorithms which can be applied to a large class of problems described through some

[⋆] Supported by KONTAKT II LH12095 and SVV 267313.

general mechanism, thus saving the need to give an algorithm for each such problem separately. Due to their generality, the time complexity of meta-algorithms typically involves large multiplicative constant, making them mostly interesting from theoretical point of view (although they can give a guidance for designing practical algorithms for particular problems). Let us mention two important examples.

– Courcelle [5] proved that any property that can be expressed in Monadic Second Order (MSO) logic is decidable in linear time for any class of graphs with bounded tree-width.
  Note that many interesting hard problems such as 3-colorability or the existence of Hamiltonian cycle can be expressed in MSO logic. Furthermore, the choice of bounded tree-width is natural, since MSO logic is not polynomial-time tractable on graph classes with large tree-width under further reasonable assumptions [6,7].
– Dvořák et al. [8] proved that any property that can be expressed in First Order (FO) logic is decidable in linear time for any class of graphs with bounded expansion. Grohe et al. [9] extended this result to nowhere-dense graph classes (see [10] for the definitions of bounded expansion and nowhere-denseness).
  The problems expressible in FO logic include the existence of a fixed subgraph, or the existence of a dominating set of bounded size. Unlike the MSO case, all FO properties are decidable in polynomial time. However, in this case it is natural to study Fixed-Parameter Tractability, i.e., finding algorithms with time complexity $K \cdot n^c$, where only K may depend on the problem (while c is an absolute constant). The restriction to nowhere-dense graph classes is necessary, unless FPT = W[1], see [8].

We are interested in extending these results to dynamic setting, finding dynamic data structures able to preserve MSO or FO properties efficiently for the respective classes of graphs (bounded tree-width or bounded expansion/nowhere-dense). In this paper, we consider the special case of graphs with *bounded tree-depth*, which naturally appears in both of these settings.

The *depth* of a rooted forest T is the maximum number of vertices of a path from a root to a leaf of T. The *closure* $\mathrm{clos}(T)$ of the rooted forest T is the graph obtained from T by adding all edges (x, y) such that x is an ancestor of y. For example, the closure of a path rooted in one of its ends is a complete graph.

Definition 1. *The* tree-depth $\mathrm{td}(G)$ *of a graph G is the minimum integer $t \geq 0$ such that there exists a rooted forest of depth t whose closure contains G as a subgraph.*

For example, the tree-depth of a path on n vertices is $\lceil \log_2(n+1) \rceil$, see Figure 1. Alternatively, the tree-depth can be defined using rank function, vertex ranking number, minimum elimination tree or weak-coloring numbers. Tree-depth is also related to other structural graph parameters—it is greater or equal to path-width (and thus also tree-width), and smaller or equal to the smallest vertex

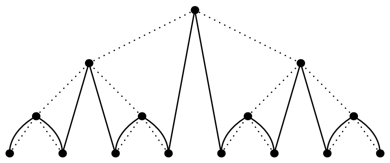


Fig. 1. The tree-depth of a path

cover. An n-vertex graph with tree-width t has tree-depth at most $(t+1)\log_2 n$. See [11,10] for more information.

Let us remark that a subgraph-closed class of graphs has bounded tree-depth if and only if it does not contain arbitrarily long paths. Long paths turn out to be related to the hardness of model checking for MSO logic [12,13]. This motivated a search for meta-theorems similar to [5] on more restricted classes of graphs, such as the result of Lampis [14] that provides algorithms with better dependence on the size of the formula for classes with bounded vertex cover or bounded max-leaf number. This result was subsequently generalized by Gajarský and Hliněný to graphs with bounded tree-depth [15].

Tree-depth also appears prominently in the sparse graph theory and particularly in the theory of graph classes with bounded expansion and nowhere-dense graph classes. Graphs in such classes have *low tree-depth colorings* by a bounded number of colors, as shown by the following result (which can also be taken as a definition of bounded expansion).

Theorem 1 (Nešetřil and Ossona de Mendez [16]). *Let $\mathcal{G}$ be a class of graphs with bounded expansion. For any $k \geq 1$ there exists $c \geq 1$ such that every graph from $\mathcal{G}$ has a coloring by at most c colors in that the union of any $t \leq k$ classes induces a subgraph of tree-depth at most t.*

These colorings can be found in linear time, thus giving a basis of many algorithmic results for classes with bounded expansion, by making it possible to focus only on graphs with bounded tree-depth. This is also the case with the meta-algorithm of Dvořák et al. [8] for FO properties.

Thus, giving a dynamic data structure for maintaining MSO properties on graphs with bounded tree-depth is a necessary ingredient for an attempt to dynamize the result of Dvořák et al. [8]. Furthermore, it is a step towards the dynamization of Courcelle's result on graphs with bounded tree-width [5]. The main theorem of our paper follows.

Theorem 2. *Let ϕ be an MSO formula and let $D \geq 1$ be an integer. There exists a data structure for representing a graph G with $\mathrm{td}(G) \leq D$ supporting the following operations:*

- *insert an edge e, provided that $\mathrm{td}(G + \{e\}) \leq D$,*
- *delete an edge e,*

– *add or delete an isolated vertex,*
– *query—determine whether G satisfies the formula ϕ.*

The time complexity of all the operations is constant (depending only on D and ϕ).

The basic idea of the data structure is to explicitly maintain a forest of smallest depth whose closure contains G, together with its compact constant-size (depending only on D and ϕ) summary obtained by identifying "equivalent" subtrees. This summary is sufficient to decide the property expressed by ϕ, as shown by Gajarský and Hliněný [15].

2 Preliminaries

In Theorem 2, we can assume that the represented graph is connected—we add a new vertex adjacent to all the vertices of the represented graph (this increases tree-depth by at most 1), and modify the formula ϕ to exclude this new universal vertex, i.e., we consider the formula

$$(\exists u)[(\forall v)u = v \vee \mathrm{edge}(u,v)] \wedge \phi',$$

where ϕ' is obtained from ϕ by asserting in each quantification that the quantified vertex is not equal to u or that the quantified set does not contain u or edges incident with u. Hence, we only consider connected graphs in the rest of the paper.

All trees we work with are rooted. Two trees are isomorphic if there exists a graph isomorphism between them that preserves the root. Given rooted trees T_1, …, T_n, let $R(\{T_1,\dots,T_n\})$ denote the tree whose root is a new vertex adjacent to the roots of T_1, …, T_n. If T is a rooted tree and a graph H is a spanning subgraph of $\mathrm{clos}(T)$, we say that T is a *tree-depth decomposition* of H. Given two pairs (H_1,T_1) and (H_2,T_2), where T_1 is a tree-depth decomposition of H_1 and T_2 is a tree-depth decomposition of H_2, we say that (H_1,T_1) is *isomorphic* to (H_2,T_2) if there exists a common isomorphism of H_1 with H_2, and of T_1 with T_2.

It is useful to keep the following simple observation in mind.

Lemma 1 (Nešetřil and Ossona de Mendez [11], Lemma 2.2). *Let H be a connected graph. If H is a single vertex, then* $\mathrm{td}(H) = 1$. *Otherwise, there exists $v \in V(H)$ such that if H_1, …, H_m are the components of $H - v$ and for $1 \le i \le m$, T_i is a tree-depth decomposition of H_i of depth* $\mathrm{td}(H_i)$, *then $R(T_1,\dots,T_m)$ is a tree-depth decomposition of H of depth* $\mathrm{td}(H)$. *Consequently,*

$$\mathrm{td}(H) = 1 + \min\big\{\max\{\mathrm{td}(H') : H' \textit{ is a component of } H - v\} : v \in V(H)\big\}.$$

If v is a vertex of a rooted tree T, we say that the subtree of T induced by v and all its descendants is the *limb of T (rooted in v).* Consider a connected graph H and its tree-depth decomposition T. We say that T is *tight for H* if its depth is equal to the tree-depth of H. The tree T is *optimal (for H)* if every limb T_0 of T is tight for the subgraph of H induced by $V[T_0]$ and additionally, this subgraph is connected. We say that T is *near-optimal* if this condition holds for all limbs other than T itself. Note that the tree-depth decomposition obtained by recursively applying Lemma 1 is optimal.

Given a limb T_0 of T rooted at v, let $\overline{T_0}$ denote the subtree of T consisting of T_0 and of the path from v to the root of T. Two limbs T_1 and T_2 of a tree-depth decomposition T of a graph G are *interchangeable* if $(G[V(\overline{T_1})], \overline{T_1})$ is isomorphic to $(G[V(\overline{T_2})], \overline{T_2})$. The following result is a reformulation of Lemma 3.1 from [15].

Lemma 2. *Let ϕ be a formula with at most one free variable in MSO logic, and let $D \geq 1$ be an integer. There exists an integer $S \geq 1$ with the following property. Let G be a connected graph, let T be its tree-depth decomposition of depth at most D, let v be a vertex of T such that more than S limbs of T rooted in the sons of v are pairwise interchangeable, and let G' be the graph obtained from G by deleting the vertices of one of these limbs. If ϕ has no free variables, then*

$$G' \models \phi \text{ iff } G \models \phi.$$

If ϕ has a free variable x, then for every $u \in V(G')$, we have

$$G', x := u \models \phi \text{ iff } G, x := u \models \phi.$$

Let us remark that in the lemma, the subtree of T induced by $V(G')$ is a tree-depth decomposition of G'. Hence, we can repeat the described reduction until each vertex of the tree has at most S sons such that the limbs rooted in them are pairwise interchangeable. As we will see below in Lemma 3, the size of the resulting graph $G^\star$ is bounded by a constant depending only on ϕ and D, and thus we can decide whether $G^\star$ (and thus also the original graph G) satisfies ϕ by brute force in constant time.

We reorganize this procedure as follows. Let G be a graph and T its tree-depth decomposition of depth D. The *S-code* C_v of a vertex $v \in V(G)$ is a pair (G_v, T_v), where T_v is a subtree of T with the same root as T and G_v is the subgraph of G induced by $V(T_v)$, defined recursively as follows.

- If v is a leaf of T, then T_v consists of the path from v to the root of T and G_v is the subgraph of G induced by its vertices.
- Suppose that v is not a leaf and let $\mathcal{C}$ be the set of S-codes of the sons of v. Let $\mathcal{C}'$ be a maximal subset of $\mathcal{C}$ such that each element of $\mathcal{C}'$ is isomorphic to less than S other elements of $\mathcal{C}'$. Let $G_v = \bigcup_{(G',T') \in \mathcal{C}'} G'$ and $T_v = \bigcup_{(G',T') \in \mathcal{C}'} T'$.

Observe that the S-codes are defined uniquely up to isomorphism, and that if v is the root of G, then G_v is isomorphic to $G^\star$. We can bound the size of each S-code.

Lemma 3. *For any integers $D, S \geq 1$, there exists $s \geq 0$ such that if G is a graph and T is its tree-depth decomposition of depth D, then the S-code of each vertex of G has at most s vertices.*

Proof. Let $K_1 = D$, $N_1 = 2^{\binom{d}{2}}$, and for $d \geq 2$, let $K_d = SN_{d-1}K_{d-1}$ and $N_d = N_{d-1} + (S+1)^{N_{d-1}}$. We let $s = K_D$.

Let $\mathcal{C}_d$ be the set of all pairwise non-isomorphic S-codes of vertices $w \in V(G)$ such that the limb of T rooted in w has depth at most d. We prove by induction on d that $|\mathcal{C}_d| \leq N_d$ and that every element of $\mathcal{C}_d$ has at most K_d vertices. Note that $\mathcal{C}_1$ contains exactly the S-codes of leaves of T, and observe that it satisfies the claim.

Hence, suppose that $d \geq 2$. Consider a vertex $v \in V(G)$ such that the limb of T rooted in v has depth exactly d, and let $\mathcal{C}'$ be as in the definition of the S-code. By induction, each element of $\mathcal{C}'$ is isomorphic to an element of $\mathcal{C}_{d-1}$, and each of them has at most K_{d-1} vertices. It follow that $|\mathcal{C}'| \leq SN_{d-1}$ and that the S-code of v has at most $|\mathcal{C}'|K_{d-1} \leq K_d$ vertices. Furthermore, the S-code of v is determined by how many elements of $\mathcal{C}_S$ are isomorphic to each of the elements of $\mathcal{C}_{d-1}$, and thus there are at most $(S+1)^{N_{d-1}}$ choices for the S-code of v.

We are also going to need the following observation.

Lemma 4. *Let $D, S \geq 1$ be fixed integers. Let G be a graph and T its tree-depth decomposition of depth D. Let v be a vertex of T, let T' be the limb of T rooted at v and let $G' = G[V(T')]$. Let (G_v, T_v) be the S-code of v in (G,T) and let Z be the set of vertices of G on the path from v to the root, excluding v. Then the S-code of v in (G', T') is isomorphic to the S-code of v in $(G_v - X, T_v - X)$. In particular, given the S-code of v in (G,T), its S-code in (G', T') can be determined in constant time.*

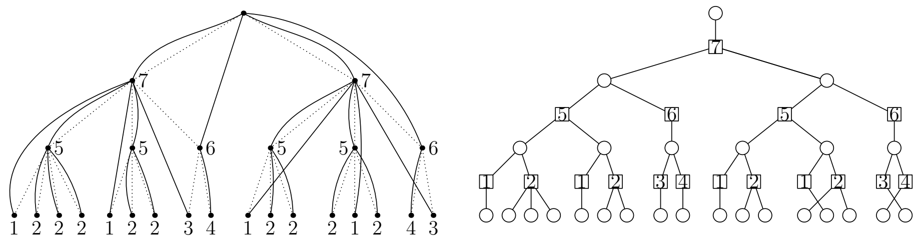


Fig. 2. A 2-archive. Merge nodes and cabinets are drawn by squares and circles, respectively. The numbers at vertices and merge nodes indicate the isomorphism classes of 2-codes.

Let G be a graph and T its tree-depth decomposition of depth D. Let A be obtained from T as follows (see Figure 2 for an illustration). For each non-leaf vertex $v \in V(T)$, we divide the sons of v to groups according to the isomorphism class of their S-code. For each such group M, we remove the edges between M and v, add a new vertex v_M, add edges between v_M and the vertices of M, and add the edge vv_M. We call A the *S-archive* of (G,T). We call the vertices of

A at even distance from the root (the original vertices of T) cabinets, and the vertices of A at odd distance *merge nodes*. Note that each cabinet has at most N_{D-1} sons, while the merge nodes may have arbitrarily large degree.

3 Data Structure

Suppose that $D \geq 1$ and an MSO formula ϕ are fixed. Let S be the smallest integer such that the conclusion of Lemma 2 holds for both ϕ and the formula (3) below, and for graphs with tree-depth decompositions of depth at most $2D$ (rather than D).

Let us now describe our data structure for representing a graph G of tree-depth at most D. We maintain the graph G, using any standard way of representing a graph, and the S-archive A of some *optimal* tree-depth decomposition T of G. To represent the tree A, each vertex v of A records a pointer to its father and a double-linked list of its sons. Furthermore, v records its position in the list of sons of its father, so that given v, we can cut the edge between v and its father in constant time. At each cabinet of A, we record the corresponding vertex of G.

For a vertex v of A, let U_v denote the set of vertices of G whose cabinets lie on the path from v to the root of A; note that $|U_v| \leq D$ and that the vertices of U_v can be enumerated in constant time.

For each cabinet, we need to be able to determine its S-code in (G, T). Hence, for each cabinet z, we maintain a set $Q_z \subseteq V(G)$ such that

$$\textit{the } S\textit{-code of } z \textit{ is isomorphic to } (G[Q_z \cup U_z], T[Q_z \cup U_z]), \tag{1}$$

and

$$\textit{all cabinets in } Q_z \textit{ are contained in the limb of } A \textit{ rooted in } z. \tag{2}$$

As we observed before, if we choose S large enough (depending only on the formula ψ and the depth D), then the S-code represented in the root cabinet of A determines whether $G \models \psi$, and thus we can answer the queries in constant time. Hence, we only need to consider the implementation of the update operations.

Note that the tree-depth decomposition T can be determined from its S-archive A—to enumerate the sons of a vertex $v \in V(T)$ in T, it suffices to enumerate the grandsons of v in A. Since the degree of the cabinet v is bounded by a constant, this can be done in time $O(\deg(v))$. Similarly, the father of v in T is equal to the grandfather of v in A. On the other hand, we cannot directly maintain T, since our implementations of the operations with the data structure may result in a non-constant number of changes in T (which however correspond to only a constant number of changes in its S-archive).

4 Auxiliary Operations

First, let us describe several auxiliary operations. Let us remark that the invariant that the tree-depth decomposition represented by the S-archive A is optimal might not hold when these operations are applied, and we list the optimality assumptions where required. However, we always maintain the invariant that the depth of the tree-depth decomposition represented by A is at most $2D$.

4.1 Correcting *S*-Codes

Let z_1 be a cabinet of A, and let z_1, z_2, …, z_k be the cabinets A in the path from z_1 to the root z_k of A. Suppose that the invariants (1) and (2) hold for all cabinets of A other than z_1, …, z_k, but they might be violated for some of z_1, …, z_k. The operation of *correcting S-codes above* z_1 changes A and the values of Q_{z_1}, …, Q_{z_k} to ensure that the invariants (1) and (2) hold for all vertices of A.

First, we determine the value Q_{z_1} as follows. If z_1 is a leaf, then we set $Q_{z_1} = \emptyset$. Otherwise, let m_1, …, m_t be the sons of z_1, and for $1 \le i \le t$, let Z_i be the set containing all sons of m_i if m_i has at most S sons, and the first S sons of m_i otherwise. Note that m_i is a merge node and the elements of Z_i are cabinets. We set $Q_{z_1} = \bigcup_{i=1}^{t} \bigcup_{z \in Z_i} (\{z\} \cup Q_z)$.

Next, we need to ensure that z_1 is the son of the correct merge node that appears as a son of z_2; i.e., we remove z_1 from the list of sons of its father, find the son m of z_2 such that the S-code of the sons of m is isomorphic to the S-code of z_1 (creating a new merge node if no such son m exists), and add z_1 as the son of m.

Then, we repeat the previous two paragraphs for z_2, …, z_k in turn.

4.2 Extracting a Path

Given a cabinet v of the S-archive A, the operation of *extracting* v modifies A so that each merge node on the path from v to the root of A has exactly one son. That is, for each cabinet z on the path from v to the root of A such that the father m_0 of z in A has at least two sons, we create a new merge node m, make m a son of the grandfather of z, remove z from the list of sons of m_0, and add z as the son of m. This violates the property of S-archives that no two merge nodes with the common father have sons with the same S-code; in the applications of extraction, the property is restored (by applying the operation of correcting S-codes) immediately after performing some futher simple change.

4.3 Finding a Best Root

Consider a limb B of A rooted in a cabinet, and let T_B be the corresponding limb of the tree-depth decomposition represented by A. Assuming that T_B is a *near-optimal* tree-depth decomposition of a connected subgraph $G[V(T_B)]$, the operation of *finding a best root for B* returns a vertex $v \in V(T_B)$ such that there exists an an *optimal* tree-depth decomposition of $G[V(T_B)]$ rooted in v.

Let us argue that this property of v is expressible in MSO logic. Note that

$$\gamma(X) := (\forall Y \subsetneq X)\, Y \neq \emptyset \Rightarrow (\exists x \in X \setminus Y)(\exists y \in Y)\, \mathrm{edge}(x, y)$$

is a formula expressing that X induces a connected subgraph. Next, we recursively define a formula $\tau_d(X,r)$ which decides whether the connected subgraph induced by X is contained in the closure of a tree of depth d rooted in r. We let $\tau_1(X,r) := X = \{r\}$, and for any $d \geq 2$, we set

$$\tau_d(X,r) := (\forall Y \subseteq X \setminus \{r\})(Y \neq \emptyset \wedge \gamma(Y)) \Rightarrow \sigma_{d-1}(Y),$$

where $\sigma_d(X) := (\exists r \in X)\tau_d(X,r)$ tests whether X induces a subgraph of tree-depth at most d. Finally, the formula

$$\rho(r) := \tau_1(V,r) \vee (\tau_2(V,r) \wedge \neg\sigma_1(V)) \vee \ldots \vee (\tau_D(V,r) \wedge \neg\sigma_{D-1}(V)) \qquad (3)$$

tests whether there exists an optimal tree-depth decomposition of depth at most D rooted in r. Thus, it suffices to find a vertex of $G[V(T_B)]$ satisfying ρ.

By the choice of S, we can apply Lemma 2 to test the validity of ρ in $G[V(T_B)]$, assuming that we know the S-code of the root of T_B in $(G[V(T_B)], T_B)$. By Lemma 4, this S-code can be determined in constant time. Furthermore, observe that since $G[V(T_B)]$ has some optimal tree-depth decomposition, ρ is satisfied for at least one cabinet of B reachable from the root by a path which always takes the first son in each merge node. Hence, we only need to test ρ for constantly many vertices of $G[V(T_B)]$, and thus the time complexity of finding a best root for B is constant.

4.4 Rerooting and Restoring Optimality

Finally, let us describe two auxiliary operations, whose implementations are mutually recursive. Assuming that the tree-depth decomposition represented by A is optimal and given a vertex $v \in V(G)$, *rerooting in* v modifies the S-archive A so that it represents a *near-optimal* tree-depth decomposition of G whose root is v. Conversely, assuming that the tree-depth decomposition represented by A is *near-optimal*, the operation of *optimization* modifies the S-archive A so that it represents an optimal tree-depth decomposition of G.

To facilitate recursive implementation, we actually need more general versions of these operations, specified as follows. For a cabinet v of A, let $d(v)$ denote the number of cabinets on the path from v to the root of A, excluding v itself.

– Let r and v be two vertices of G, such that v is a descendant of r in A, and let $d_r = d(r)$. Let A' be the limb of A rooted in r, let T' be the tree-depth decomposition represented by A', let $G' = G[V(T')]$ and let $d_{T'}$ be the depth of T'. We assume that T' is a *near-optimal* tree-depth decomposition of G', that G' is connected, and that

$$d_r + d_{T'} + \mathrm{td}(G' - v) \leq 2D. \qquad (4)$$

The generalized rerooting operation modifies the part of A contained in A' so that it represents a *near-optimal* tree-depth decomposition T'' of G' rooted in v. Note that the depth of T'' is at most $\mathrm{td}(G'-v)+1 \leq 2D+1-d_{T'}-d_r \leq 2D - d_r$, and thus the depth of the tree-depth decomposition represented by A after this operation is at most $2D$.

– Let r be a vertex of G, let A' be the limb of A rooted in r, let T' be the tree-depth decomposition represented by A', let $G' = G[V(T')]$, and let $d_{T'}$ be the depth of T'. Assuming that T' is a *near-optimal* tree-depth decomposition of G', that G' is connected, and that

$$d(r) + d_{T'} + \mathrm{td}(G') \leq 2D + 1, \tag{5}$$

the generalized optimization operation modifies the part of A contained in A' so that it represents an *optimal* tree-depth decomposition of G'.

The generalized optimization first finds a best root v for A' as described in Subsection 4.3. It then uses the generalized rerooting operation to modify A' so that it represents a near-optimal tree-depth decomposition T'' of G' rooted in v. Since there exists an optimal tree-depth decomposition of G' rooted in v and G' is connected, it follows that all components of $G' - v$ have tree-depth at most $\mathrm{td}(G') - 1$, and since T'' is near-optimal, each limb of T'' rooted in a son of v has depth at most $\mathrm{td}(G') - 1$. Consequently, T'' has depth $\mathrm{td}(G')$, and thus it is optimal. Furthermore, we have $\mathrm{td}(G' - v) = \mathrm{td}(G') - 1$, and thus (5) ensures that the assumption (4) holds in the rerooting call.

The rest of this subsection is devoted to the operation of rerooting. We assume that $v \neq r$, as otherwise the operation does not need to do anything.

– First, we find the son w of r in T' such that the limb T_1 rooted in w contains v, and recursively reroots this limb in v (thus, at this point, v is a son of r in T'). Let $G_1 = G[V(T_1)]$ and note that $\mathrm{td}(G_1 - v) \leq \mathrm{td}(G' - v)$, while the depth of T_1 is at most $d_{T'} - 1$. Hence, the assumption (4) is satisfied in this recursive call.
– Now, remove the edge vr and let T_v and T_r be the components of T' containing v and r, respectively. To perform this operation in the archive, we first extract v as described in Subsection 4.2, then delete the edge between the father of v and r.
– For each limb T_x of T_v rooted in a son x of v such that there exists an edge of G' from r to a vertex of T_x (which can be determined by examining the S-code of x), remove the edge vx from T_v and add the edge rx to T_r, making T_x a limb of T_r. In the S-archive, we do this at once for all limbs with the same S-code which are grouped together in a merge node (by cutting the edge between the merge node and v and adding the edge to r) and thus this phase can be performed in constant time.
– Next, cut the edge between r and its father f in T, add r to T_v as a son of v and add v as a son of f, by performing the corresponding modifications in A. After this, we correct the S-codes above r, which also undoes the effects of the extraction of v in the second step.
– Finally, we apply the generalized optimization operation to the limb T_2 rooted in r. Note that this ensures that the tree-depth decomposition represented by the limb of A rooted in v is near-optimal, since all other limbs of T_v are optimal tree-depth decompositions of the subgraphs that they induce by the assumption that initially, T' is near-optimal. The depth of T_2 is at

most $d_{T'}$, $\text{td}(G[V(T_2)]) \leq \text{td}(G' - v)$ and that at this point, $d(r) = d_r + 1$, hence (4) implies that (5) holds in the optimization call.

Note that the rerooting operation may result in two recursive calls to itself—once directly in the first step (applied to the limb rooted in w), once indirectly through optimization in the last step (applied to the limb rooted in r). However, at the time of the recursive calls, we have $d(w) > d_r$ and $d(r) > d_r$, respectively. Since both recursive calls satisfy (4), we conclude that the maximum depth of the recursion is at most $2D$, and thus the number of recursive calls to generalized rerooting during one invocation of the non-generalized rerooting or optimization operation is at most $2^{2D} - 1$, which is a constant. Consequently, the time complexity of rerooting as well as optimization is constant, bounded by a function of D and ϕ.

5 Updating the Data Structure

Implementing the update operations required by Theorem 2 is now straightforward. Let us mention that unlike the previous section, we assume that the tree-decomposition represented by A is optimal (and thus has depth at most D) at he beginning of each of the operations, and this invariant also holds at the end of each operation (while it may be temporarily violated during the operation).

5.1 Edge Insertion and Deletion

Suppose that we are inserting or deleting an edge uv of G. First, we reroot the tree-depth decomposition in v. Next, we extract u and add or remove the edge uv from G. Then, we correct the S-codes above u; note that this also undoes the extraction, i.e., restores the invariant that no two merge nodes with the common father have sons with the same S-code. Finally, we optimize all the limbs rooted in the cabinets of path from u to the root.

The time complexity of these operations is bounded by a function of D and S, and thus it is constant. Note that for edge deletion, one might be tempted to skip the first rerooting. However, making v a root of the tree ensures that even after removing the edge uv, each limb induces a connected subgraph of G, which is required in the optimization.

5.2 Isolated Vertex Insertion or Deletion

As we mentioned in Section 2, we only work with connected graphs, and to be able to do so, we added a dummy universal vertex u. Thus, addition or removal of an isolated vertex in the original graph corresponds to addition or removal of a vertex adjacent only to u in the represented graph. To do so, we first reroot the tree-depth decomposition in u, and let T' be the resulting near-optimal tree-depth decomposition. If we are removing a vertex v adjacent only to u, observe that v is a son of u and forms a limb of T' by itself. Hence, we can now extract

v, remove v and its father merge node from A and correct the S-code of u in constant time. If we are adding v, we can conversely add a new merge node m as a son of u, add v as a son of m, and and correct the S-codes above v (which also merges m with the appropriate other son of u, if there exists one).

References

1. Kapron, B.M., King, V., Mountjoy, B.: Dynamic graph connectivity in polylogarithmic worst case time. In: Khanna, S. (ed.) SODA, pp. 1131–1142. SIAM (2013)
2. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms. The MIT Press (2009)
3. Sleator, D.D., Endre Tarjan, R.: A data structure for dynamic trees. Journal of Computer and System Sciences 26, 362–391 (1983)
4. Dvořák, Z., Tůma, V.: A dynamic data structure for counting subgraphs in sparse graphs. In: Dehne, F., Solis-Oba, R., Sack, J.-R. (eds.) WADS 2013. LNCS, vol. 8037, pp. 304–315. Springer, Heidelberg (2013)
5. Courcelle, B.: The monadic second-order logic of graphs. I. recognizable sets of finite graphs. Information and Computation 85, 12–75 (1990)
6. Kreutzer, S., Tazari, S.: On brambles, grid-like minors, and parameterized intractability of monadic second-order logic. In: Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, pp. 354–364. Society for Industrial and Applied Mathematics, Philadelphia (2010)
7. Kreutzer, S., Tazari, S.: Lower bounds for the complexity of monadic second-order logic. In: 2010 25th Annual IEEE Symposium on Logic in Computer Science (LICS), pp. 189–198. IEEE (2010)
8. Dvořák, Z., Král', D., Thomas, R.: Deciding first-order properties for sparse graphs. In: FOCS, pp. 133–142. IEEE Computer Society (2010)
9. Grohe, M., Kreutzer, S., Siebertz, S.: Deciding first-order properties of nowhere dense graphs. CoRR abs/1311.3899 (2013)
10. Nešetřil, J., Ossona de Mendez, P.: Sparsity: Graphs, Structures, and Algorithms, vol. 28. Springer (2012)
11. Nešetřil, J., Ossona de Mendez, P.: Tree-depth, subgraph coloring and homomorphism bounds. European Journal of Combinatorics 27, 1022–1041 (2006)
12. Lampis, M.: Model checking lower bounds for simple graphs. CoRR abs/1302.4266 (2013)
13. Frick, M., Grohe, M.: The complexity of first-order and monadic second-order logic revisited. In: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science, pp. 215–224. IEEE (2002)
14. Lampis, M.: Algorithmic meta-theorems for restrictions of treewidth. Algorithmica 64, 19–37 (2012)
15. Gajarsky, J., Hlineny, P.: Faster Deciding MSO Properties of Trees of Fixed Height, and Some Consequences. In: D'Souza, D., Kavitha, T., Radhakrishnan, J. (eds.) IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2012), Germany. Leibniz International Proceedings in Informatics (LIPIcs), vol. 18, pp. 112–123. Dagstuhl, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2012)
16. Nešetřil, J., Ossona de Mendez, P.: Grad and classes with bounded expansion I. decompositions. European Journal of Combinatorics 29, 760–776 (2008)

Large Independent Sets in Triangle-Free Planar Graphs

Zdeněk Dvořák[1,⋆] and Matthias Mnich[2]

[1] Computer Science Institute, Charles University, Prague, Czech Republic
rakdver@iuuk.mff.cuni.cz
[2] Cluster of Excellence MMCI, Campus E1-7, 66123 Saarbrücken
mmnich@mmci.uni-saarland.de

Abstract. Every triangle-free planar graph on n vertices has an independent set of size at least $(n+1)/3$, and this lower bound is tight. We give an algorithm that, given a triangle-free planar graph G on n vertices and an integer $k \geq 0$, decides whether G has an independent set of size at least $(n+k)/3$, in time $2^{O(\sqrt{k})}n$. Thus, the problem is fixed-parameter tractable when parameterized by k. Furthermore, as a corollary of the result used to prove the correctness of the algorithm, we show that there exists $\varepsilon > 0$ such that every planar graph of girth at least five on n vertices has an independent set of size at least $n/(3-\varepsilon)$.

1 Introduction

Every planar graph is 4-colorable by the deep Four-Colour-Theorem, whose proof was first announced by Appel and Haken in 1976 [1] and later simplified by Robertson, Sanders, Seymour and Thomas [2]. As a corollary, every planar graph on n vertices has an independent set of size at least $n/4$. The proof by Robertson et al. [2] comes with a quadratic-time algorithm to find a valid coloring of G with 4 colors, which can be used to find such an independent set. Yet, determining the size of a maximum independent set is NP-complete in planar graphs (even if they are triangle-free [3]). This motivates a search for an efficient algorithm that decides, for a fixed parameter $k \geq 0$ and an input n-vertex planar graph G, whether G has an independent set of size at least $(n+k)/4$.

The problem—which we call Planar Independent Set Above Tight Lower Bound, or Planar Independent Set-ATLB for short—has received a lot of attention, although there has been essentially no progress. In fact, the question of whether Planar Independent Set-ATLB is fixed-parameter tractable has been raised several times: first by Niedermeier [4], later by Bodlaender et al. [5], Mahajan et al. [6], by Sikdar [7], by Mnich [8], and by Crowston et al. [9]. Then the problem was raised again in June 2012 as a "Though Customer", at WorKer 2012 [10]. We remark that until now, there is not even a polynomial-time algorithm known for the case of $k = 1$, and finding such an

⋆ Supported by the project LL1201 (Complex Structures: Regularities in Combinatorics and Discrete Mathematics) of the Ministry of Education of Czech Republic.

algorithm has been an open problem for more than 30 years. Yet, the existence of such an algorithm for $k = 1$ is certainly a necessary condition for the fixed-parameter tractability of PLANAR INDEPENDENT SET-ATLB. The lower bound of $n/4$ on the size of a maximum independent set is tight for an infinite family of planar graphs: for example, take a set of copies of K_4 or C_8^2 or the icosahedron, and connect these copies arbitrarily in a planar way.

We consider a variant of the problem for triangle-free planar graphs. By a theorem of Grötzsch [11], every triangle-free planar graph is 3-colorable, and thus admits an independent set that contains at least one-third of its vertices. Such a coloring, and thus also an independent set, can be found in linear time [12]. Later, Steinberg and Tovey [13] showed that every triangle-free planar graph contains a *non-uniform* 3-coloring, where one color class is guaranteed to contain one vertex more than the other two color classes. Thus, any n-vertex triangle-free planar graph contains an independent set of size at least $(n+1)/3$, when $n \geq 3$. On the other hand, Jones [14] found triangle-free planar graphs on n vertices (for any $n \geq 2$ such that $n \equiv 2 \pmod 3$) with maximum independent sets of size exactly $(n+1)/3$; see Figure 1. This motivates a search for an efficient algorithm

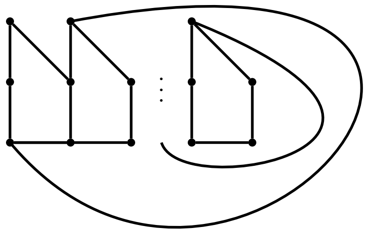


Fig. 1. Triangle-free planar graphs on n vertices with maximum independent sets of size exactly $(n+1)/3$

that decides, for a given n-vertex triangle-free planar graph G and integer $k \geq 0$, whether G has an independent set of size at least $(n+k)/3$. In particular it was open whether for $k = 2$ there is a polynomial-time algorithm. Notice that it is non-trivial even to solve this problem in time $n^{O(k)}$, as a brute-force approach does not suffice.

1.1 Our Contributions

As our main result, we show that the problem is fixed-parameter tractable when parameterized by k.

Theorem 1. *There is an algorithm that, given any n-vertex triangle-free planar graph G and any integer $k \geq 0$, in time $2^{O(\sqrt{k})}n$ decides whether G has an independent set of size at least $(n+k)/3$.*

We conveniently write $(n+k)/3$ rather than $n/3+1/3+k$ (since the tight lower bound on the size of a maximum independent set is $n/3+1/3$); notice that this change does not affect the fixed-parameter tractability of the problem.

Though many different techniques have been devised for solving optimization problems parameterized above lower bounds in fixed-parameter time, none of these techniques is applicable to our problem. Instead, our algorithm seems to be the first fixed-parameter algorithm for problems parameterized above guarantee that is based on the tree-width of a graph. The algorithm is an easy corollary of the following result.

Theorem 2. *There is a constant $c > 0$ such that every planar triangle-free graph on n vertices with tree-width $\geq t$ has an independent set of size $\geq \frac{n+ct^2}{3}$.*

According to Theorem 2, the algorithm of Theorem 1 is extremely simple. First, we test whether the treewidth of G is $O(\sqrt{k})$, using the linear-time constant factor approximation algorithm of Bodlaender et al. [15]. If that is the case, we find the largest independent set in G by dynamic programming in time $2^{O(\sqrt{k})}n$, see e.g. the book of Niedermeier [4]. Otherwise, we answer "yes".

The situation is more complicated when we want to report the independent set of size $(n+k)/3$ if it exists. An inspection of the proof of Theorem 2 (as well as the previous results used to establish its correctness, especially Theorem 4) shows that it is constructive and leads to a quadratic-time algorithm. The description of the algorithm is rather involved and we omit it.

As a by-product of the proof of Theorem 2, we also obtain the following result which is of independent interest.

Theorem 3. *There exists a constant $\varepsilon > 0$ such that every planar graph of girth at least 5 on n vertices has an independent set of size at least $\frac{n}{3-\varepsilon}$.*

A well-known graph parameter giving a lower bound for the independence number is the *fractional chromatic number*, see [16] for a definition and other properties. The fractional chromatic number of planar graphs of girth at least 8 is at most $5/2$ (Dvořák et al. [17]), implying that for girth 8, we can set $\varepsilon = 1/2$. Not much is known about fractional chromatic number of graphs of girth between 5 and 7, but Theorem 3 makes the following conjecture plausible.

Conjecture. *There exists a constant $\varepsilon > 0$ such that every planar graph of girth ≥ 5 has fractional chromatic number at most $3-\varepsilon$.*

Note that if the conjecture holds, then $\varepsilon \leq 1/4$ by a construction of Pirnazar and Ullman [18]. Furthermore, the girth assumption cannot be reduced to 4 (or replaced by assuming odd girth at least 5) because of the construction in Fig. 1.

Let us also remark that the assumption of Theorem 3 that G is planar can be relaxed, since every graph on n vertices embeddable in a surface of genus g can be planarized by removing $O(\sqrt{gn})$ vertices [19].

Corollary 1. *There exist constants $\varepsilon, c > 0$ such that every graph of girth ≥ 5 and genus $\leq g$ on n vertices has an independent set of size $\geq \frac{n}{3-\varepsilon}(1-c\sqrt{g/n})$.*

Organization and proof outline. In Section 3, we review some results on classes of graphs with bounded expansion, which we use to show that in every planar graph, we can remove a small fraction of vertices so that the resulting graph contains a large set of vertices that are pairwise far apart (Section 4). This enables us to apply coloring theory developed by Dvořák, Král' and Thomas to obtain a 3-coloring of the graph with further constraints in the neighborhoods of these distant vertices, which guarantee the existence of a large independent set (Section 5). In Section 6, we combine the results and give proofs of our theorems. Due to space constraints, the proofs of statements marked by $\star$ are omitted.

1.2 Related Work

From the combinatorial side, the study of lower bounds on the independence number in triangle-free graphs has a long history. Every n-vertex triangle-free graph has an independent set of size $\Omega(\sqrt{n \log n})$, and this bound is tight [20]. Staton [21] proved that every subcubic n-vertex triangle-free graph G satisfies $\alpha(G) \geq \frac{5n}{14}$. Furthermore, Heckman and Thomas [22] showed that if G is additionally planar, then $\alpha(G) \geq \frac{3n}{8}$.

From the algorithmic side, the studying the complexity of maximization problems parameterized above polynomial-time computable lower bounds is an active area of research. Since the influental survey by Mahajan et al. [6], research in this area has led to development of many new algorithmic techniques for fixed-parameter algorithms: algebraic methods [23,9], probabilistic methods [24,25], combinatorial methods [26,27], and methods based on linear programming [28].

2 Preliminaries

Throughout, we consider graphs that are finite, undirected and loopless, and do not have parallel edges unless explicitly stated otherwise. For a graph G, let $V(G)$ denote its vertex set and $E(G)$ its set of edges. The degree of a vertex $v \in V(G)$ is the number $\deg_G(v)$ of edges that are incident to it. A graph is *planar* if it admits an embedding in the plane such that no two edges cross; a *plane graph* is an embedding of a planar graph without any edge crossings.

A *tree decomposition* of a graph G is a pair $(T, \mathcal{B})$, where $\mathcal{B}$ is a set of subsets of $V(G)$ (called the *bags* of the decomposition) with $V(G) = \bigcup_{B \in \mathcal{B}} B$, and T is a tree with vertex set $\mathcal{B}$, such that for each edge $uv \in E(G)$, there exists $B \in \mathcal{B}$ containing both u and v, and for every $v \in V(G)$, the set $\{B \in \mathcal{B} : v \in B\}$ induces a connected subtree of T. The width of the decomposition is the size of its largest bag minus one. The *tree-width* of a graph G, denoted by $\mathsf{tw}(G)$, is the minimum width of its tree decompositions.

3 Classes of Graphs with Bounded Expansion

In this section, we survey results on classes of graphs with bounded expansion that we need in the paper. Let G be a graph and let $r \geq 0$ be an integer. Let us recall that a graph H is an *r-shallow minor* of G if H can be obtained from a subgraph of G by contracting vertex-disjoint subgraphs of radii at most r and deleting the resulting loops and parallel edges. Following Nešetřil and Ossona de Mendez [29], we denote by $\nabla_r(G)$ the maximum of $|E(G')|/|V(G')|$ over all r-shallow minors G' of G. Thus, $\nabla_0(G)$ is the maximum of $|E(G')|/|V(G')|$ taken over all subgraphs G' of G. Since every subgraph of a graph G has a vertex of degree at most $2\nabla_0(G)$, we see that G has an (acyclic) orientation with maximum in-degree at most $2\nabla_0(G)$.

A class $\mathcal{G}$ of graphs has *bounded expansion* if there exist constants c_0, c_1, ... such that $\nabla_r(G) \leq c_r$ for every $G \in \mathcal{G}$ and $r \geq 0$. Many natural classes of sparse graphs have bounded expansion. Here, we only need that the class of planar graphs has bounded expansion. See the book of Nešetřil and Ossona de Mendez [30] for more information on the topic.

Let D be a directed graph, and let D' be a directed graph obtained from D by adding, for every pair of vertices $x, y \in V(D)$,

– the edge xy if D has no edge from x to y and there exists a vertex $z \in V(D)$ such that D has an edge oriented from x to z and an edge oriented from z to y (*transitivity*), and
– either the edge xy or the edge yx if x is not adjacent to y and there exists a vertex z such that D has an edge oriented from x to z and an edge oriented from y to z (*fraternality*).

We call D' an *oriented augmentation* of D.

Let G be a graph. We construct a sequence D_0, ..., D_ℓ of directed graphs as follows. Let D_0 be an orientation of G with maximum in-degree at most $2\nabla_0(G)$. For $1 \leq i \leq \ell$, let D_i be an oriented augmentation of D_{i-1}, in that the orientations of the edges added according to the fraternality rule are chosen so that the subgraph of D_i formed by these edges has maximum in-degree at most $2\nabla_0(G_i)$, where G_i is the underlying undirected graph of D_i. This is possible, because G_i itself has an orientation with maximum in-degree at most $2\nabla_0(G_i)$. We say that D_ℓ is an *ℓ-th oriented augmentation* of G. We use the following important property of classes of graphs with bounded expansion [29].

Proposition 1. *Let $\mathcal{G}$ be a class of graphs and let $\ell \geq 0$ be an integer. If $\mathcal{G}$ has bounded expansion, then there exists an integer $m_d \geq 0$ such that all ℓ-th oriented augmentations of graphs from $\mathcal{G}$ have maximum indegree at most m_d.*

4 Large Scattered Sets

For an integer $d \geq 1$ and a graph G, a vertex set $Q \subseteq V(G)$ is *d-scattered* if the distance between any two vertices of Q in G is greater than d. In this section we discuss graph classes whose members admit large scattered subsets, possibly after removal of a small number of vertices.

For integers $d, m, N \geq 1$ and $r \geq 0$, a graph G is *(d, r, m, N)-wide* if for any set $S \subseteq V(G)$ of size at least N, there exists a set $Q \subseteq S$ of size at least m and a set $X \subseteq V(G)$ of size at most r such that Q is d-scattered in $G - X$. For integers $d \geq 1$ and $r \geq 0$, a class of graphs $\mathcal{G}$ is *(d, r)-wide* if for every m, there exists N such that every graph in $\mathcal{G}$ is (d, r, m, N)-wide. A class of graphs $\mathcal{G}$ is *uniformly quasi-wide* if for every integer $d \geq 1$ there exists an integer $r \geq 0$ such that $\mathcal{G}$ is (d, r)-wide. By results of Nešetřil and Ossona de Mendez [29], every class of graphs with bounded expansion is uniformly quasi-wide (in fact, they prove a stronger result concerning nowhere-dense graph classes).

We need a variant of wideness where the value of N is linear in m. Note that there exist classes of graphs with bounded expansion which do not have this property—for instance, consider the class containing the graphs H_a for each integer $a > 0$, where H_a is the disjoint union of a stars with a rays. With $S = V(H_a)$, the largest possible 2-scattered set after removal of r vertices has size $ra + (a - r) = \Theta(\sqrt{|S|})$. However, note that we can obtain a d-scattered set of size $a^2 = |S| - a$ at the cost of removing a vertices, which suggests the variant of wideness where the set of removed vertices is small relatively to the size of the d-scattered set.

For integers $d, t, K \geq 1$, a graph G is *(d, t, K)-fat* if for any set $S \subseteq V(G)$ there exists a set $Q \subseteq S$ of size at least $|S|/K$ and a set $X \subseteq V(G) \setminus Q$ of size at most $|Q|/t$ such that Q is d-scattered in $G - X$. A class of graphs $\mathcal{G}$ is *fat* if for all integers $d, t \geq 1$, there exists K such that every graph in $\mathcal{G}$ is (d, t, K)-fat.

We are going to show that every class of graphs with bounded expansion is fat. However, first we need to derive the following auxiliary result.

Lemma 1. *Let $c, t \geq 1$ be integers, and let $K_0 = c^{2c}(2c + 2)^{c+1}t^c$. Let G be a bipartite graph with parts S and Z. If all vertices in S have degree at most c, then there exist $Q \subseteq S$ and $X \subseteq Z$ such that $|Q| \geq |S|/K_0$, $|X| \leq |Q|/t$ and for all distinct $u, v \in Q$, all common neighbors of u and v belong to X.*

Proof. For $0 \leq i \leq c$, let $d_i = \dfrac{K_0}{c^{2i+1}(2c + 2)^{i+1}t^i}$. Let Z_0 consist of vertices in Z of degree at least d_0 and for $1 \leq i \leq c$, let Z_i consist of the vertices in Z of degree at least d_i, but less than d_{i-1}. Since all vertices of S have degree at most c, for each $v \in S$ there exists $i \in \{0, \ldots, c\}$ such that v has no neighbors in Z_i. By the pigeonhole principle, there exists $i \in \{0, \ldots, c\}$ and a set $B \subseteq S$ of size at least $|S|/(c + 1)$ such that no vertex of B has a neigbor in Z_i. We set $X = Z_0 \cup \ldots \cup Z_{i-1}$ and we choose Q as an inclusion-wise maximal subset of B such that no two vertices of Q have a common neighbor in $Z \setminus X$.

First, let us estimate the size of Q. For each vertex $v \in B$, its neighbors either belong to X or have degree less than d_i, and thus at most cd_i vertices are at distance exactly two from v in $G - X$. Since Q is maximal, each vertex of $B \setminus Q$ is at distance two from a vertex of Q in $G - X$, and thus

$$|Q| \geq \frac{|S|/(c + 1)}{1 + cd_i} \geq \frac{|S|/(c + 1)}{2cd_i} \geq \frac{|S|}{2c(c + 1)d_0} = \frac{|S|}{K_0} .$$

Note that we use the fact that $cd_i \geq cd_c = 1$.

If $i = 0$, then $X = \emptyset$, and thus Q and X satisfy the conclusions of the lemma. If $i \geq 1$, then we need to estimate the size of X. Note that each vertex of X has degree at least d_{i-1}, and thus $|X|d_{i-1} \leq |E(G)| \leq c|S|$. Consequently,

$$\frac{|Q|}{|X|} \geq \frac{\frac{|S|}{2c(c+1)d_i}}{c|S|/d_{i-1}} = \frac{d_{i-1}}{2c^2(c+1)d_i} = t \ . \qquad \square$$

For a path P, let $\ell(P)$ denote its length (the number of its edges). We say that a path P with directed edges is *reduced* if either $\ell(P) = 1$, or $\ell(P) = 2$ and both of its edges are directed away from the middle vertex of P.

Lemma 2. *Every class of graphs with bounded expansion is fat.*

Proof. Let $\mathcal{G}$ be a class of graphs with bounded expansion and consider fixed integers $d, t \geq 1$. Let $\mathcal{G}_d$ be the class of d-th oriented augmentations of graphs in $\mathcal{G}$, and let m_d be the smallest integer such that the maximum in-degree of every graph in $\mathcal{G}_d$ is at most m_d, which exists by Proposition 1. Let $K_0 = m_d^{2m_d}(2m_d + 2)^{m_d+1}t_d^m$ and let $K = (2m_d + 1)K_0$. We will show that every $G \in \mathcal{G}$ is (d, t, K)-fat.

Consider a set $S \subseteq V(G)$. Let G_d be a d-th oriented augmentation of G and let G'_d be the underlying undirected graph of G_d. Since G_d has maximum in-degree at most m_d, it follows that the maximum average degree of G'_d is at most $2m_d$, and thus G'_d has a proper coloring by at most $2m_d + 1$ colors. By considering the intersections of the color classes of this coloring with S, we conclude that there exists a set $S_0 \subseteq S$ of size at least $|S|/(2m_d + 1)$ which is independent in G'_d. Let Z be the set of in-neighbors of vertices of S_0 in G_d. Let H be the bipartite graph with parts S_0 and Z such that sz is an edge of H if and only if $s \in S_0$, $z \in Z$ and G_d contains an edge directed from z to s. Let Q and X be the sets obtained by applying Lemma 1 to H. Note that $|Q| \geq |S_0|/K_0 \geq |S|/K$ and $|X| \leq |Q|/t$ as required.

It remains to show that Q is d-scattered in $G - X$. Suppose that there exists a path $P_0 \subseteq G - X$ of length at most d between two vertices $u, v \in Q$. For $1 \leq i \leq d - 1$, let G_i denote the intermediate i-th oriented augmentation of G obtained during the construction of G_d. For $1 \leq i \leq d$, let P_i be a path between u and v in the underlying undirected graph of G_i such that $V(P_i) \subseteq V(P_0)$ and P_i is as short as possible. Note that $\ell(P_i) \leq \ell(P_{i-1})$, and if $\ell(P_i) = \ell(P_{i-1})$, then P_i (taken with the orientation of its edges as in G_i) is reduced. Since the length of P_0 is at most d, we conclude that P_d with the orientation as in G_d is reduced. Since Q is an independent set in G'_d, it follows that $\ell(P_d) \neq 1$. Therefore, $\ell(P_d) = 2$ and the middle vertex x of P_d is a common in-neighbor of u and v in G_d. However, this implies that x belongs to X, contrary to the assumption that P is disjoint with X. $\square$

5 Colorings and Independent Sets

Let us now turn our attention back to independent sets. As we mentioned before, if G is a 3-colorable graph on n vertices, then G has an independent set of size at least $n/3$. This bound can be improved when some vertices in the coloring have monochromatic neighborhood, since such vertices can be moved to two different color classes.

Lemma 3 ($\star$). *Let G be a graph on n vertices and let $Q, X \subseteq V(G)$ be disjoint sets. If Q is an independent set of G and $G - X$ has a 3-coloring φ such that the neighborhood of each vertex in Q is monochromatic, then $\alpha(G) \geq \frac{n-|X|+|Q|}{3}$.*

For a plane graph G, let $F(G)$ denote the set of faces of G. Consider a cycle $C \subset G$. Removing C splits the plane into two open connected subsets, the bounded one is called the *open interior* of C. The *closed interior* of C is the closure of the open interior of C. The cycle C is *separating* if both the open interior of C and the complement of the closed interior of C contain a vertex of G. The following result of Dvořák, Král' and Thomas [31] is our main tool for obtaining colorings with monochromatic neighborhoods.

Theorem 4 ([31]). *There exists an integer $D_0 \geq 0$ such that for any triangle-free plane graph G without separating 4-cycles, for any sets $Q_1 \subseteq V(G)$ of vertices of degree at most 4 and $Q_2 \subseteq F(G)$ of 4-faces such that the elements of $Q_1 \cup Q_2$ have pairwise distance at least D_0, and for any 3-coloring ψ of the boundaries of the faces in Q_2, there exists a 3-coloring φ of G such that the neighborhood of each vertex in Q_1 is monochromatic and the pattern of the coloring on each face in Q_2 is the same as in the coloring ψ.*

In the statement, two colorings have the same *pattern* on a subgraph F if they differ on F only by a permutation of colors.

We use the following result by Gimbel and Thomassen [32].

Proposition 2 ([32]). *Let G be a triangle-free planar graph and let $C = v_1v_2\ldots$ be an induced cycle of length at most 6 in G. If a 3-coloring ψ of C does not extend to a 3-coloring of G, then $|C| = 6$ and $\psi(v_1) = \psi(v_4) \neq \psi(v_2) = \psi(v_5) \neq \psi(v_3) = \psi(v_6) \neq \psi(v_1)$.*

Corollary 2 ($\star$). *Let G be a triangle-free planar graph. For any vertex $v \in V(G)$ of degree at most 3, there exists a 3-coloring of G such that the neighborhood of v is monochromatic.*

We need a variation of Theorem 4 which allows some separating 4-cycles. Let G be a plane graph. A subgraph G_0 of G is 4-*swept* if G_0 has no separating 4-cycles and every face of G_0 which is not a face of G has length 4.

Lemma 4. *There exists an integer $D_1 \geq 1$ such that the following holds for any triangle-free plane graph G and any 4-swept subgraph G_0 of G. Let $X, Q \subseteq V(G_0)$ be disjoint sets such that each vertex of Q has degree at most 4 in G_0. If Q is D_1-scattered in $G_0 - X$, then $G - X$ has a 3-coloring such that at least $|Q| - 6|X|$ vertices have monochromatic neighborhood and form an independent set.*

Proof. Let $D_1 = D_0 + 4$, where D_0 is the constant of Theorem 4.

Let R be the set of 4-faces of G_0 which are not faces of G. Let Z be the set of vertices $z \in Q$ such that there exists a face $x_1zx_2v \in R$ such that x_1 and x_2 belong to X. Let H be the graph (possibly with parallel edges) with vertex set X such that two vertices x_1 and x_2 are adjacent if there exists a face $x_1v_1x_2v_2 \in R$, for some $v_1, v_2 \in V(G_0)$. Since G_0 has no separating 4-cycles, we conclude that either G_0 is isomorphic to $K_{2,3}$ and $|X| \geq 2$, or H has no parallel edges. Since H is a planar graph, it follows that $|E(H)| \leq 3|X|$. Note that $|Z| \leq 2|E(H)| \leq 6|X|$.

Let $Q_0 = Q \setminus Z$. Let Q_1 be the set of vertices of Q_0 that are not incident with the faces of R. For each vertex $v \in Q_0 \setminus Q_1$, we choose one incident 4-face in R; let Q_2' be the set of these faces. By the choice of Z, each face in Q_2' is incident with exactly one vertex of $Q_0 \setminus Q_1$, and thus $|Q_2'| = |Q_0 \setminus Q_1|$. For each $f \in Q_2'$, let G_f be the subgraph of G drawn in the closure of f. By Euler's formula, there exists a vertex $v_f \in V(G_f) \setminus V(f)$ whose degree in G is at most 3. Let ψ_f be a 3-coloring of G_f such that the neighborhood of v_f is monochromatic, which exists by Corollary 2. Let $I = \{v_f : f \in Q_2'\}$.

Let G_1 be the graph obtained from $G_0 - X$ as follows. For each face $f \in Q_2'$ whose boundary intersects X, note that by the choice of Z, there exists a subpath P of the boundary walk of f such that $X \cap V(F)$ are exactly the internal vertices of P. Add to G_1 a path of length $|P|$, with the same endvertices as P and with new internal vertices of degree two.

Note that each face in Q_2' corresponds to a 4-face of G_1; let Q_2 be the set of such faces of G_1. Observe that the distance in G_1 between any two elements of $Q_1 \cup Q_2$ is at least D_0. Furthermore, for each $f \in Q_2'$, we can naturally interpret ψ_f as a coloring of the corresponding face of Q_2.

By Theorem 4, there is a 3-coloring φ_0 of G_1 such that the neighborhood of every vertex of Q_1 is monochromatic and the pattern of φ_0 on every face $f \in Q_2$ is the same as the pattern of ψ_f. By permuting the colors in the colorings $\psi_f : f \in Q_2$, we can assume that their restrictions to the boundaries of the faces in Q_2 are equal to the restriction of φ_0. For each 4-face $f \in R \setminus Q_2$, let ψ_f be a 3-coloring of the subgraph of G drawn in the closure of f matching φ_0 on f, which exists by Proposition 2.

Let φ be the union of φ_0 and the colorings ψ_f for all 4-faces f of G_0, restricted to $V(G) \setminus X$. Note that φ is a 3-coloring of $G - X$ such that all vertices in $Q_1 \cup I$ have monochromatic neighborhood in φ. The claim of this lemma holds, since $|Q_1 \cup I| = |Q_0|$. □

6 Proofs

It is well known that planar triangle-free graphs have many vertices of degree at most 4.

Lemma 5 ($\star$). *Every planar triangle-free graph G on n vertices contains at least $n/5$ vertices of degree at most 4.*

For a plane graph G, let $s(G)$ denote the maximum number of vertices of a 4-swept subgraph of G. We can now state the result from which Theorems 2 and 3 will be derived.

Theorem 5. *There exists a constant $c > 0$ such that every plane triangle-free graph G on n vertices has an independent set of size at least $\frac{n+cs(G)}{3}$.*

Proof. Let D_1 be the distance from Lemma 4. By Lemma 2, there exists a constant K such that every planar graph is $(D_1, 14, K)$-fat. Let $c = \frac{1}{10K}$.

Let G_0 be a 4-swept subgraph of G such that $|V(G_0)| = s(G)$. Let S be the set of vertices of G_0 of degree at most 4; by Lemma 5, we have $|S| \geq s(G)/5$. Since G_0 is $(D_1, 14, K)$-fat, there exist sets $Q \subseteq S$ and $X \subseteq V(G_0) \setminus Q$ such that Q is D_1-scattered in $G_0 - X$, $|Q| \geq |S|/K \geq \frac{s(G)}{5K}$, and $|X| \leq |Q|/14$.

By Lemma 4, $G - X$ has a proper 3-coloring and an independent set Q' with $|Q'| \geq |Q| - 6|X|$ such that the neighborhood of each vertex in Q' is monochromatic. By Lemma 3, we have, as required,

$$\alpha(G) \geq \frac{n - |X| + |Q| - 6|X|}{3} \geq \frac{n + |Q|/2}{3} \geq \frac{n + \frac{s(G)}{10K}}{3}\ . \qquad \square$$

Proof (of Theorem 3). Since G has girth at least 5, we have $s(G) = n$. Therefore, $\alpha(G) \geq \frac{n+cn}{3} = \frac{n}{3/(1+c)}$ by Theorem 5, and we can set $\varepsilon = \frac{3c}{1+c}$. $\square$

Theorem 2 is a corollary of Theorem 5 and the following observation.

Lemma 6. *Every plane graph G has tree-width at most $41\sqrt{s(G)}$.*

Proof. We obtain a tree decomposition $(T, \mathcal{B})$ of G as follows. Let $\mathcal{C}$ be a set of separating 4-cycles in G that have disjoint open interiors, and such that the union of their open interiors is maximal. Let B_0 be the set of vertices of G that are not contained in the open interiors of cycles in $\mathcal{C}$. Recursively, we apply the same procedure to each subgraph of G drawn in the closed interior of a cycle $C \in \mathcal{C}$, obtaining its tree decomposition $(T_C, \mathcal{B}_C)$. We let T be the tree obtained from the trees $T_C \mid C \in \mathcal{C}$ by adding B_0 as a neighbor of their roots, and we set $\mathcal{B} = \{B_0\} \cup \bigcup_{C\in\mathcal{C}} \mathcal{B}_C$. Note that each bag $B \in \mathcal{B}$ induces a 4-swept subgraph of G, and thus its size is at most $s(G)$.

Consider a bag $B \in \mathcal{B}$ and let $H = G[B]$. Let R be the set of faces of H which are not faces of G. Note that for every bag $B' \neq B$, the intersection $B \cap B'$ is a subset of the vertices of some face of R. Let H' be obtained from H by adding, for each face $f \in R$, a vertex v_f adjacent to all vertices of f. Note that H is triangle-free, and thus it has at most $|B|$ faces. Consequently, $|V(H')| \leq 2|B|$. Since H' is planar, it has tree-width at most $6\sqrt{|V(H')|} + 1$ by a result of Robertson, Seymour and Thomas [33]. Thus, H' has a tree decomposition $\mathcal{T}'_B$ with all bags of size at most $6\sqrt{|V(H')|} + 2$. Let $\mathcal{T}_B$ be the tree decomposition of H obtained from $\mathcal{T}'_B$ by, for each bag of the decomposition and each $f \in R$, replacing v_f by $V(f)$. The bags of the decomposition $\mathcal{T}_B$ have size at most $24\sqrt{|V(H')|} + 8 \leq 24\sqrt{2|B|} + 8$. Furthermore, for each $f \in R$, there exists a bag of $\mathcal{T}_B$ containing $V(f)$.

It follows that we can combine the decompositions $\mathcal{T}_B$, $B \in \mathcal{B}$, to a decomposition of G of width at most $24\sqrt{2s(G)} + 7 \leq 41\sqrt{s(G)}$. □

7 Discussion

We gave a fixed-parameter algorithm for finding an independent set of size at least $n/3 + k$ in triangle-free planar graphs on n vertices, for every integer $k \geq 0$. Let us remark that the subexponential dependence on k in the running time of our algorithm is optimal, under the Exponential Time Hypothesis (this follows from a reduction by Madhavan [3]).

Several intriguing questions remain. Does the problem admit a polynomial kernel? That is, can any triangle-free planar graph on n vertices be efficiently (in polynomial time) compressed to an equivalent graph G' on $k^{O(1)}$ vertices? Also, while we can decide the existence of the independent set in linear time (in n), we can only find such an independent set in quadratic time. Can this be improved?

Unfortunately, it is unlikely our techniques could be used for Planar Independent Set-ATLB in general planar graphs. The analogue of Proposition 4 is false for general planar graphs, and there exist n-vertex planar graphs with largest independent set of size $n/4$ and arbitrarily large tree-width.

References

1. Appel, K., Haken, W.: Every planar map is four colorable. Bull. Amer. Math. Soc. 82, 711–712 (1976)
2. Robertson, N., Sanders, D., Seymour, P., Thomas, R.: The four-colour theorem. J. Combin. Theory Ser. B 70, 2–44 (1997)
3. Madhavan, C.: Approximation algorithm for maximum independent set in planar traingle-free graphs. In: Joseph, M., Shyamasundar, R.K. (eds.) FSTTCS 1984. LNCS, vol. 181, pp. 381–392. Springer, Heidelberg (1984)
4. Niedermeier, R.: Invitation to fixed-parameter algorithms. Oxford Lecture Series in Mathematics and its Applications, vol. 31. Oxford University Press, Oxford (2006)
5. Bodlaender, H., Demaine, E., Fellows, M.R., Guo, J., Hermelin, D., Lokshtanov, D., Müller, M., Raman, V., van Rooij, J., Rosamond, F.A.: Open problems in parameterized and exact computation. Technical report, Utrecht University (2008)
6. Mahajan, M., Raman, V., Sikdar, S.: Parameterizing above or below guaranteed values. J. Comput. System Sci. 75, 137–153 (2009)
7. Sikdar, S.: Parameterizing from the extremes. PhD thesis, The Institute of Mathematical Sciences, Chennai (2010)
8. Mnich, M.: Algorithms in Moderately Exponential Time. PhD thesis, TU Eindhoven (2010)
9. Crowston, R., Fellows, M., Gutin, G., Jones, M., Rosamond, F., Thomassé, S., Yeo, A.: Simultaneously Satisfying Linear Equations Over $\mathbb{F}_2$: MaxLin2 and Max-r-Lin2 Parameterized Above Average. In: Proc. FSTTCS 2011. LIPIcs, vol. 13, pp. 229–240 (2011)
10. Fellows, M.R., Guo, J., Marx, D., Saurabh, S.: Data Reduction and Problem Kernels (Dagstuhl Seminar 12241). Dagstuhl Reports 2, 26–50 (2012)

11. Grötzsch, H.: Zur Theorie der diskreten Gebilde. VII. Ein Dreifarbensatz für dreikreisfreie Netze auf der Kugel. Wiss. Z. Martin-Luther-Univ. Halle-Wittenberg. Math.-Nat. Reihe 8(/1959), 109–120 (1958/1959)
12. Dvořák, Z., Kawarabayashi, K.I., Thomas, R.: Three-coloring triangle-free planar graphs in linear time. ACM Trans. Algorithms 7, Art. 41, 14 (2011)
13. Steinberg, R., Tovey, C.A.: Planar Ramsey numbers. J. Combin. Theory Ser. B 59, 288–296 (1993)
14. Jones, K.F.: Minimum independence graphs with maximum degree four. In: Graphs and Applications, Boulder, Colo., pp. 221–230. Wiley-Intersci. Publ. (1985)
15. Bodlaender, H.L., Drange, P.G., Dregi, M.S., Fomin, F.V., Lokshtanov, D., Pilipczuk, M.: A $O(c^k n)$ 5-approximation algorithm for treewidth. In: Proc. FOCS 2013, pp. 499–508 (2013)
16. Scheinerman, E.R., Ullman, D.H.: Fractional graph theory. Dover Publications Inc., Mineola (2011)
17. Dvořák, Z., Škrekovski, R., Valla, T.: Planar graphs of odd-girth at least 9 are homomorphic to the Petersen graph. SIAM J. Discrete Math. 22, 568–591 (2008)
18. Pirnazar, A., Ullman, D.H.: Girth and fractional chromatic number of planar graphs. J. Graph Theory 39, 201–217 (2002)
19. Djidjev, H.N.: On some properties of nonplanar graphs. Compt. Rend. Acad. Bulg. Sci. 37, 1183–1185 (1984)
20. Kim, J.H.: The Ramsey number $R(3, t)$ has order of magnitude $t^2/\log t$. Random Structures Algorithms 7, 173–207 (1995)
21. Staton, W.: Some Ramsey-type numbers and the independence ratio. Trans. Amer. Math. Soc. 256, 353–370 (1979)
22. Heckman, C.C., Thomas, R.: Independent sets in triangle-free cubic planar graphs. J. Comb. Theory, Ser. B 96, 253–275 (2006)
23. Alon, N., Gutin, G., Kim, E., Szeider, S., Yeo, A.: Solving MAX-r-SAT above a tight lower bound. Algorithmica 61, 638–655 (2011)
24. Gutin, G., Kim, E.J., Mnich, M., Yeo, A.: Betweenness parameterized above tight lower bound. J. Comput. System Sci. 76, 872–878 (2010)
25. Gutin, G., van Iersel, L., Mnich, M., Yeo, A.: Every ternary permutation constraint satisfaction problem parameterized above average has a kernel with a quadratic number of variables. J. Comput. System Sci. 78, 151–163 (2012)
26. Crowston, R., Jones, M., Mnich, M.: Max-cut parameterized above the Edwards-Erdős bound. In: Czumaj, A., Mehlhorn, K., Pitts, A., Wattenhofer, R. (eds.) ICALP 2012, Part I. LNCS, vol. 7391, pp. 242–253. Springer, Heidelberg (2012)
27. Mnich, M., Zenklusen, R.: Bisections above tight lower bounds. In: Golumbic, M.C., Stern, M., Levy, A., Morgenstern, G. (eds.) WG 2012. LNCS, vol. 7551, pp. 184–193. Springer, Heidelberg (2012)
28. Cygan, M., Pilipczuk, M., Pilipczuk, M., Wojtaszczyk, J.O.: On multiway cut parameterized above lower bounds. ACM Trans. Comput. Theory 5, 3:1–3:11 (2013)
29. Nešetřil, J., Ossona de Mendez, P.: Grad and classes with bounded expansion. I. Decompositions. European J. Combin. 29, 760–776 (2008)
30. Nešetřil, J., Ossona de Mendez, P.: Sparsity – Graphs, Structures, and Algorithms. Springer (2012)
31. Dvořák, Z., Král', D., Thomas, R.: Three-coloring triangle-free graphs on surfaces V. Coloring planar graphs with distant anomalies (2013) (manuscript)
32. Gimbel, J., Thomassen, C.: Coloring graphs with fixed genus and girth. Trans. Amer. Math. Soc. 349, 4555–4564 (1997)
33. Robertson, N., Seymour, P., Thomas, R.: Quickly excluding a planar graph. J. Combin. Theory Ser. B 62, 323–348 (1994)

GRASP. Extending Graph Separators for the Single-Source Shortest-Path Problem

Alexandros Efentakis[1] and Dieter Pfoser[2]

[1] Research Center "Athena"
efentakis@imis.athena-innovation.gr
[2] Department of Geography and GeoInformation Science, George Mason University
dpfoser@gmu.edu

Abstract. Many existing solutions focus on point-to-point shortest-path queries in road networks. In contrast, only few contributions address the related single-source shortest-path problem, i.e., finding shortest-path distances from a single source s to all other graph vertices. This work extends graph separator methods to handle this specific problem and its one-to-many variant, i.e., calculating the shortest path distances from a single source to a set of targets $T \subseteq V$. This novel family of so-called GRASP algorithms provides exceptional preprocessing times, making them suitable for dynamic travel time scenarios. GRASP algorithms also efficiently solve range / isochrone queries not handled by previous approaches.

Keywords: Shortest-path computation, GRASP algorithm, Range queries, Isochrones.

1 Introduction

For the single-pair shortest-path problem (SPSP) in road networks, several techniques can be much faster than the Dijkstra algorithm by using a two-phase approach: *Preprocessing* requires a few minutes (or hours) and produces auxiliary data that is subsequently used to accelerate shortest-path (SP) queries. The related research has been so rapidly evolving that even recent surveys [6] had to be updated in later publications [1].

While many efficient techniques exist, an important category of SP algorithms is based on graph separators (GS). The most prominent example of this category is Customizable Route Planning (CRP) [4,8]. Although CRP is one order of magnitude slower than hierarchical approaches such as Contraction Hierarchies (CH) [14] it is still fast enough for real-time services and offers multiple advantages: (i) It offers very fast preprocessing times of few seconds for continental road networks, making this method suitable for dynamic road networks, (ii) it is very resilient to the metric used, including travel distances and turning costs. This is the reason why global Mapping services such as Bing Maps prefer to use CRP, over faster but less practical solutions such as CH.

In the case of the single-source shortest-path problem (SSSP), given a source vertex s, the goal is to find SP distances from s to *all other graph vertices*. This problem is also addressed by the classic Dijkstra algorithm. Quite recently, [3] introduced the PHAST algorithm that, compared to Dijkstra, is orders of magnitude faster. Later works

[5] presented RPHAST for solving the one-to-many variant: given a set of targets T, compute the distances between s and all vertices in T.

Since both PHAST and RPHAST are extensions of the CH algorithm, their preprocessing is slow, making those methods practically unsuitable for dynamic scenarios. Moreover, since CH is essentially a bidirectional method, when applied to the SSSP problem, it may only be used if we know the target nodes in advance, as in the entire road network in PHAST, or a subset T of nodes in RPHAST. Therefore, these methods cannot be extended to *range queries*, i.e., find all nodes reachable from s within a given timespan) or *isochrone queries*, i.e., find all nodes AND edges reachable from s within a given timespan, since for those queries we do not know the target nodes in advance.

In this work, our goal is to create SP methods that (i) are very fast, (ii) have very short pre-processing times and (iii) may be used for most (if not all) SSSP cases. We achieve this by extending Graph Separators methods and create a novel set of algorithms named GRASP (Graph separators, RAnge, Shortest Path) that have none of the inherent shortcomings of previous approaches. All GRASP algorithms (i) have very short preprocessing time of few seconds, making them suitable for dynamic road networks, i.e., road networks with changing edge weights due to traffic updates, (ii) are very robust with respect to the metric used (either travel times or travel distances), and most of all, (iii) they may efficiently solve range/isochrone queries. As recent works have suggested (cf. [10,11]), this type of queries is very important for a spectrum of application contexts, including fleet management, urban planning and geomarketing.

The outline of this work is as follows. Section 2 describes previous work related to our method. Section 3 describes our family of GRASP algorithms and their implementation. Experiments establishing the benefits of our approach are provided in Section 4. Finally, Section 5 gives conclusions and directions for future work.

2 Related work

In this work we focus on directed weighted graphs $G(V, E, w)$, where V is a finite set of vertices, $E \subseteq V\mathrm{x}V$ are the arcs of the graph and w is a positive weight function $E \longrightarrow R^+$. A partition of V is a family $C = \{c_0, c_1, \dots c_L\}$ of sets, such that each node $u \in V$ is contained in exactly one set c_i. An element of a partition is called a *cell*. A multilevel partition of V is a family of partitions $\{C^0, C^1, \dots C^L\}$ where ℓ denotes the level of a partition C^ℓ. Similar to [4], level 0 refers to the original graph, L is the highest partition level and we use *nested multilevel partitions*, i.e., for each $\ell < L$ and each cell c_i^ℓ there exists a unique cell $c_j^{\ell+1}$ (called the supercell of c_i^ℓ) with $c_i^\ell \subseteq c_j^{\ell+1}$. Accordingly, c_i^ℓ is a subcell of $c_j^{\ell+1}$. According to this notation, $c^\ell(v)$ is the cell that contains the vertex v on level ℓ. Moreover, the number of cells of the partition C^ℓ will be denoted as $|C^\ell|$. A boundary arc on level ℓ is an arc with start and end vertices located in different level ℓ cells; a boundary vertex on level ℓ is a vertex which is connected with at least one neighbor in another level-ℓ cell. Note that for *nested multilevel partitions*, a boundary vertex / arc at level ℓ is also a boundary vertex / arc for all levels below.

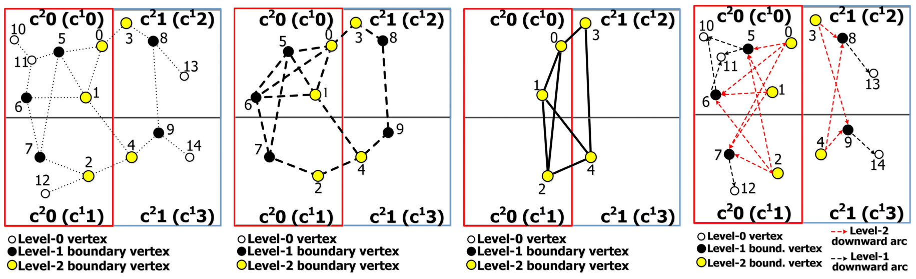
Contraction Hierarchies. Contraction hierarchies (CH) [14] efficiently solve the SPSP problem on road networks. During preprocessing, CH picks an ordering of the vertices and shortcuts them according to this order, i.e., a shortcut is created between each pair u, w of neighboring vertices of v, if the shortest path from u to w is unique and contains v. The final output of the CH preprocessing routine is the set E^+ of shortcut edges and the position of each vertex v in the node ordering (denoted by $rank(v)$).

If $E\uparrow = \{(v, w) \in E \cup E^+ : rank(v) < rank(w)\}$ and $E\downarrow = \{(v, w) \in E \cup E^+ : rank(v) > rank(w)\}$, then the CH algorithm runs a bidirectional Dijkstra algorithm, in which the forward search is restricted to $G\uparrow = (V, E\uparrow)$ and the reverse search to $G\downarrow = (V, E\downarrow)$. As showed by [14], if u is the maximum-rank vertex that minimizes $d_s(u) + d_t(u)$, then the actual shortest path from s to t is given by the concatenation of $s \to u$ and $u \to t$ paths.

PHAST & RPHAST. In the single-source shortest-path problem (SSSP) or the *one-to-all* problem, given a source vertex s, the goal is to find SP distances from s to all other graph vertices. The PHAST algorithm [3] extended CH to efficiently answer SSSP queries. The preprocessing phase of PHAST is similar to CH and outputs a set of shortcuts E^+ and a vertex ordering. A PHAST query initially sets $d(v)=\infty$ for all $v \neq s$, and $d(s)=0$. It then executes the actual SSSP query in two subphases. First, it performs a simple forward CH search (denoted hereafter as the *upward phase*): it runs Dijkstra algorithm from s in $G\uparrow$, stopping when the priority queue becomes empty. This sets the distance labels $d(v)$ of all vertices visited by the search. The second (scanning) subphase scans all vertices in $G\downarrow$ in descending rank order, i.e., vertices on level ℓ are only visited after all vertices on levels greater than ℓ have been processed. To scan v, the PHAST algorithm examines each incoming arc $(u, v) \in E\downarrow$; if $d(v) > d(u) + w(u, v)$, PHAST sets $d(v) = d(u) + w(u, v)$. The main advantage of PHAST is that only the (cheap) upward phase depends on the source s. The expensive scanning phase a) visits all vertices and arcs in the same order regardless of the source and b) vertices belonging to the same CH level may be processed in parallel. As a result, parallel PHAST requires merely *39-64ms* for continental road networks .

Later, [5] introduced the RPHAST algorithm for solving the *one-to-many* variant of finding SP distances from the source s to set of targets $T \subseteq V$. RPHAST uses the exact same CH preprocessing but it has an additional target selection phase, in which RPHAST extracts from $G\downarrow$ the information necessary to compute the distances from any source $s \in V$ to all targets T, creating a restricted downward graph denoted as $G_T^{\downarrow}$. RPHAST then runs the same query phase as PHAST but using $G_T^{\downarrow}$ instead of $G\downarrow$.

Graph Separators. In Graph Separator (GS) methods, such as CRP [4], a partition C of the graph is computed. Then, the preprocessing stage builds a graph H containing all boundary nodes and boundary arcs of G. It also contains a clique for each cell c: for every pair (u, v) of boundary nodes in c, a clique arc (u, v) is created whose cost is the same as the shortest path (restricted to the inner edges of c) between u and v. For a SP query between s and t, Dijkstra's algorithm must be run on the graph consisting of the union of H, $c_0(s)$ and $c_0(t)$. To accelerate queries, multiple levels of overlay graphs may be used. Since each clique is calculated by using only the inner edges of c, GS preprocessing may be easily parallelized. Moreover, overlay graphs of higher level partitions may be computed by using the overlay graphs of lower levels to further reduce preprocessing time. By using those two optimizations, CRP is the most efficient



(a) A graph G. $|V|= 15$, L=2, $|C^L|$=2 (b) Level-1 overlay graph (c) Level-2 overlay graph (d) The $G_{GS}\downarrow$ graph

Fig. 1. Overlay graphs and the $G_{GS}\downarrow$ graph for an example graph G

SPSP algorithm in terms of preprocessing time (requiring few seconds for continental road networks) and is thus suitable for dynamic road networks.

Recently, [7] adjusted CRP to handle *one-to-many* queries. Unfortunately, results were not impressive: Even with an indexing scheme, the target selection phase of CRP is still 32 times worse than RPHAST. Thus, CRP cannot efficiently answer SSSP queries.

3 The GRASP Algorithm

In this work we extend graph separator (GS) methods to efficiently solve all variations of the SSSP problem. Our novel family of algorithms denoted GRASP have all the advantages of graph separator methods: Extremely fast preprocessing times, fast SP query performance and robustness to the metric used. But they also efficiently answer range / isochrone queries not addressed by previous CH based methods.

In GS techniques, during preprocessing, we compute SP distances (restricted to the inner arcs) between the border vertices of each cell, at each partition level. As shown in [4], the SP distances between (level-ℓ) border vertices of cell c^ℓ may be calculated by running one Dijkstra search per border vertex, in the union of all subcells of c^ℓ at level $\ell-1$. Since those searches use the level $\ell-1$ overlay graph, they are extremely fast.

Extending this observation, GRASP preprocessing (by using the exact same Dijkstra searches as before) additionally computes the SP distances between all border vertices of level ℓ and all vertices of level $\ell-1$ within each cell c^ℓ. To differentiate between the two kind of arcs calculated by the same search, we will denote as (i) *clique arcs* those added overlay arcs that connect border vertices of the same level ℓ and (ii) *downward arcs* of level ℓ those connecting vertices at different levels, i.e., ℓ and $\ell-1$. Adopting the notation of [4], the *overlay graph* containing border vertices, border arcs and clique arcs will be denoted hereafter as H. For added efficiency, *downward arcs* are stored as a separate graph (using a single adjacency array representation [16]), hereafter referred to as $G_{GS}\downarrow$. Since we use nested partitions, H and $G_{GS}\downarrow$ do not necessarily need to have the same number of levels. The intermediate steps for building the overlay graph H and the $G_{GS}\downarrow$ for a sample graph G, are shown in Fig. 1.

When GRASP executes a SSSP query from source s, it runs a simple Dijkstra search in the the union of the cell $c^0(s)$ and H until the priority queue is empty. This will be referred to as the *upward phase* of GRASP. The upward phase settles all border vertices of level L, all vertices of $c^0(s)$ and some additional vertices inside $c^L(s)$. Note that, *all vertices settled by the upward phase are assigned correct SP distances from s*, due to the definition of the overlay graphs. For a random cell c^L by using the level-L downward arcs (inside c^L) we may calculate correct SP distances to all level-L-1 border vertices from s. Recursively, in descending order of GS levels, we may calculate correct SP distances for all vertices inside c^L. This process is repeated for all (level-L) cells. This second stage is denoted as the *scanning phase* of GRASP (see Procedure GRASP).

```
GRASP($s, d(V), nsVec(V), G, H, G_{GS}\downarrow$)
1  Init $nsVec(V)$ to FALSE for $level - L$
2  Dijkstra($s, c_0(s) \cup H, d(V), nsVec(V)$)
3  **parallel for** each $cell$ in $C^L$ partition
4      **for** $level = L - 1$ **to** 0
5          **for** vertex $v$ of $level$ in $cell$
6              **if** $nsVec[v]$ == FALSE
7                  Scan($v$, $d(V)$, $G_{GS}\downarrow$)
8              **else** $nsVec[v]$ = FALSE
```

```
SCAN($v, d(V), G_{GS}\downarrow$)
1  **for** $edge$ in $G_{GS}\downarrow$ incoming to $v$
2      $tl = edge.tail$
3      **if** $d[v] > d[tl] + edge.weight$
4          $d[v] = d[tl] + edge.weight$
```

Theorem 1. *The GRASP algorithm is correct (Proof omitted due to space restrictions)*

3.1 GRASP Tuning

Although GRASP's simplicity and correctness is evident by its definition,we need to take several steps to further improve its query performance. Those steps include:

Initialization. All Dijkstra based variants assume that distance labels for all vertices are set to ∞ during initialization. This requires a linear sweep over all distance labels, which can be slow. To improve speed and to avoid scanning a considerably percentage of downward arcs, GRASP uses a bit vector of size $|V|$, termed *nodeScannedVector* (cf. [5]). In the GRASP upward phase, visited vertices have their associated bit set to true and correct distance labels assigned. During the scanning phase, GRASP avoids scanning vertices with set bits (line 6 of procedure GRASP) and resets their associated bit (line 8) for subsequent searches. This avoids scanning a considerably percentage of downward arcs and saves computation time.

Number of Levels. With respect to the required levels of overlay graphs, four levels of overlay graphs suffice to achieve fast SPSP query times [4]. In our case, minimizing the number of downward arcs for $G_{GS}\downarrow$ requires as many intermediate levels as possible. Experimentation showed that $L = 16$ yields best results. For the upward phase of GRASP and overlay graph H we need to only "use" four of those levels (namely: C^4, C^9, C^{12}, C^{16}). For point-to-point SP queries, typically performance improves with a decreasing number of $|C^L|$ cells at the highest level partition. However in our case, decreasing $|C^L|$ creates a larger number of downward arcs in $G_{GS}\downarrow$ and limits

GRASP's parallel performance. Thus, we get optimal results for medium number of cells at the highest level partition, experimentally established to be $|C^L| = 128$.

Nodes Reordering. To improve performance, we *reorder the vertices of graph G* (cf. [2]), such that overlay vertices are assigned smaller IDs (ordered by descending level), breaking ties with cell. Non-border vertices are ordered by their level-1 cells. In addition, within the same cell (and level) we order vertices by a DFS layout similar to [12]. This nodes' reordering improves (i) spatial locality for preprocessing and (ii) both the upward and scanning phases' performance of GRASP.

Arc Reduction. Previous GS approaches [4,2] compute all available overlay arcs between border vertices. This would be very wasteful for GRASP, especially for the downward arcs. For GRASP, we extend the arc-reduction optimization of [13], which during preprocessing, reports only distances of boundary nodes that are direct descendants of the root of each executed Dijkstra algorithm. This optimization preserves correct distances between boundary vertices on the overlay graph and leads to a 56-71% reduction in the number of arcs created (depending on the cell size). Although originally used for clique arcs, it works even better for downward arcs. This optimization has (i) no negative impact on preprocessing time, since it is integrated in each Dijkstra search, (ii) creates fewer arcs (both clique and downward) and therefore makes GRASP less memory intensive, (iii) arc-reduction is achieved individually per cell (each cell computation is still independent from other cells), which is especially important in cases where traffic updates are restricted to a limited number of cells.

Parallelization. A final performance boost comes from exploiting the parallel nature of GRASP. For a single-tree computation, PHAST requires to pause and synchronize threads at each CH level. Since the number of CH levels is quite large, (140 in the case of continent-size networks), parallel performance is not optimal. Contrarily, GRASP only has 16 GS levels. This allows for better parallel scaling of GRASP. Still, we can do even better. By definition, each level-L cell may be processed independently - see line 3 of procedure GRASP. Thus, the parallel implementation of GRASP requires no intermediate barriers for a single-tree computation and consequently scales much better for a large number of cores. As will be shown in Section 4, parallel GRASP is as fast as parallel PHAST, while requiring only a fraction of PHAST's preprocessing time. As a result, GRASP is the most competitive solution for answering SSSP queries for dynamic (time-dependent) road networks.

3.2 Range and Isochrone Queries

Another variation of the SSSP problem, is Range / Isochrone Queries. *Range queries* identify and assign correct SP distances to all vertices reachable from a source s within a given range (either travel times or distance) limit e. *Isochrone queries* find all nodes *and* edges reachable from s within a given limit e. They are an

```
RangeToIsoc($e, d(V), nsVec(V), esVec(E), G$)
1  for vertex $v = 0$ to $|V| - 1$
2      if $nsVec(V) == $ TRUE
3          for edge in $G$ outgoing from $v$
4              if $d[v] + edge.weight \leq e$
5                  $esVec[edge] = $ TRUE
```

extension of range queries, which simply requires an additional linear sweep over the original graph adjacency array of G to discover which edges are reachable from s within the given range. This process (described by the procedure RangeToIsoc) is very fast and requires less than few ms for continental road networks. Unfortunately, this type of queries cannot be handled by previous approaches, such as PHAST or RPHAST due to the inherent bidirectional nature of CH, which requires to know the target vertices in advance. In the following, we will describe how our novel isoGRASP algorithm (i.e., isochrone GRASP, an adaptation of GRASP) efficiently solves range queries.

Range queries mark and assign correct SP distances to all vertices reachable within the limit e. To this end, isoGRASP uses the nodeScannedVector (similar to GRASP). The isoGRASP algorithm uses the exact same preprocessing as before, i.e., building the H and $G_{GS}\downarrow$ graphs and also has an upward and a scanning phase. In the upward phase, isoGRASP runs a *RangeDijkstra* search in the union of $c_0(s)$ and H, which is a modified Dijkstra algorithm that only allows vertices u with SP distance $d(u) \leq e$ to enter the priority queue. As a result, the upward phase of isoGRASP terminates early and settles only those level-L border vertices and some additional vertices inside $c^L(s)$ that are reachable from source within the specified limit e. Moreover, during the upward phase, isoGRASP marks those level-L cells which are reachable from source. To achieve this, we use an additional bit vector of size $|C^L|$ (128 in our experiments), denoted hereafter as the *cellScannedVector* (and $csVec(C^L)$ in the pseudocode).

During isoGRASP's scanning phase, by utilizing the cellScannedVector, we restrict calculations to those level-L cells reachable from source within the limit e. This saves a lot of computation time for smaller values of e. Contrary to GRASP, when we scan a node u, we only look at downward arcs where the tail vertex v of this arc has its associated nodeScannedVector bit set to true, i.e, the tail v of this arc is reachable within e. If $d(v)+w(v,u) \leq e$ and $d(v)+w(v,u) < d(u)$ then $d(u) = d(v)+w(v,u)$ and the node-ScannedVector bit of u is set to true. Thus, after isoGRASP's scanning phase, all vertices reachable from a source within e are assigned correct SP distances and have their nodeScannedVector bit set to true (see procedure isoGRASP).

Lemma 1. *The isoGRASP algorithm is correct* (Proof omitted due to space restrictions)

```
isoGRASP ($s, e, d(V), nsVec(V), csVec(C^L)$, $G, H, G_{GS}\downarrow$)
1  Initialize $nsVec(V)$ to FALSE
2  Initialze $csVec(C^L)$ to FALSE
3  RangeDijkstra($s, c_0(s) \cup H, d(V)$,
               $nsVec(V), csVec(C^L)$)
4  parallel for each cell in $C^L$ partition
5      if $csVec[cell]$ == TRUE
6          for $level = L - 1$ to 0
7              for vertex $v$ of $level$ in $cell$
8                  if $nsVec[v]$==FALSE
9                      RScan($v, e, d(V)$,
                       $nsVec(V), G_{GS}\downarrow$)
```

```
RScan($v, e, d(V), nsVec(V), G_{GS}\downarrow$)
1  for edge in $G_{GS}\downarrow$ incoming to $v$
2      $tl = edge.tail$
3      if $nsVec(tl) == true$
4          if $d[v] > d[tl]+edge.weight$
            & $d[tl]+edge.weight \leq e$
5              $d[v] = d[tl]+edge.weight$
6              $nsVec[v]$ = TRUE
```

3.3 One-to-Many Queries

The *one-to-many* SSSP variant finds SP distances from a source s to a non-empty set of targets $T \subseteq V$. We call the respective algorithm *reGRASP* (restricted GRASP). Similar to [5], reGRASP has a *target selection phase*, which only depends on the targets' set T. During the *target selection phase*, reGRASP marks the vertices T' ($T \subseteq T' \subseteq V$) that are necessary for computing SP distances to all vertices $\in T$.

Here, we use a new bit vector of size $|V|$, the *restrictedVector* (denoted as $rVec(V)$ in the pseudocode). All its bits are set to false except those referring to T vertices IDs. Then we sweep this vector from the higher bits to the lower. When we meet a marked vertex (true bit), by using the adjacency array representation of the $G_{GS}\downarrow$ graph we mark the vertices that need to be added to T'. Since, downward arcs connect higher level vertices (which correspond to smaller IDs, according to our nodes reordering) with lower level vertices (with larger IDs), each vertex needs to be scanned only once.

We may further accelerate the target selection phase by using again the *cellScanned-Vector*. By the definition of $G_{GS}\downarrow$, we know that all vertices belonging to T' belong to the same level-L cells as T vertices. Therefore at step 1 of the target selection process, we mark the corresponding bits of level-L cells of vertices belonging to T to true. Therefore during this phase, we may safely ignore level-L cells with their corresponding cell bits set to false. Moreover, each such cell may be processed in parallel since the $G_{GS}\downarrow$ graph only connects vertices belonging to the same level-L cell. This is a major advantage of GRASP over RPHAST, where the respective time-consuming target selection phase cannot be parallelized. Procedure TS shows the finalized pseudocode.

```
reGRASP($s, T, d(V), nsVec(V), csVec(C^L), rVec(V), G, H, G_{GS}\downarrow$)
1  TS($T, csVec(C^L), rVec(V), G_{GS}\downarrow$)
2  Init $nsVec(V)$ to FALSE for $level - L$
3  Dijkstra($s, c_0(s) \cup H, d(V), nsVec(V)$)
4  parallel for each $cell$ in $C^L$ partition
5      if $csVec[cell]$ == TRUE
6          for $level = L - 1$ to 0
7              for vertex $v$ of $level$ in $cell$
8                  if $nsVec[v]$==FALSE
                   & $rVec[v]$==TRUE
9                      Scan($v, d(V), G_{GS}\downarrow$)
10                 else
11                     $nsVec[v]$ = FALSE
```

```
TS($T, csVec(C^L), rVec(V), G_{GS}\downarrow$)
1  Initialze $rVec(V)$ to FALSE
2  Initialze $csVec(C^L)$ to FALSE
3  for each vertex $t$ in $T$
4      $rVec[t]$ = TRUE
5      $csVec[c^L(t)]$ = TRUE
6  parallel for each $cell$ in $C^L$ partition
7      if $csVec[cell]$ == TRUE
           // Skip level-L
8          for $level = 0$ to $L - 1$
9              for vertex $v$ of $level$ in $cell$
10                 if $rVec[v]$ == TRUE
11                     for $edge$ in $G_{GS}\downarrow$
                       incoming to $v$
12                         $tl = edge.tail$
13                         $rVec[tl]$ = TRUE
```

After the target selection phase, reGRASP has an upward and a scanning phase. The upward phase is exactly the same as GRASP and settles all level-L border vertices and some additional nodes inside $c^L(s)$. Then during the scanning phase (i) we ignore cells with their *cellScannedVector* bits set to false and (ii) we do not scan vertices not belonging to T' (i.e, have their restrictedVector bits set to false). Again, during the scanning phase each level-L cell may be processed in parallel (see Procedure reGRASP).

Lemma 2. *The reGRASP algorithm is correct (Proof omitted due to space restrictions)*

The main advantage of reGRASP over RPHAST, is that (i) we may ignore level-L cells that do not contain T vertices and (ii) that *each level-L cell may be processed in parallel, even during the target selection phase*. This way, parallel reGRASP always exhibits excellent performance.

Conclusively, the most important aspect and outcome of our approach is not three distinct algorithms but a unified framework that solves all variants of the SSSP problem. The exact same preprocessing and the same data structures (the overlay graph H and the $G_{GS}\downarrow$ graph) suffice to solve all these different variations. In this scenario, building a server that concurrently answers all types of queries (one-to-all, range/isochrone and one-to-many) as they arrive from clients, is possible for the first time.

4 Experiments

The scope of our experiments is to evaluate the performance of all GRASP algorithms for different SSSP variations. The experiments were performed on a workstation with a 4-core Intel i7-3770 processor clocked at 3.4GHz and 32 GB of RAM, running Ubuntu 12.10. Our code was written in C++ with GCC 4.7 and optimization level 3. We used OpenMP for parallelization. We used the the European and the full USA road networks (18M nodes / 42M arcs and 24M nodes / 58M arcs respectively) made available from the 9th DIMACS Implementation Challenge [9]. We experimented with both travel times and travel distances. For partitioning the graph into nested-multilevel partitions, similarly to [12], we used Buffoon / KaFFPa [17] in a top-down approach. For the lowest four level partitions ($C^1, \ldots C^4$), we switched to METIS [15] for faster computation. For both benchmark road networks used, we used a total of 16 partitioning levels (i.e., $L = 16$) and the C^{16} partition contains 128 cells. Each partition below that, contains double the cells of the previous highest level partition (i.e., C^{15} has 256 cells, C^{14} has 512 cells and so-on).

To compare the performance to existing approaches, authors of PHAST / RPHAST have kindly conducted all experiments for the same road networks on a similar workstation setup to ours (they could not provide direct access to source code due to IPR claims). Their setup is a workstation with an Intel i7-3770K (almost identical to ours, except it is clocked higher at 3.5GHz) with the same 32 GB of main memory. Their workstation runs Windows 8.1, their code is also written in C++ (and OpenMP) and was compiled using Visual Studio 2012. Since their processor is clocked 0.1Ghz higher (is 2.9% faster than hours), we multiply their timing results with 1.029, similar to [12]. Still, in most cases this tweaking has minimum impact on the observed results.

Table 1. Comparison of GRASP and PHAST for both travel times and travel distances

	Preprocessing Time (s)				**Query Time (ms)**			
	Travel times		**Travel distances**		**Travel times**		**Travel distances**	
	Europe	**USA**	**Europe**	**USA**	**Europe**	**USA**	**Europe**	**USA**
PHAST	99	160	824	475	**39 (103)**	60 (**169**)	50 (**139**)	**64 (179)**
GRASP	**8**	**12**	**10**	**13**	43 (150)	**58** (207)	**46** (156)	66 (218)

GRASP vs. PHAST. We compare GRASP and PHAST's preprocessing and query times for calculating SP distances of all graph vertices from a single source. Preprocessing times refer to parallel execution and for query times we report both sequential and parallel times for a single-tree computation. Query times are averaged over 10,000 randomly selected sources. Results are presented in Table 1. The best results in each case are highlighted in bold and the numbers in parentheses refer to sequential times.

Regarding preprocessing, GRASP is notably more efficient than PHAST. GRASP's preprocessing is *13 times* faster than PHAST for travel times and *37-82* times faster for travel distances. *GRASP's preprocessing takes less than 15s for both metrics* and shows little change when using travel distances, which is in stark contrast to PHAST. In terms of memory consumption, PHAST is slightly better, but GRASP has also very modest requirements, since it requires no more than $1Gb$ for storing the H and $G_{GS}\downarrow$ graphs.

In terms of *query performance*, results are evenly mixed. Although sequential PHAST is slightly faster than GRASP, the parallel implementation of GRASP scales better. As a result, parallel GRASP is faster for the USA network for travel times and Europe network for travel distances.

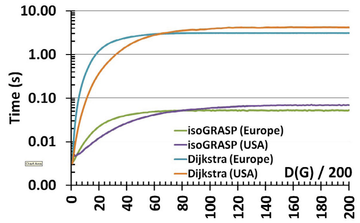
As a result of this first experiment, we observe that GRASP is a more complete solution for solving the one-to-all variant of the SSSP problem. It requires a fraction of PHAST's preprocessing time, is more robust to the metric used and scales better for multicore processors. Also, GRASP's robustness to the metric used, indicates that it will probably outperform PHAST when we switch to other graphs, where CH based solutions typically perform poorly, e.g. social networks.

isoGRASP. The following experimentation evaluates isoGRASP's performance for the *range variant of the SSSP problem*, i.e., assign correct SP distances to all nodes reachable from a single source within a range limit e. To this end, we chose 1000 random vertices as sources s and performed range queries for multiple values of e. Since our benchmark Europe and USA networks have different units for travel times, we use range limits that are fractions of the road network diameters (denoted hereafter as $D(G)$). Since no other pure algorithmic methods exist for this particular problem, we compare our parallel isoGRASP implementation to a sequential Dijkstra implementation. Results are presented in Fig. 2(a) and 2(b), respectively.

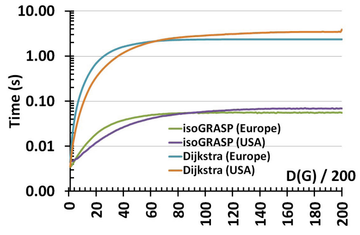
Results are similar for both networks and metrics. IsoGRASP is orders of magnitude faster than Dijkstra and exhibits stable performance for both metrics. In addition for $e < 0.2 \times D(G)$, parallel isoGRASP is at least twice as fast as parallel GRASP / PHAST. Since in range queries we are usually interested in small ranges close to the source vertex, isoGRASP should always be the algorithm of choice, instead of using PHAST / GRASP for calculating all graph vertices distances and then sweep the distances vector to determine which of those are reachable within limit.

reGRASP vs RPHAST. The final set of experiments compares reGRASP and RPHAST for the *one-to-many* variant of finding SP distances from a source s to a non-empty set of targets $T \subseteq V$. We adopt the methodology of [5], which picks a random vertex and performs a Dijkstra search until reaching a fixed number of vertices. If B is the set of vertices settled during this search, our targets T are chosen as a random subset of B.

Hence, our input parameters are the number of targets $|T|$ and the size of $|B|$. This simulates different scenarios, with either targets close together or spread throughout the graph. For each set of parameters, we test 100 different sets of targets, each with 100 different sources. We keep the number of targets $|T|$ fixed at 16,384 (2^{14}) and experimented with different values of $|B|$ ranging from $2^{14} \dots 2^{24}$. We report total times (target selection + query phase) for sequential RPHAST, sequential reGRASP and parallel reGRASP. Results for the Europe and USA road networks are shown in Fig. 3.



(a) Travel times

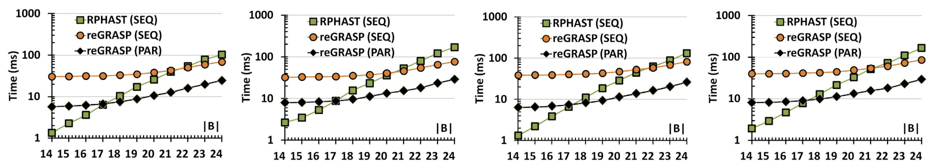


(b) Travel distances

Fig. 2. Parallel isoGRASP performance compared to a sequential Dijkstra for varying values of e compared to total road-network diameter $D(G)$. Y-axis is on logarithmic scale

In terms of sequential performance, RPHAST is faster for $|B| \leq 2^{20}$ and reGRASP in all other cases. However, RPHAST performance degrades quickly and for random objects (i.e., $|B| = 2^{24}$) it takes more than $100ms$, i.e., it is even slower than parallel PHAST / GRASP. In contrast, parallel reGRASP scales well using all available cores and offers stable performance. Even for $|B| = 2^{24}$ a search takes less than $30ms$ for both networks and metrics, i.e., parallel reGRASP is always faster than parallel PHAST or GRASP. Thus, parallel reGRASP is the method of choice, even for large $|B|$ values, or with changing targets $|T|$. RPHAST on the other hand should only be used for the case of a static set of points of interest and small $|B|$ values. When compared to RPHAST, parallel reGRASP does not require any additional graph structures and is 2.6 – 7 times faster, while requiring only a fraction of RPHAST's preprocessing time.

Although at first, it seems unfair to compare sequential RPHAST with parallel reGRASP, RPHAST stands to benefit little from parallelization due to its time-consuming target selection, which is hard to parallelize. Also, RPHAST's query phase uses the $G_T^{\downarrow}$ graph and therefore it is already fast. As a result, very minimal improvements could be achieved through parallelization, due to the small number of edges present in $G_T^{\downarrow}$.



(a) Europe (TT) (b) Europe (TD) (c) USA (TT) (d) USA (TD)

Fig. 3. Parallel and sequential reGRASP performance compared to RPHAST for travel times (TT) and travel distances (TD). X and Y-axes are on logarithmic scale

Table 2. Summary of results for all variants of the SSSP problem

Type of queries	Parameter	Best solution
One-to-all	**Preproc. time / Dynamic networks**	GRASP
	Query time (SEQ)	PHAST
	Query time (PAR)	GRASP / PHAST
	Scales better on multicores	GRASP
Range / Isochrone	**Preproc. time / Dynamic networks**	isoGRASP / GRASP
	Query time (PAR)	isoGRASP
One-to-many	**Preproc. time / Dynamic networks**	GRASP / reGRASP
	No extra data structures	GRASP / reGRASP / PHAST
	Query time (static T, $\|B\| \leq 20$)	RPHAST
	Query time (rest)	reGRASP (PAR)

Summary. Table 2 summarizes our results. It is evident that all GRASP variants are overall the most complete algorithmic solutions for solving all variations of the SSSP problem. They have very short preprocessing times, are therefore suitable for dynamic road networks, provide excellent parallel performance, scale better on multicore processors and offer better or comparable performance to state-of-the-art approaches. Moreover, all GRASP variants utilize the same graph structures and therefore can be used within the same server infrastructure to concurrently answer all related queries. Thus, it is the only approach to offer this kind of simplicity and efficiency for all SSSP cases.

5 Conclusion and Future Work

This work introduced novel graph separator methods to efficiently handle all variations of the single-source shortest-path (SSSP) problem. The three proposed algorithms, GRASP, isoGRASP and reGRASP are each tailored to a specific SSSP problem (one-to-all, range, and one-to-many). All GRASP algorithms rely on the same preprocessing and graph structures and hence, they may be used as a unified framework for answering SSSP queries of any kind. They also require minimal preprocessing time and, thus, are the only viable solution for handling dynamic road networks, i.e., road networks with changing edge weights due to traffic updates. Moreover, they provide excellent parallel performance for both travel times and travel distances metrics. As a result, the GRASP family of algorithms is the best overall solution for processing most SSSP problems.

In terms of future work, we will focus on expanding our results to other types of graphs, for which existing approaches typically do not perform well. Moreover, we will aim at creating a server application that showcases the efficiency of our approach. This might result in the first public server application especially focused on SSSP queries.

Acknowledgments. The research leading to these results has received funding from the EU FP7 Projects "TALOS Mobile Guides" (`http://www.tmguide.eu`, GA: FP7-SME-2012-315333) and "GEOSTREAM" (`http://geocontentstream.eu`, GA: FP7-SME-2012-315631). The authors would also like to thank Daniel Delling for providing PHAST / RPHAST results and Christian Schulz for providing access to the Buffoon partitioning tool.

References

1. Bast, H., Delling, D., Goldberg, A., Müller-Hannemann, M., Pajor, T., Sanders, P., Wagner, D., Werneck, R.: Route planning in transportation networks. Technical Report MSR-TR-2014-4 (January 2014)
2. Baum, M., Dibbelt, J., Pajor, T., Wagner, D.: Energy-optimal routes for electric vehicles. In: SIGSPATIAL/GIS, pp. 54–63 (2013)
3. Delling, D., Goldberg, A.V., Nowatzyk, A., Werneck, R.F.F.: Phast: Hardware-accelerated shortest path trees. In: IPDPS, pp. 921–931 (2011)
4. Delling, D., Goldberg, A.V., Pajor, T., Werneck, R.F.: Customizable route planning. In: Pardalos, P.M., Rebennack, S. (eds.) SEA 2011. LNCS, vol. 6630, pp. 376–387. Springer, Heidelberg (2011)
5. Delling, D., Goldberg, A.V., Werneck, R.F.F.: Faster batched shortest paths in road networks. In: ATMOS, pp. 52–63 (2011)
6. Delling, D., Sanders, P., Schultes, D., Wagner, D.: Engineering route planning algorithms. In: Lerner, J., Wagner, D., Zweig, K.A. (eds.) Algorithmics of Large and Complex Networks. LNCS, vol. 5515, pp. 117–139. Springer, Heidelberg (2009)
7. Delling, D., Werneck, R.F.: Customizable point-of-interest queries in road networks. In: SIGSPATIAL/GIS, pp. 490–493 (2013)
8. Delling, D., Werneck, R.F.: Faster customization of road networks. In: Bonifaci, V., Demetrescu, C., Marchetti-Spaccamela, A. (eds.) SEA 2013. LNCS, vol. 7933, pp. 30–42. Springer, Heidelberg (2013)
9. Demetrescu, C., Goldberg, A.V., Johnson, D.: The shortest path problem. Ninth DIMACS implementation challenge. DIMACS Book 74. AMS (2009)
10. Efentakis, A., Brakatsoulas, S., Grivas, N., Lamprianidis, G., Patroumpas, K., Pfoser, D.: Towards a flexible and scalable fleet management service. In: CTS@SIGSPATIAL (2013)
11. Efentakis, A., Grivas, N., Lamprianidis, G., Magenschab, G., Pfoser, D.: Isochrones, traffic and demographics. In: SIGSPATIAL/GIS, pp. 538–541 (2013)
12. Efentakis, A., Pfoser, D.: Optimizing landmark-based routing and preprocessing. In: CTS@SIGSPATIAL (2013)
13. Efentakis, A., Theodorakis, D., Pfoser, D.: Crowdsourcing computing resources for shortest-path computation. In: SIGSPATIAL/GIS, pp. 434–437 (2012)
14. Geisberger, R., Sanders, P., Schultes, D., Delling, D.: Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In: McGeoch, C.C. (ed.) WEA 2008. LNCS, vol. 5038, pp. 319–333. Springer, Heidelberg (2008)
15. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. 20, 359–392 (1998)
16. Mehlhorn, K., Sanders, P.: Algorithms and Data Structures: The Basic Toolbox. Springer, Berlin (2008)
17. Sanders, P., Schulz, C.: Distributed evolutionary graph partitioning. In: ALENEX (2012)

Switching Colouring of $G(n, d/n)$ for Sampling up to Gibbs Uniqueness Threshold

Charilaos Efthymiou

Goethe University of Frankfurt, 60325, Germany
efthymiou@gmail.com

Abstract. Approximate random k-colouring of a graph $G = (V, E)$, efficiently, is a very well studied problem in computer science and statistical physics. It amounts to constructing, in polynomial time, a k-colouring of G which is distributed close to *Gibbs distribution*. Here, we deal with the problem when the underlying graph is an instance of Erdős-Rényi random graph $G(n, d/n)$, where d is fixed.

This paper improves on the approximate sampling colouring algorithm proposed in SODA 2012. We provide improved performance guarantees for this efficient algorithm, as we reduce the lower bound of the number of colours required by a factor of 1/2. In particular, we show the following statement for the accuracy of algorithm: For typical instances of $G(n, d/n)$ the algorithm outputs a k-colouring of $G(n, d/n)$ which is asymptotically uniform as long $k \geq (1 + \epsilon)d$. For the improvement we make an extensive use of the spatial correlation decay properties of the Gibbs distribution and the local treelike structure of the underlying graph.

1 Introduction

Approximate random k-colouring of a graph $G = (V, E)$, efficiently, is a very well studied problem in computer science and statistical physics. It amounts to constructing, in polynomial time, a k-colouring of G which is distributed close to *Gibbs distribution*, i.e. the uniform distribution over all the k-colourings of G. Here, we deal with the problem when the underlying graph is an instance of Erdős-Rényi random graph $G(n, p)$, where $p = d/n$ and d is fixed. We say that $G(n, p)$ has a property with high probability (*w.h.p.*) if the probability that the property holds tends to 1 as $n \to \infty$.

The problem of sampling colourings when the underlying graph is $G(n, d/n)$ is rather interesting due to *high degree effect*. That is, there is a relative large fluctuation on the degrees of the vertices in the random graph. E.g. it is elementary to show that typical instances of $G(n, d/n)$ have maximum degree $\Theta\left(\frac{\log n}{\log \log n}\right)$, while more than $1 - e^{-O(d)}$ fraction of the vertices have degree in the interval $(1 \pm \epsilon)d$. Usually the bounds for sampling k-colourings w.r.t. k are expressed it terms of the maximum degree e.g. [14,5]. However, for $G(n, d/n)$ the natural bounds for k should be in terms of the expected degree, rather than the maximum.

The most powerful and most popular algorithms for this kind of problems are based on the Markov Chain Monte Carlo (MCMC) method. There the main technical challenge is to establish that the underlying Markov chain mixes in polynomial time (see [11]). The MCMC version of sampling colourings of $G(n, d/n)$ is a well studied problem [7,13,4]. The work in [7] shows that the well known Markov chain *Glauber block dynamics* for k-colourings has polynomial time mixing for typical instances of $G(n, d/n)$ as long as the number of colours $k \geq \frac{11}{2}d$. This is the lowest bound for k as far as MCMC sampling is concerned.

Recently, in [8], the author of this paper suggested a novel non MCMC approach for the approximate sampling colouring problem on $G(n, d/n)$. Roughly, the idea is as follows: Given the input graph, first we remove sufficiently many vertices such that the resulting graph has a "very simple" structure and it can be randomly coloured *efficiently*. Once we have a random colouring of this, simple, graph we start adding one by one all the edges we have removed in the first place. Each time we add a new edge we *update* the colouring so as the graph with the new edge remains (asymptotically) randomly coloured. Once the algorithm has rebuilt the initial graph it returns its colouring.

Let us be more specific on how we *update* the colouring once we add an extra edge. Assume that we are given the fixed graphs $G = (V, E)$ and $G' = (V, E')$ such that $E' = E \cup \{v, u\}$ for some $v, u \in V$. Given X, a random k-colouring of G, we want to create efficiently a random k-colouring of the slightly more complex graph G'. It is easy to show that if the vertices v, u take different colour assignments under X, then the colouring X is a random k-colouring of G'. The interesting case is when $X(v) = X(u)$. Then, the algorithm in [8] uses an operation called "switching" so as to alter the colouring of only one of the two vertices. E.g. using switching, from X which assigns v, u the same colour, we get Y a colouring which assigns v, u different colours while Y is very close to being random. Essentially, switching repermutes the colour classes of an appropriate subgraph of G that contains v[1]. Of course the "switching" can be implemented efficiently.

The approximate sampling algorithm in [8] w.h.p. over the input graph instances returns, in polynomial time, a k-colouring which is asymptotically random as long as $k \geq (2 + \epsilon)d$, for any fixed $\epsilon > 0$. Until this work this was the best bound for sampling colourings of $G(n, d/n)$ in terms of the minimum number of colours required. In this paper we improve on this bound even further by reducing it to $k \geq (1 + \epsilon)d$. For this improvement, we make an extensive use of the the spatial mixing properties of the Gibbs distribution and the local tree structure of the underlying graph.

The technical challenge for the analysis is to show that a random k-colouring does not specify large paths in $G(n, d/n)$ which are coloured with exactly two colours. In particular, we have to argue about the probability of a specific path in $G(n, d/n)$ to be 2-coloured, under the Gibbs distribution. This is a challenging task because of the highly complex structure that $G(n, d/n)$, typically, has. In [8], we deal with this problem by following a very pessimistic scenario about the

[1] In different contexts this component is called Kempe chain.

colouring around the path. I.e. we simplify the problem by making a worst case assumptions about the colouring of the vertices which are incident to the path.

In this work, we follow a more elaborate approach. That is, we consider the colouring of vertices at sufficiently large distance from the path and we show that their colouring does not affect (very much) the colouring of the path. This allows a more accurate estimation of the probability of a path being two coloured (See Section 4).

For presenting our main result we need to use the total variation distance as a measure of distance between distributions.

Definition 1. *For the distributions ν_a, ν_b on $[k]^V$, let $||\nu_a - \nu_b||$ denote their* total variation distance*, i.e.*

$$||\nu_a - \nu_b|| = \max_{\Omega' \subseteq [k]^V} |\nu_a(\Omega') - \nu_b(\Omega')|.$$

For $\Lambda \subseteq V$ let $||\nu_a - \nu_b||_\Lambda$ denote the total variation distance between the projections of ν_a and ν_b on $[k]^\Lambda$.

Theorem 1. *Let $\epsilon > 0$ be fixed and $k = (1+\epsilon)d$. Assume that the input of our algorithm is an instance of $G(n, d/n)$ and we let μ be the uniform distribution over its k-colourings. Also, we let μ' be the distribution of the colouring that is returned by the algorithm. With probability at least $1 - n^{-c}$ over the input instances $G(n, d/n)$ it holds that*

$$||\mu - \mu'|| = O\left(n^{-c}\right),$$

for sufficiently large $c = c(\epsilon) > 0$ and any fixed $d > d_0(\epsilon)$.

The proof of Theorem 1 appears in the full version of this paper in [6].

As far as the time complexity of the algorithm is regarded we provide Theorem 2, its proof appears in the full version of the paper in [6].

Theorem 2. *With probability at least $1 - n^{-2/3}$ over the input instances $G(n, d/n)$, the time complexity of the random colouring algorithm is $O(n^2)$.*

In this extended abstract, we are going to omit most of the technical details which are already known from [8]. In the full version of the paper in [6] we provide the proofs of all the results above. As a matter of fact we provide an improved and to a large extend simplified presentation of the results and their proofs that appeared in [8].

Structure of the paper. In Section 2 we provide a basic description of the algorithm. In Section 3 we give an overview of how the analysis works. In Section 4 we give a sketch of how do we use correlation decay to bound the probability of a path being two coloured.

Notation. We denote with small letters of the greek alphabet the colourings of a graph G, e.g. σ, η, τ, while we use capital letters for the random variables which take values over the colourings e.g. X, Y, Z. We denote with $\sigma_v, X(v)$ the colour assignment of the vertex v under the colouring σ and X, respectively. Finally, for an integer $k > 0$ let $[k] = \{1, \ldots, k\}$.

2 Basic Description

First we need the notion of *switching* colouring. For this we define the *disagreement graph* [2]. Consider a fixed graph G and let v be a distinguished vertex in G. Let σ be a k-colouring of G and let some colour $q \neq \sigma(v)$. Under the colouring σ, we denote by $V_{\sigma(v)}, V_q$ the colour classes of $\sigma(v)$, and q, respectively. We call *disagreement graph* $Q_{\sigma(v),q}$ the maximal, connected, induced subgraph of G which includes v and vertices only from the set $V_{\sigma_v} \cup V_q$. In the colouring of Figure 1, the disagreement graph $Q_{B,G}$ is the one with the fat lines.

Definition 2 (Switching). *Consider G, v, σ and q as specified above. The "q-switching of σ" corresponds to the proper colouring of G which is derived by exchanging the assignments in the two colour classes in $Q_{\sigma_v,q}$.*

We would like to emphasize that the q-switching of any proper colouring of G is always a proper colouring too. Figure 2 illustrates a switching of the colouring that appears in Figure 1. Observe that the colouring in Figure 2 differs from the colouring in Figure 1 to that we have exchanged the two colour classes of the subgraph with the fat lines.

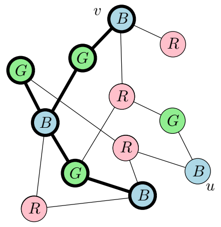


Fig. 1. "Disagreement graph"

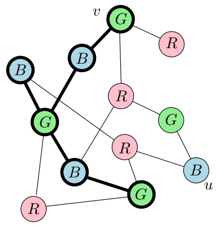


Fig. 2. "g-switching"

We assume that the input of the algorithm is an instance of $G(n, d/n)$ and k, the numbers of colours. The algorithm is as follows:

Set up: We construct a sequence $G_0, \ldots, G_r$ such that every G_i is a subgraph of $G(n, d/n)$. The graph G_r is identical to $G(n, d/n)$. Each G_i is derived by deleting $e_i = \{v_i, u_i\}$ from G_{i+1}, a random edge among those which do not belong to a *small cycle* of G_{i+1}. We call small, any cycle of length less than $(\log n)/(9 \log d)$. G_0 is the graph we get when there are no other edges to delete.

[2] What we call disagreement graph here, is also known as *Kempe chain* e.g. see [12].

With probability $1 - n^{-\Omega(1)}$, over the instances of $G(n, d/n)$, G_0 is simple enough and we can k-colour it randomly in polynomial time[3]. Assuming that we deal with such an instance, the algorithm works as follows:

Updates: Take a random colouring of G_0. Colour the rest of the graph according to the following inductive rule: Given that G_i is coloured Y_i, so as to get Y_{i+1}, the colouring of G_{i+1}, we distinguish two cases

Case a: In the colouring of the graph G_i the vertices v_i and u_i are assigned different colours (i.e. $Y_i(v_i) \neq Y(u_i)$).
Case b: In the colouring of the graph G_i the vertices v_i and u_i are assigned the same colour (i.e. $Y_i(v_i) = Y(u_i)$).

In the first case, we just set $Y_{i+1} = Y_i$, i.e. G_{i+1} gets the same colouring as G_i. In the second case, we choose q uniformly at random among all the colours but $Y_i(v_i)$. Then we set Y_{i+1} equal to the q-switching of Y_i. The q-switching is w.r.t. the vertex v_i.

With the above we conclude the description of the algorithm.

The reader may have observed that the switching does not necessarily provide a k-colouring where the assignments of v_i and u_i are different. That is, it may be that both vertices v_i, u_i belong to the disagreement graph. Then, after the q-switching of Y_i the colour assignments of v_i and u_i remain the same. We will show, that such a situation is rare as long as $k \geq (1 + \epsilon)d$. Typically, after the q-switching v_i, u_i get different colour assignments. The approximate nature of the algorithm amounts exactly to the fact that on some, rare, occasions the switching somehow fails[4].

3 The Setting for the Analysis of the Algorithm

In this section we provide the setting for analyzing the algorithm.

Definition 3 (Good & Bad Colourings). *Consider a graph G and let v, u be two distinguished vertices in this graph. Let σ be a proper k-colouring of G. We call σ a* bad colouring *w.r.t. the vertices v, u if $\sigma_v = \sigma_u$. Otherwise, we call σ* good*.*

The idea that underlies the algorithm, essentially, reduces the problem of sampling to dealing with the following problem.

Problem 1. Consider the graph G and two non adjacent vertices v, u. Given a *bad* random colouring of G w.r.t. v, u, turn it to a *good* random colouring, in polynomial time.

[3] The graph G_0, typically, contains connected components of two kinds. The first one is isolated vertices and the second one is simple cycles. Such graph can be randomly coloured trivially. Whether the algorithm is polynomial time or not depends exactly on whether G_0 can be coloured randomly efficiently. For more details see in the full version of this paper in [6].

[4] For $k \leq d$ our analysis cannot guarantee that these are fails sufficiently rare.

Consider two different colours $c, q \in [k]$ and let $\Omega_{c,c}$ and $\Omega_{q,c}$ be the set of colourings of G which assign the pair of vertices (v, u) colours (c, c) and (q, c), respectively. Essentially, Problem 1 asks to find a mapping $H_{c,q} : \Omega_{c,c} \to \Omega_{q,c}$, for every pair of different colours (c, q), such that the following two hold: (A) If Z is uniformly random in $\Omega_{c,c}$ then $H_{c,q}(Z)$ is uniformly random in $\Omega_{q,c}$. (B) The computation of $H_{c,q}(Z)$ can be accomplished in polynomial time.

For dealing with (A) an ideal, and to a great extent untrue, situation would have been if $\Omega_{c,c}$ and $\Omega_{q,c}$ admitted a bijection and $H_{c,q}$ was a bijection between the two sets. Since, this situation is not expected to hold in general, our approach is based on introducing an *approximate bijection* between $\Omega_{c,c}$ and $\Omega_{q,c}$. That is, we consider a mapping which is a bijection between two sufficiently large subsets of $\Omega_{c,c}$ and $\Omega_{q,c}$, respectively. Each of these two subsets will contain all but a vanishing fraction of the colourings of the original sets.

To be more specific, we assume that $H_{c,q}$ represents the operation of q-switching over the colourings in $\Omega_{c,c}$, as we described in Section 2. Then, there are sufficiently large sets $\Omega'_{c,c} \subseteq \Omega_{c,c}$ and $\Omega'_{q,c} \subseteq \Omega_{q,c}$ such that $H_{c,q}$ is a bijection between $\Omega'_{c,c}$ and $\Omega'_{q,c}$. In particular, we show the following: For Z which is distributed uniformly at random in $\Omega_{c,c}$, $H_{c,q}(Z)$ is distributed within total variation distance $\max\left\{\frac{\Omega_{c,c}\backslash\Omega'_{c,c}}{\Omega_{c,c}}, \frac{\Omega_{q,c}\backslash\Omega'_{q,c}}{\Omega_{q,c}}\right\}$ from the uniform distribution over $\Omega_{q,c}$. That is, the error we introduce depends on the relative size of the subset of colourings in $\Omega_{c,c}$ (resp. $\Omega_{q,c}$) for which $H_{c,q}$ fails to be a bijection. The colourings in $\Omega_{c,c}$ (resp. $\Omega_{q,c}$) that cannot be included in $\Omega'_{c,c}$ (resp. $\Omega'_{q,c}$) are called *pathological*.

We estimate the accuracy of the algorithm by providing upper bounds on the relative number of pathological colourings in $\Omega_{c,c}$ and $\Omega_{q,c}$, respectively. It turns out that the pathological colourings in $\Omega_{c,c}$ (resp. $\Omega_{q,c}$) are exactly these ones for which there is at least one path between v, u which is coloured only with c, q. Applying a q-switching to a pathological colouring in $\Omega_{c,c}$ we will get a new one which assigns both v, u the colour q, i.e. the switching fails. Also, it is direct to show that a pathological colouring in $\Omega_{q,c}$ cannot be generated by q-switching some colouring in $\Omega_{c,c}$.

3.1 Bounding the Error

The ratio $\frac{\Omega_{c,c}\backslash\Omega'_{c,c}}{\Omega_{c,c}}$, $\left(\text{resp. } \frac{\Omega_{q,c}\backslash\Omega'_{q,c}}{\Omega_{q,c}}\right)$ essentially expresses the probability of getting a pathological colouring if we choose uniformly at random from $\Omega_{c,c}$ (resp. $\Omega_{q,c}$).

Assume that we choose u.a.r. from $\Omega_{c,c}$. For every path P that connects v, u in the graph G, we let $I_{\{P\}}$ be the indicator variable which is one if the vertices in the path P are coloured only with colours c, q[5]. It is elementary to verify that

$$\frac{\Omega_{c,c}\backslash\Omega'_{c,c}}{\Omega_{c,c}} \leq \sum_{P} \Pr\left[I_{\{P\}} = 1\right].$$

Of course the same holds for $\Omega_{q,c}$.

[5] Observe that this is equivalent to having P in the disagreement graph $Q_{q,c}$.

In general, computing the probability $\Pr\left[I_{\{P\}} = 1\right]$ exactly is a formidable task. The challenging part in analyzing the performance of the algorithm is to bound this probability as precisely as possible. In [8] we used the idea of the so-called "Disagreement percolation" [3], illustrated in Figure 3. That is, we consider a path $P = (v, a, b, c, d, e, u)$. The vertices with the lines, in the figure, are exactly these vertices which are adjacent to the path. So as to bound the probability that P is coloured with c, q, we assume a worst case boundary colouring for the lined vertices and fixed colouring for v, u[6]. That is, given the fixed colourings we take a random colouring of the uncoloured vertices in the path and estimate the probability that P is coloured using only c, q. Observe that the exact probability is derived by considering an appropriate convex combination of boundary conditions for the lined vertices.

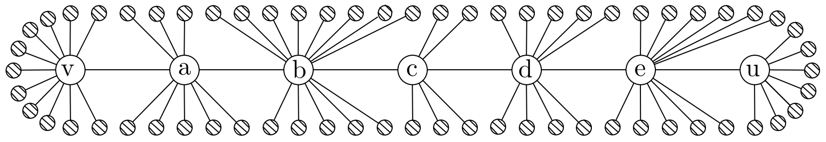


Fig. 3. Boundary at distance 1 from the path

Considering the worst case boundary condition around P is too pessimistic. Our improvement relies on dropping this assumption. The new approach is illustrated in Figure 4. Roughly speaking, we consider boundary conditions at the vertices around P which are at graph distance r, where r is a sufficiently large fixed number. We assume that the colouring of the vertices at the boundary is still a worst case one. However, now we can exploit spatial mixing properties of the Gibbs distributions. That is, the colouring of the distant vertices does not bias the colour of the vertices in P by too much. The weak dependence between the colourings of the vertices in the path and the boundary is achieved by choosing sufficiently large r and k. This allows to estimate more accurately the probability that the path P is 2-coloured.

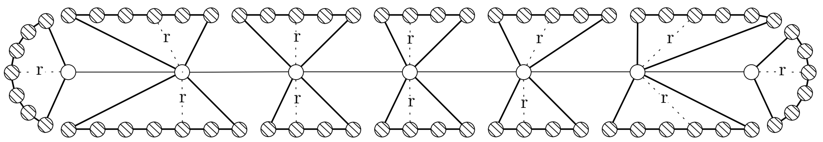


Fig. 4. Boundary at distance r from the path

[6] The vertices v, u are coloured c

4 Correlation Decay to Bound the Number of Bichromatic Paths

In this section we provide a sketch of how do we use correlation decay to show that the bichromatic paths between v_i and u_i are sufficiently rare. In what follows we let k, d, ϵ as in the statement of Theorem 1.

To be more precise, the main task is the following one: Consider an instance of $G(n, d/n)$ conditional that the path $P = (v_0, \dots, v_l)$ appears in the graph, for some integer $l > 0$. Let X be a random k-colouring of the graph, and let $\mathcal{D}_P$ be the event that the path P is coloured only by the colours $c, q \in [k]$. Show that for any positive integer $l \leq \log^2 n$ it holds that

$$\Pr[\mathcal{D}_P] \leq \left(\frac{1}{(1+\epsilon/4)d} \right)^l . \tag{1}$$

For more details of how do we use the above bound in the analysis of the algorithm, see the full version in [6].

Observe that the probability term above is w.r.t. two levels of randomness. The underlying random graph and the random colouring X. To this end, we work as described in the following paragraphs.

Instead of revealing the whole graph, we restrict ourselves to revealing a small area around the path P as well as the edges and vertices between this area and the rest of the graph (outside the area). Let $\mathcal{N}$ denote this area around P[7] Also, let $\partial\mathcal{N}$ be the set of vertices outside $\mathcal{N}$ which are adjacent to some vertex in $\mathcal{N}$.

We consider the subgraph induced by $\mathcal{N}$ and $\partial\mathcal{N}$. We are going to consider a random k-colouring in this graph, conditional some worst case colouring at the vertices in $\partial\mathcal{N}$. That is, the colouring at $\partial\mathcal{N}$ maximizes the probability of the event $\mathcal{D}_P$.

Now, essentially, we have to deal with the randomness of $\mathcal{N}$ and the worst case colourings in $\partial\mathcal{N}$. However, for the vast majority of the vertices in P their neighbourhood (outside P) will be a tree of sufficiently large height and maximum degree at most $(1+\epsilon/3)d+1$. For such a good case of vertex v the boundary colouring will be at a relatively large distance away from v and, essentially, it won't affect its colouring too much. For the exceptional vertices which do not have such well behaved neighbourhood we are very pessimistic, i.e. we give the vertex on the path the appropriate colour for free. The bad cases are expected to be very rare. Somehow this setting reduces the randomness to considering how many good (resp. bad) behaved neighbourhood we have along the path.

In what follows we describe in detail how do we consider the subgraph around the path P. We use some integer parameter $r > 0$. Around each vertex $v_i \in P$ we are going to reveal a subgraph of maximum radius r.

Area Around Path P. For each vertex $v_i \in P$ we need to define the sets of vertices $\mathcal{L}_s(v_i)$, for integer s such that $0 \leq s \leq r$. By definition, we set $\mathcal{L}_0(v_i) = \{v_i\}$. Also, we let $\mathcal{N}_r(v_i)$ be the induced subgraph of $G(n, d/n)$ which contains the vertices $\bigcup_{s=0}^{r} \mathcal{L}_s(v_i)$.

[7] $\mathcal{N}$ is a subgraph which also contains P.

We are going to describe how do we get each $\mathcal{L}_s(v_i)$, inductively. Given $\mathcal{L}_s(v_i)$ we get $\mathcal{L}_{s+1}(v_i)$ by working as follows: Let $\mathcal{R}_{i,s}$ be the set that contains all the vertices of $G(n, d/n)$ but those which belong in the path P, those which belong in $\bigcup_{j\leq s} \mathcal{L}_j(v_i)$ and those which belong in $\mathcal{N}_r(v_j)$, for $j < i$. The set $\mathcal{L}_{s+1}(v_i)$ contains (possibly all) neighbours that the vertices in $\mathcal{L}_s(v_i)$ have in $\mathcal{R}_{i,s}$.

Consider a specific (arbitrary) ordering of the vertices in $\mathcal{R}_{i,s}$. For each vertex $u \in \mathcal{L}_s(v_i)$ we examine adjacency with the vertices in $\mathcal{R}_{i,s}$ in the predefined order. We stop revealing once we either have revealed $(1+\epsilon/3)d+1$ neighbours of u into $\mathcal{R}_{i,s}$ or if we have checked all the possible adjacencies of u with $\mathcal{R}_{i,s}$ (whatever happens first). In both cases, the number of neighbours of u we reveal is at most $(1+\epsilon/3)d+1$.

Once we have the sets $\mathcal{L}_s(v_i)$, for $0 \leq s \leq r$, it is direct to get the subgraph $\mathcal{N}_r(v_i)$. Ideally we would like each of these $\mathcal{N}_r(v_i)$ to be a tree of sufficiently small maximum degree[8]. Also, have would like these $\mathcal{N}_r(v_i)$s to not intersect with each other. If any of these conditions is not true for some $\mathcal{N}_r(v_i)$, then $\mathcal{N}_r(v_i)$ is considered `Fail`. To be more specific, once we the subgraphs $\mathcal{N}_r(v_i)$, for each $0 \leq i \leq l$ we let the following: $\mathcal{N}_r(v_i)$ is `Fail` if at least one of the following happens:

– The maximum degree in $\mathcal{N}_r(v_i)$ is equal to $(1+\epsilon/3)d+2$.
– The graph $\mathcal{N}_r(v_i)$ is not a tree.
– There is at least one integer $j \neq i$ such that some vertex $w_0 \in \mathcal{N}_r(v_j)$ is adjacent to some vertex $w_1 \in \mathcal{N}_r(v_i)$, unless w_0, w_1 are consecutive vertices in the path P.

Having specified $\mathcal{N}_r(v_i)$ for every $v_i \in P$, the sets $\mathcal{N}$ and $\partial\mathcal{N}$ are defined as follows: The set $\partial\mathcal{N}$ contains the vertices $v_i \in P$ for which $\mathcal{N}_r(v_i)$ is `Fail` and the vertices at distance exactly r from v_i for which $\mathcal{N}_r(v_i)$ is not `Fail`. The set $\mathcal{N}$ includes the vertices of the sets $\mathcal{N}_r(v_i)$ which are not `Fail` and do not belong to $\partial\mathcal{N}$.

A vertex v_i in the path is called disagreeing if the following holds: For i even, the color of v_i is c. For i odd, the colour is q . Let the event D_i that "v_i is disagreeing". Clearly it holds that

$$\Pr\left[D_P\right] \leq \Pr\left[\cap_{i=1}^{l} D_i\right]. \tag{2}$$

The proposition will follow by bounding appropriately $\Pr\left[\cap_{i=1}^{l} D_i\right]$.

Let the event A_i, B_i, C_i be defined as follows: $A_i =$ "$\mathcal{N}_r(x_i)$ is `Fail`". $B_i =$ "$\mathcal{N}_r(x_i)$ is not `Fail` and x_i is disagreeing". Finally, $C_i = A_i \cup B_i$. In the full version of this work in [6] we get the following inequality

$$\Pr\left[\cap_{i=1}^{l} D_i\right] \leq \Pr\left[\cap_{i=1}^{l} C_i\right]. \tag{3}$$

The above inequality, which is easy to prove, somehow, follows from the discussion at the beginning of this section. From (2) and (3) we get that

$$\Pr\left[\mathcal{D}_P\right] \leq \prod_{i=1}^{l} \Pr\left[C_i \middle| \cap_{j=1}^{i-1} C_j\right] \leq \prod_{i=1}^{l} \left(\Pr\left[A_i \middle| \cap_{j=1}^{i-1} C_j\right] + \Pr\left[B_i \middle| \cap_{j=1}^{i-1} C_j\right]\right), \tag{4}$$

[8] Maximum degree at most $(1+\epsilon/3)d+1$

where the second inequality follows from a simple the union bound. Then, (1) follows by bounding appropriately the rightmost part in (4).

For bounding the terms $\Pr\left[A_i | \cap_{j=1}^{i-1} C_j\right]$ we use the following lemma.

Lemma 1. *Consider a sufficiently large fixed integer $r > 0$, independent of d. Consider the set $\mathcal{N}_r(v_j)$, for every $v_j \in P$. For sufficiently large d it holds that*

$$\Pr\left[\mathcal{N}_r(v_i) \text{ is } \texttt{Fail} \,|\mathcal{N}_r(v_j) \text{ for } j \neq i\right] \leq \exp\left(-\epsilon^2 d/35\right).$$

That is, it holds that

$$\Pr\left[A_i | \cap_{j=1}^{i-1} C_j\right] \leq \exp\left(-\epsilon^2 d/35\right). \tag{5}$$

For bounding the terms $\Pr\left[B_i | \cap_{j=1}^{i-1} C_j\right]$ we use the following spatial mixing result.

Proposition 1. *Let k, ϵ and d as in Theorem 1. Consider a path $P = (v_0, \ldots, v_l)$ in $G(n, d/n)$ such that $\mathcal{N}_r(v_i)$ is not* `Fail`. *Let X be a random colouring of $G(n, d/n)$. Let M denote the set of vertices outside $\mathcal{N}_r(v_i)$. For any proper colouring σ of $G(n, d/n)$ and any colour $c \in [k] \backslash \{\sigma_{v_{i-1}}, \sigma_{v_{i+1}}\}$ it holds that*

$$\left|\Pr[X(v_i) = c | X(M) = \sigma_M] - \frac{1}{k - t_\sigma}\right| \leq \frac{f_{\epsilon,r}}{k - t_\sigma},$$

where for a fixed $\epsilon > 0$, $f_{\epsilon,r} > 0$ is a decreasing function of r. Also, t_σ is number of different colours that σ_M uses for colouring v_{i-1} and v_{i+1}.

Using the proposition above, we get that

$$\Pr\left[B_i | \cap_{j=1}^{i-1} C_j\right] \leq \frac{1}{k-2} + \frac{f_{\epsilon,r}}{k-2},$$

where $f_{\epsilon,r}$ is defined in the statement of Proposition 1. Taking sufficiently large d and $r = r(\epsilon)$ we bound appropriately the rightmost part of (4).

Acknowledgement. The author would like to thank the anonymous reviewers for their very useful comments and remarks.

References

1. Achlioptas, D., Coja-Oghlan, A.: Algorithmic Barriers from Phase Transitions. In: Proc. of FOCS 2008, pp. 793–802 (2008)
2. Aldous, D.: Random walks of finite groups and rapidly mixing Markov chains. In: Séminaire de Probabilités XVII 1981/82, pp. 243–297. Springer, Heidelberg (1983)
3. van den Berg, J., Maes, C.: Disagreement percolation in the study of Markov fields. Annals of Probability 22, 749–763 (1994)
4. Dyer, M., Flaxman, A., Frieze, A.M., Vigoda, E.: Random colouring sparse random graphs with fewer colours than the maximum degree. Journal R.S.A. 29, 450–465 (2006)

5. Dyer, M., Frieze, A.M., Hayes, A., Vigoda, E.: Randomly colouring constant degree graphs. In: Proc. of 45th FOCS, pp. 582–589 (2004)
6. Efthymiou, C.: Switching colouring of G(n,d/n) for sampling up to Gibbs Uniqueness Threshold. Technical Report on arxiv.org
7. Efthymiou, C.: MCMC sampling colourings and independent sets of G(n, d/n) near uniqueness threshold. In: Proc. of SODA 2014, pp. 305–316 (2014)
8. Efthymiou, C.: A simple algorithm for random colouring $G(n, d/n)$ using $(2 + \epsilon)d$ colours. In: Proc. of SODA 2012, pp. 272–280 (2012)
9. Jonasson, J.: Uniqueness of Uniform Random Colorings of Regular Trees. Statistics & Probability Letters 57, 243–248 (2001)
10. Janson, S., Luczak, T., Ruciński, A.: Random graphs. Wiley and Sons, Inc. (2000)
11. Jerrum, M., Sinclair, A.: The Markov chain Monte Carlo method:an approach to approximate counting and integration. In: Dorit, H. (ed.) Approximation Algorithms for NP-Hard Problems. PWS (1996)
12. Molloy, M.: The freezing threshold for k-colourings of a random graph. In: Proc. of the 44th ACM Symposium on Theory of Computing (STOC 2012), pp. 921–930 (2012)
13. Mossel, E., Sly, A.: Gibbs Rapidly Samples Colorings of $G_{n,d/n}$. Journal Probability Theory and Related Fields 148(1-2) (2010)
14. Vigoda, E.: Improved bounds for sampling colorings. Journal of Mathematical Physics 41(3), 1555–1569 (2000); A preliminary version appears in FOCS 1999

From Graph to Hypergraph Multiway Partition: Is the Single Threshold the Only Route?

Alina Ene[1] and Huy L. Nguyễn[2]

[1] Center for Computational Intractability, Princeton University, Department of Computer Science and DIMAP, University of Warwick
aene@cs.princeton.edu

[2] Department of Computer Science, Princeton University
hlnguyen@princeton.edu

Abstract. We consider the Hypergraph Multiway Partition problem (Hyper-MP). The input consists of an edge-weighted hypergraph $\mathcal{G} = (V, \mathcal{E})$ and k vertices $s_1, \ldots, s_k$ called terminals. A multiway partition of the hypergraph is a partition (or labeling) of the vertices of $\mathcal{G}$ into k sets $A_1, \ldots, A_k$ such that $s_i \in A_i$ for each $i \in [k]$. The cost of a multiway partition $(A_1, \ldots, A_k)$ is $\sum_{i=1}^{k} w(\delta(A_i))$, where $w(\delta(\cdot\,))$ is the hypergraph cut function. The Hyper-MP problem asks for a multiway partition of minimum cost.

Our main result is a 4/3 approximation for the Hyper-MP problem on 3-uniform hypergraphs, which is the first improvement over the $(1.5 - 1/k)$ approximation of [5]. The algorithm combines the single-threshold rounding strategy of Calinescu *et al.* [3] with the rounding strategy of Kleinberg and Tardos [8], and it parallels the recent algorithm of Buchbinder *et al.* [2] for the Graph Multiway Cut problem, which is a special case.

On the negative side, we show that the KT rounding scheme [8] and the exponential clocks rounding scheme [2] cannot break the $(1.5 - 1/k)$ barrier for arbitrary hypergraphs. We give a family of instances for which both rounding schemes have an approximation ratio bounded from below by $\Omega(\sqrt{k})$, and thus the Graph Multiway Cut rounding schemes may not be sufficient for the Hyper-MP problem when the maximum hyperedge size is large. We remark that these instances have $k = \Theta(\log n)$.

1 Introduction

In this paper, we consider the Hypergraph Multiway Partition problem (Hyper-MP). The input consists of an edge-weighted hypergraph $\mathcal{G} = (V, \mathcal{E})$ and k vertices $s_1, \ldots, s_k$ called terminals. A multiway partition of the hypergraph is a partition (or labeling) of the vertices of $\mathcal{G}$ into k sets $A_1, \ldots, A_k$ such that $s_i \in A_i$ for each $i \in [k]$. The cost of a multiway partition $(A_1, \ldots, A_k)$ is $\sum_{i=1}^{k} w(\delta(A_i))$, where $w(\delta(\cdot\,))$ is the hypergraph cut function[1]. The Hyper-MP problem asks for

[1] For each set $A \subseteq V$ of vertices, we use $\delta(A)$ to denote the set of all hyperedges leaving A, i.e., hyperedges $e \in \mathcal{E}$ such that $A \cap e$ and $(V - A) \cap e$ are both non-empty. We use $w(\delta(A))$ to denote the total weight of the edges leaving A, i.e., $w(\delta(A)) = \sum_{e \in \delta(A)} w(e)$.

a multiway partition of minimum cost. We also parameterize the problem by the maximum cardinality of a hyperedge, denoted by c; the well-studied Graph Multiway Cut problem is the special case for which $c = 2$.

The Hyper-MP problem was introduced by Lawler [9] and it has applications in information storage and retrieval, numerical taxonomy, packaging of electric circuits, and VLSI designs [9,1]. Alpert *et al.* [1] emphasize that, in the context of VLSI design, the Hyper-MP objective function better reflects the true cost than simply counting the number of hyperedges being cut because a net (represented by a hyperedge) spanning more clusters consumes more I/O and timing resources.

Multiway cut and partition problems are well-studied from a theoretical point of view as well, with the Graph Multiway Cut problem receiving the most attention. Dahlhaus *et al.* [6] initiated the study of the Graph Multiway Cut problem; they showed that the problem is MAX SNP-hard even for $k = 3$ and they gave a combinatorial algorithm that achieves a $(2 - 2/k)$ approximation. In a breakthrough result, Calinescu, Karloff, and Rabani [3] obtained an $(1.5 - 1/k)$ approximation via a novel geometric relaxation. Since the work of [3], the approximation factor has been improved in a series of papers [7,2,10], culminating with the 1.30217 approximation of [10]. All of these improvements use the CKR relaxation as a starting point and they combine the single-threshold rounding strategy of [3] with other rounding schemes.

The progress on the Hyper-MP problem has been much slower, however. It was only recently that Chekuri and Ene [5] gave a $(1.5 - 1/k)$ approximation for the Hyper-MP problem, improving on a previous $(2 - 2/k)$ approximation [11]. The algorithm of [5] uses a single-threshold scheme to round a fractional solution to the CKR relaxation and it achieves the best approximation known for the problem. An interesting open question, which is the main motivation behind this work, is whether one can improve the $(1.5 - 1/k)$ factor by using other rounding schemes, and in particular the strategies underpinning the Graph Multiway Cut algorithms. Calinescu *et al.* [3] observed that, in the graph setting, one can assume without loss of generality that the fractional solution has certain properties, namely that each edge is mapped to a segment on the simplex that is arbitrarily small and it is aligned with the simplex. These properties are crucially exploited by the analyses of all of the rounding schemes for the graph problem. Unfortunately, it is unclear how to extend these simplifying assumptions to the hypergraph setting (in the graph setting, they can be easily achieved by subdividing the edges). The work of [5] provides an analysis of the single-threshold rounding scheme without any assumptions on the fractional solution, but this seems challenging for the other rounding schemes even for 3-uniform[2] hypergraphs.

Our Contributions. Given the obstacles mentioned above, in this paper we focus on bridging the gap between the graph setting ($c = 2$) and the 3-uniform hypergraph setting ($c = 3$). Our main result is a 4/3 approximation for the Hyper-MP problem on 3-uniform hypergraphs, which is the first improvement over the

[2] A hypergraph is ℓ-uniform if every hyperedge has size ℓ.

$(1.5 - 1/k)$ approximation of [5]. We remark that the result immediately extends to the setting in which each hyperedge has at most 3 vertices (instead of exactly 3 vertices).

Theorem 1. *There is a* $4/3$ *approximation algorithm for the* Hyper-MP *problem on* 3*-uniform hypergraphs.*

The algorithm of Theorem 1 combines the single-threshold rounding strategy of Calinescu *et al.* [3] with the rounding strategy of Kleinberg and Tardos [8], and it parallels the recent algorithm of Buchbinder *et al.* [2] for the Graph Multiway Cut problem. As we mentioned above, it seems to be very challenging to analyze these rounding strategies in the absence of simplifying assumptions (such as edge alignment in the graph case). A key ingredient in our approach is a replacement for the alignment property that allows us to simplify the instance and the fractional solution when the hypergraph is 3-uniform. This ingredient together with some additional insights made the analysis tractable, although it remains quite technical and it is more involved than the analysis for graphs.

On the negative side, we show that the KT rounding scheme [8] and the exponential clocks rounding scheme [2] cannot break the $(1.5 - 1/k)$ barrier for arbitrary hypergraphs. More precisely, we give a family of instances with $c \gg k$ for which both rounding schemes have an approximation ratio bounded from below by $\Omega(\sqrt{k})$, and thus the Graph Multiway Cut rounding schemes may not be sufficient for the Hyper-MP problem when c is large. We remark that these instances have $k = \Theta(\log n)$. Due to space constraints, we defer these results to a longer version of this paper.

Other Related Work. As we have already mentioned, the Graph Multiway Cut problem and its generalizations to hypergraphs and submodular functions have been studied extensively over the past two decades. Due to space constraints, we omit a detailed discussion of these results, and we refer the reader to [10,5,4] for additional pointers and references.

2 LP Relaxation

We use a standard LP relaxation for the problem (see Figure 1). For each vertex $v \in V$ and each label $i \in [k]$, we have a variable $x(v,i)$ with the interpretation that $x(v,i) = 1$ if vertex v receives label i. It is convenient to write the LP in the form described in Fig. 1; although the objective function is not linear, we can easily rewrite it so that it becomes linear. We remark that the LP relaxation is equivalent to the relaxation of [5].

In the remainder of this section, we show that it suffices to round fractional solutions to the above LP that have some additional properties. We start by introducing some notation and a definition. For a vector $v \in \mathbb{R}^k$ and a set $S \subset [k]$, we denote by $v|_S$ the $|S|$-dimensional vector equal to the restriction of v to the coordinates in S.

(Hyper-MP LP)

$$\begin{array}{lll} \min & \displaystyle\sum_{e \in \mathcal{E}} \sum_{i=1}^{k} \left(\max_{u \in e} x(u,i) - \min_{v \in e} x(v,i) \right) & \\ & \displaystyle\sum_{i=1}^{k} x(v,i) = 1 & \forall v \in V \\ & x(s_i, i) = 1 & \forall i \in [k] \\ & x(v,i) \geq 0 & \forall v \in V, i \in [k] \end{array}$$

Fig. 1. LP relaxation for Hyper-MP

Definition 1. *Consider an instance of the* Hyper-MP *problem on a* 3*-uniform hypergraph* $\mathcal{G} = (V, \mathcal{E})$*. Let* $\boldsymbol{x}$ *be a feasible LP solution for the instance. We classify the hyperedges of* $\mathcal{E}$ *as follows:*

(A) A hyperedge e *is of type (A) if there is a permutation* a, b, c *of the vertices of* e *such that:*
  - $\boldsymbol{x}^b = \boldsymbol{x}^c$
  - $\boldsymbol{x}^a$ *and* $\boldsymbol{x}^b$ *differ in only* 2 *coordinates*

(B) A hyperedge e *is of type (B) if there is a permutation* a, b, c *of the vertices of* e *and a partition* (L_1, L_2, L_3, L_4) *of* $[k]$ *such that:*
  - $\boldsymbol{x}^a|_{L_1} > \boldsymbol{x}^b|_{L_1} = \boldsymbol{x}^c|_{L_1}$
  - $\boldsymbol{x}^b|_{L_2} > \boldsymbol{x}^c|_{L_2} = \boldsymbol{x}^a|_{L_2}$
  - $\boldsymbol{x}^c|_{L_3} > \boldsymbol{x}^a|_{L_3} = \boldsymbol{x}^b|_{L_3}$
  - $\boldsymbol{x}^a|_{L_4} = \boldsymbol{x}^b|_{L_4} = \boldsymbol{x}^c|_{L_4}$

(C) A hyperedge e *is of type (C) if there is a permutation* a, b, c *of the vertices of* e *and a partition* (L_1, L_2, L_3, L_4) *of* $[k]$ *such that:*
  - $\boldsymbol{x}^a|_{L_1} < \boldsymbol{x}^b|_{L_1} = \boldsymbol{x}^c|_{L_1}$
  - $\boldsymbol{x}^b|_{L_2} < \boldsymbol{x}^c|_{L_2} = \boldsymbol{x}^a|_{L_2}$
  - $\boldsymbol{x}^c|_{L_3} < \boldsymbol{x}^a|_{L_3} = \boldsymbol{x}^b|_{L_3}$
  - $\boldsymbol{x}^a|_{L_4} = \boldsymbol{x}^b|_{L_4} = \boldsymbol{x}^c|_{L_4}$

(D) A hyperedge e *is of type (D) if it does not fall into any of the types above.*

As shown in the following lemma, it suffices to consider instances of the problem where all the hyperedges fall into one of the first three types (that is, there is no hyperedge of type (D)). It is convenient to have the following definition.

Definition 2. *A randomized rounding scheme for the* Hyper-MP LP *relaxation is* c-preserving *if it constructs an integral solution such that, for each hyperedge* e*, the expected number of parts in which* e *is split is at most* c *times the fractional cost for* e*.*

Lemma 1. *Suppose that there is a rounding scheme for the* Hyper-MP LP *relaxation that is c-preserving for instances of the problem in which all of the hyperedges are of type (A), (B), or (C). Then there is an c-preserving rounding scheme for arbitrary instances of the problem on* 3*-uniform hypergraphs. Moreover, if the former rounding scheme runs in polynomial time then the latter rounding scheme also runs in polynomial time.*

Proof: Consider an instance of Hyper-MP on a 3-uniform hypergraph $\mathcal{G} = (V, \mathcal{E})$, and let $\mathbf{x}$ be a fractional solution for the instance. In the following, we modify the instance and the fractional solution in order to ensure that there are no hyperedges of type (D).

Let e be a hyperedge of type (D). Suppose that there exists a permutation a, b, c of the vertices of e and two labels $i, j \in [k]$ such that $x(a,i) > \max\{x(b,i), x(c,i)\}$ and $x(a,j) < \min\{x(b,j), x(c,j)\}$. We modify the instance and the fractional solution as follows. We add a new vertex a' and we replace the hyperedge $\{a,b,c\}$ by two hyperedges, $\{a,a'\}$ and $\{a',b,c\}$. We define a fractional assignment for a' as follows. Let

$$\epsilon = \min\{x(a,i) - \max\{x(b,i), x(c,i)\}, \min\{x(b,j), x(c,j)\} - x(a,j)\} > 0$$

We set $x(a',i) = x(a,i) - \epsilon$, $x(a',j) = x(a,j) + \epsilon$, and $x(a',\ell) = x(a,\ell)$ for all labels $\ell \neq i, j$. Note that the fractional cost of the hyperedges $\{a,a'\}$ and $\{a',b,c\}$ is equal to the fractional cost of the hyperedge $\{a,b,c\}$. Additionally, we can map a multiway partition of $V \cup \{a'\}$ to a multiway partition of V by simply removing a' from the part that contains it; a straightforward case analysis shows that this mapping does not increase the integral cost, since the total contribution of $\{a',b,c\}$ and $\{a,a'\}$ to the integral cost of the former partition is at most the contribution of $\{a,b,c\}$ to the integral cost of the latter partition.

By repeatedly applying the transformation above we may assume that, for any permutation a, b, c of the vertices of e, there do not exist two labels $i, j \in [k]$ such that $x(a,i) > \max\{x(b,i), x(c,i)\}$ and $x(a,j) < \min\{x(b,j), x(c,j)\}$. We now show that, for every permutation a, b, c of the vertices of e, there is a partition (L_1, L_2, L_3, L_4) of $[k]$ such that:

- $\mathbf{x}^a|_{L_1} \neq \mathbf{x}^b|_{L_1} = \mathbf{x}^c|_{L_1}$
- $\mathbf{x}^b|_{L_2} \neq \mathbf{x}^c|_{L_2} = \mathbf{x}^a|_{L_2}$
- $\mathbf{x}^c|_{L_3} \neq \mathbf{x}^a|_{L_3} = \mathbf{x}^b|_{L_3}$
- $\mathbf{x}^a|_{L_4} = \mathbf{x}^b|_{L_4} = \mathbf{x}^c|_{L_4}$

Consider a permutation a, b, c of the vertices of e. We define four sets $L_1, \dots, L_4$ as follows:

- $L_1 = \{i \in [k] \colon x(a,i) \neq x(b,i) = x(c,i)\}$
- $L_2 = \{i \in [k] \colon x(b,i) \neq x(c,i) = x(a,i)\}$
- $L_3 = \{i \in [k] \colon x(c,i) \neq x(a,i) = x(b,i)\}$
- $L_4 = \{i \in [k] \colon x(a,i) = x(b,i) = x(c,i)\}$

We can verify that the sets $L_1, \dots, L_4$ above partition the labels as follows. The sets are disjoint and thus it suffices to check that their union is $[k]$. Suppose for contradiction that there is a label i such that $i \notin L_1 \cup L_2 \cup L_3 \cup L_4$; thus $x(a,i) \neq x(b,i) \neq x(c,i)$. Let a', b', c' be the permutation of $\{a,b,c\}$ such that $x(a',i) < x(b',i) < x(c',i)$. For any label $j \neq i$, one of the following must hold:

– $x(a',j) \leq \min\{x(b',j), x(c',j)\}$, or
– $x(a',j) > \min\{x(b',j), x(c',j)\}$

Suppose that $x(a',j) > \min\{x(b',j), x(c',j)\}$. It follows from our assumption that $x(c',j) \geq \min\{x(a',j), x(b',j)\}$ and $x(a',j) = \max\{x(b',j), x(c',j)\}$, and therefore $x(a',j) = x(c',j) \geq x(b',j)$. Thus, for any label $j \neq i$, we have $x(a',j) \leq x(c',j)$. Since $x(a',i) < x(c',i)$, we have $\sum_{\ell=1}^{k} x(a',\ell) < \sum_{\ell=1}^{k} x(c',\ell)$. But this is impossible, since $\sum_{\ell=1}^{k} x(a',\ell) = \sum_{\ell=1}^{k} x(c',\ell) = 1$.

Therefore the sets $L_1, \ldots, L_4$ partition the label set $[k]$, as claimed. Since $\mathbf{x}^a|_{L_1} \neq \mathbf{x}^b|_{L_1} = \mathbf{x}^c|_{L_1}$, we have two cases: $\mathbf{x}^a|_{L_1} > \mathbf{x}^b|_{L_1} = \mathbf{x}^c|_{L_1}$ and $\mathbf{x}^a|_{L_1} < \mathbf{x}^b|_{L_1} = \mathbf{x}^c|_{L_1}$. We consider each of these cases in turn.

Suppose that $\mathbf{x}^a|_{L_1} < \mathbf{x}^b|_{L_1} = \mathbf{x}^c|_{L_1}$. In the following, we show that $\mathbf{x}^b|_{L_2} < \mathbf{x}^c|_{L_2}$ and $\mathbf{x}^c|_{L_3} < \mathbf{x}^a|_{L_3}$. Suppose for contradiction that $\mathbf{x}^b|_{L_2} > \mathbf{x}^c|_{L_2}$. Then $x(a,j) \leq x(b,j)$ for each $j \in [k]$ and $x(a,\ell) < x(b,\ell)$ for at least one label ℓ. Therefore $\sum_{j=1}^{k} x(a,j) < \sum_{j=1}^{k} x(b,j)$, which is a contradiction. A similar argument shows that we have $\mathbf{x}^c|_{L_3} < \mathbf{x}^a|_{L_3}$. Thus the hyperedge is of type (B).

Suppose that $\mathbf{x}^a|_{L_1} > \mathbf{x}^b|_{L_1} = \mathbf{x}^c|_{L_1}$. In the following, we show that $\mathbf{x}^b|_{L_2} > \mathbf{x}^c|_{L_2}$ and $\mathbf{x}^c|_{L_3} > \mathbf{x}^a|_{L_3}$. Suppose for contradiction that $\mathbf{x}^b|_{L_2} < \mathbf{x}^c|_{L_2}$. Then $x(a,j) \geq x(b,j)$ for each $j \in [k]$ and $x(a,\ell) > x(b,\ell)$ for at least one label ℓ. Therefore $\sum_{j=1}^{k} x(a,j) > \sum_{j=1}^{k} x(b,j)$, which is a contradiction. A similar argument shows that $\mathbf{x}^c|_{L_3} > \mathbf{x}^a|_{L_3}$. Thus the hyperedge is of type (C). □

3 Rounding Algorithms

In this section, we give a rounding scheme that achieves a 4/3 approximation for the Hyper-MP problem on 3-uniform hypergraphs. In the following, we let $\mathbf{x}$ be a fractional solution to the LP relaxation given in Section 2. The rounding strategies that we use have been studied in previous work for the Graph Multiway Cut and Uniform Metric Labeling problems [2,10,8], and they are given in Figure 2.

We devote the rest of this section to the analysis of Algorithm 3. As shown in Lemma 1, we may assume that we have a fractional solution $\mathbf{x}$ such that each hyperedge is of type (A), (B), or (C) (see Definition 1).

For any multiway partition $(A_1, \ldots, A_k)$, the contribution to the integral cost of a hyperedge e is the number of parts in which e is split, i.e., the number of labels i such that $e \in \delta(A_i)$. We consider each hyperedge in turn and we upper bound its expected contribution to the integral cost.

Theorem 2. *Let e be a hyperedge and let $p(e)$ be a random variable equal to the number of parts in which e is split by Algorithm 3. We have*

$$\mathbf{E}[p(e)] \leq \frac{4}{3} \sum_{i=1}^{k} \Big(\max_{v \in e} x(v,i) - \min_{v \in e} x(v,i) \Big).$$

```
Algorithm 1: Single threshold rounding
  Pick $\theta \in (0,1]$ with prob. density $\phi(\theta)$
  Let $A_i \leftarrow \emptyset$ for each $i \in [k]$
  Let $U \leftarrow V$    ⟨⟨Unlabeled vertices⟩⟩
  For $i = 1$ to $k-1$
      $A_i \leftarrow U \cap \{v \in V : x(v,i) \geq \theta\}$
      $U \leftarrow U - A_i$
  $A_k \leftarrow U$
  Return $(A_1, \ldots, A_k)$
```

```
Algorithm 2: Kleinberg-Tardos rounding
  Let $A_i \leftarrow \emptyset$ for each $i \in [k]$
  Let $U \leftarrow V$    ⟨⟨Unlabeled vertices⟩⟩
  While $U$ is non-empty
      Pick $\theta \in (0,1]$ uniformly at random
      Pick $i \in [k]$ uniformly at random
      $A_i \leftarrow A_i \cup \Big(U \cap \{v \in V : x(v,i) \geq \theta\}\Big)$
      $U \leftarrow U - A_i$
  Return $(A_1, \ldots, A_k)$
```

```
Algorithm 3: Combined rounding scheme
  With probability 1/3, run Algorithm 1 with $\phi(t) = 2t$ for all $t \in [0,1]$
  With probability 2/3, run Algorithm 2
```

Fig. 2. The rounding algorithms

We consider the hyperedges of each type in turn. It follows from the work of Buchbinder *et al.* [2] that the theorem holds of hyperedges of type (A).

Lemma 2 (Buchbinder *et al.* [2]). *Let e be a hyperedge and let $p(e)$ be a random variable equal to the number of parts in which e is split by Algorithm 3. If e is of type (A), we have*

$$\mathbf{E}[p(e)] \leq \frac{4}{3}\sum_{i=1}^{k}\Big(\max_{v\in e} x(v,i) - \min_{v\in e} x(v,i)\Big).$$

Now we consider hyperedges of type (B). Let e be a hyperedge of type (B). Let a, b, c be a permutation of the vertices of e and let $L_1, \ldots, L_4$ be a partition of the labels that satisfy the conditions stated in Definition 1. Let $\alpha = \|\mathbf{x}^a|_{L_1} - \mathbf{x}^b|_{L_1}\|_1 = \|\mathbf{x}^b|_{L_2} - \mathbf{x}^a|_{L_2}\|_1$ and $\beta = \|\mathbf{x}^a|_{L_4}\|_1$. Note that the fractional cost of e is 3α. In the following, we analyze the expected contribution of e to the integral cost of the solutions constructed by Algorithms 1 and 2. Due to space constraints, we defer the proof of the next lemma to a longer version of this paper.

Lemma 3. *Let $p_1(e)$ be a random variable equal to the number of parts in which e is split in the partition constructed by Algorithm 1. We have*

$$\mathbf{E}[p_1(e)] \leq 2\Bigg(\sum_{i\in L_1}\Big(x(a,i)^2 - x(b,i)^2\Big) + \sum_{i\in L_2}\Big(x(b,i)^2 - x(c,i)^2\Big) + \sum_{i\in L_3}\Big(x(c,i)^2 - x(a,i)^2\Big)\Bigg).$$

Now we analyze the expected number of parts in which e is split by Algorithm 2. Recall that $\alpha = \|\mathbf{x}^a|_{L_1} - \mathbf{x}^b|_{L_1}\|_1 = \|\mathbf{x}^b|_{L_2} - \mathbf{x}^a|_{L_2}\|_1$.

Lemma 4. *Let $p_2(e)$ be a random variable equal to the number of parts in which e is split in the partition constructed by Algorithm 2. We have*

$$\mathbf{E}[p_2(e)] = \frac{2}{1+2\alpha}\left(3\alpha + \frac{3\alpha^3}{1+\alpha} - \sum_{i\in L_1}\frac{x(b,i)(x(a,i)-x(b,i))}{1+\alpha}\right.$$
$$\left. - \sum_{i\in L_2}\frac{x(c,i)(x(b,i)-x(c,i))}{1+\alpha} - \sum_{i\in L_3}\frac{x(a,i)(x(c,i)-x(a,i))}{1+\alpha}\right)$$

Proof: We have

$$\mathbf{E}[p_2(e)] = 2\Pr[p_2(e)=2] + 3\Pr[p_2(e)=3] = 2\Pr[p_2(e)\geq 2] + \Pr[p_2(e)=3].$$

We analyze the two probabilities separately. We start with $\Pr[p_2(e)\geq 2]$.

Let B be the event that none of the vertices a,b,c of e are assigned at the end of an iteration of Algorithm 2, conditioned on the event that none of the vertices of e are assigned at the beginning of the iteration. The probability that the vertices remain unassigned, given that the label selected in the current iteration is i, is equal to $(1-\max\{x(a,i),x(b,i),x(c,i)\})$. Thus we have

$$\begin{aligned}\Pr[B] &= \frac{1}{k}\sum_{i=1}^{k}\Big(1-\max\{x(a,i),x(b,i),x(c,i)\}\Big)\\ &= 1-\frac{1}{k}\sum_{i=1}^{k}\max\{x(a,i),x(b,i),x(c,i)\}\\ &= 1-\frac{1}{k}\left(\sum_{i\in L_1}x(a,i)+\sum_{i\in L_2}x(b,i)+\sum_{i\in L_3}x(c,i)+\sum_{i\in L_4}x(c,i)\right)\\ &= 1-\frac{1}{k}\left(\sum_{i\in L_1}x(a,i)+\sum_{i\in L_2}x(b,i)+1-\sum_{i\in L_1}x(b,i)-\sum_{i\in L_2}x(a,i)\right)\\ &= 1-\frac{1+2\alpha}{k}\end{aligned}$$

Let $(A_1,\ldots,A_k)$ be the multiway partition constructed by Algorithm 2. For each label i, let P_i denote the probability that $e\subseteq A_i$. We have

$$\Pr[p_2(e)\geq 2] = 1-\sum_{i=1}^{k}P_i$$

Thus, it suffices to analyze each probability P_i. If $i\in L_1$, the probability P_i satisfies the following recurrence:

$$P_i = \frac{x(b,i)}{k} + \frac{x(a,i)-x(b,i)}{k}\cdot\frac{x(b,i)}{1+\alpha} + \Pr[B]\cdot P_i$$

Indeed, consider an iteration and suppose that none of the vertices of e are assigned at the beginning of the iteration. Recall that, since $i\in L_1$, we have

$x(b,i) = x(c,i) < x(a,i)$. Thus, in the current iteration, one of the following holds: all vertices get a label; a gets a label and b and c remain unassigned; all vertices remain unassigned. The first term of the recurrence above, $x(b,i)/k$, corresponds to the event that all the vertices of e are assigned label i in the current iteration. The third term, $\Pr[B]\cdot P_i$, corresponds to the event that all the vertices are assigned label i in a future iteration. The second term, $((x(a,i) - x(b,i))/k)\cdot (x(b,i)/(1+\alpha))$, corresponds to the event that a is assigned label i and b and c are assigned label i in future iterations: $(x(a,i) - x(b,i))/k$ is the probability that a is assigned label i in the current iteration and b and c remain unassigned at the end of the iteration; $x(b,i)/(1+\alpha)$ is the probability that b and c are assigned label i in future iterations (see below for a proof).

We can show that the probability that b and c are assigned label i is equal to $x(b,i)/(1+\alpha)$ as follows. Let Q_i denote the probability that b and c are assigned label $i \in L_1$. The probability Q_i satisfies the following recurrence:

$$Q_i = \frac{x(b,i)}{k} + \frac{1}{k}\sum_{j=1}^{k}\Big(1 - \max\{x(b,j), x(c,j)\}\Big)\cdot Q_i$$

By rearranging, we get

$$Q_i = \frac{x(b,i)}{\sum_{j=1}^{k}\max\{x(b,j), x(c,j)\}}$$

Finally, we have

$$\begin{aligned}\sum_{j=1}^{k}\max\{x(b,j), x(c,j)\} &= \sum_{j\in L_2} x(b,j) + \sum_{j\in L_1\cup L_3\cup L_4} x(c,j)\\ &= \sum_{j\in L_2} x(b,j) + 1 - \sum_{j\in L_2} x(a,j) = 1+\alpha\end{aligned}$$

Therefore $Q_i = x(b,i)/(1+\alpha)$, as claimed. By rearranging the recurrence for P_i, we get

$$\text{For all } i\in L_1\text{: } P_i = \frac{x(b,i)}{1+2\alpha}\left(1 + \frac{x(a,i) - x(b,i)}{1+\alpha}\right)$$

A similar argument shows that:

$$\text{For all } i\in L_2\text{: } P_i = \frac{x(c,i)}{1+2\alpha}\left(1 + \frac{x(b,i) - x(c,i)}{1+\alpha}\right)$$

and

$$\text{For all } i\in L_3\text{: } P_i = \frac{x(a,i)}{1+2\alpha}\left(1 + \frac{x(c,i) - x(a,i)}{1+\alpha}\right)$$

Now consider a label $i \in L_4$; recall that $x(a,i) = x(b,i) = x(c,i)$. The probability P_i satisfies the following recurrence:

$$P_i = \frac{x(a,i)}{k} + \Pr[B]\cdot P_i$$

By rearranging, we get

$$\text{For all } i \in L_4\text{: } P_i = \frac{x(a,i)}{1+2\alpha}$$

Therefore

$$\begin{aligned}1 - \Pr[p_2(e) \geq 2] = \sum_{i=1}^{k} P_i = \frac{1-\alpha}{1+2\alpha} + \sum_{i \in L_1} \frac{x(b,i)(x(a,i)-x(b,i))}{(1+2\alpha)(1+\alpha)} \\ + \sum_{i \in L_2} \frac{x(c,i)(x(b,i)-x(c,i))}{(1+2\alpha)(1+\alpha)} + \sum_{i \in L_3} \frac{x(a,i)(x(c,i)-x(a,i))}{(1+2\alpha)(1+\alpha)}\end{aligned}$$

Finally, we analyze $\Pr[p_2(e) = 3]$. The hyperedge e is split into 3 parts iff a receives a label in L_1, b receives a label in L_2, and c receives a label in L_3. For each triple (i, j, ℓ), where $i \in L_1$, $j \in L_2$, and $\ell \in L_3$, the probability that first a receives label i then b receives j and finally c receives label ℓ is equal to

$$\frac{(x(a,i)-x(b,i))\cdot(x(b,j)-x(c,j))\cdot(x(c,\ell)-x(a,\ell))}{(1+2\alpha)(1+\alpha)}$$

We prove the identity above as follows. Let R_c be the probability that c is assigned label ℓ, given that a and b are assigned at the beginning of the current iteration and c is unassigned. The probability R_c satisfies the recurrence

$$R_c = \frac{x(c,\ell)-x(a,\ell)}{k} + \frac{1}{k}\sum_{t=1}^{k}\Big(1 - x(c,t)\Big)\cdot R_c.$$

The first term is the probability that c receives label ℓ in the current iteration and the second term is the probability that c receives label ℓ in a future iteration. By rearranging, we get

$$R_c = \frac{x(c,\ell)-x(a,\ell)}{\sum_{t=1}^{k} x(c,t)} = x(c,\ell) - x(a,\ell)$$

Let $R_{b,c}$ be the probability that b is assigned label j and then c is assigned label ℓ, given that a is assigned at the beginning of the current iteration and b and c are unassigned. The probability $R_{b,c}$ satisfies the following recurrence:

$$R_{b,c} = \frac{x(b,j)-x(c,j)}{k}\cdot R_c + \frac{1}{k}\sum_{t=1}^{k}\Big(1 - \max\{x(b,t), x(c,t)\}\Big)\cdot R_{b,c}$$

The first term is the probability that b receives label j in the current iteration and c receives label ℓ in a future iteration. The second term is the probability that, in future iterations, first b receives label j and then c receives label ℓ. By rearranging, we get

$$R_{b,c} = \frac{x(b,j)-x(c,j)}{\sum_{t=1}^{k}\max\{x(b,t), x(c,t)\}}\cdot R_c = \frac{x(b,j)-x(c,j)}{1+\alpha}\cdot R_c$$

Finally, let $R_{a,b,c}$ be the probability that first a receives label i, then b receives j, and then c receives label ℓ, given that a, b, c are unassigned at the beginning of the current iteration. The probability $R_{a,b,c}$ satisfies the following recurrence:

$$R_{a,b,c} = \frac{x(a,i) - x(b,i)}{k} \cdot R_{b,c} + \frac{1}{k} \sum_{t=1}^{k} \Big(1 - \max\{x(a,t), x(b,t), x(c,t)\}\Big) \cdot R_{a,b,c}$$

The first term is the probability that a receives label i in the current iteration and, in future iterations, first b receives label j and then c receives label ℓ. the second term is the probability that, in future iterations, first a receives label i, then b receives label j, and then c receives label ℓ. By rearranging, we get

$$R_{a,b,c} = \frac{x(a,i) - x(b,i)}{\sum_{t=1}^{k} \max\{x(a,t), x(b,t), x(c,t)\}} \cdot R_{b,c} = \frac{x(a,i) - x(b,i)}{1 + 2\alpha} \cdot R_{b,c}$$

Using a similar argument, we can analyze the probability that a, b, c receive labels i, j, ℓ in a different order. By summing over all possible choices of the labels i, j, ℓ and all possible orders in which a, b, c receive labels i, j, ℓ (respectively), we get

$$\Pr[p_2(e) = 3] = \frac{6\alpha^3}{(1 + 2\alpha)(1 + \alpha)}$$

By putting everything together, we get

$$\begin{aligned}
\mathbf{E}[p_2(e)] &= 2\Pr[p_2(e) \geq 2] + \Pr[p_2(e) = 3] \\
&= \frac{2}{1 + 2\alpha}\left(3\alpha + \frac{3\alpha^3}{1 + \alpha} - \sum_{i \in L_1} \frac{x(b,i)(x(a,i) - x(b,i))}{1 + \alpha}\right. \\
&\quad \left. - \sum_{i \in L_2} \frac{x(c,i)(x(b,i) - x(c,i))}{1 + \alpha} - \sum_{i \in L_3} \frac{x(a,i)(x(c,i) - x(a,i))}{1 + \alpha}\right)
\end{aligned}$$

□

Using Lemma 3 and Lemma 4, we can analyze the expected integral cost of e as follows. The proof of the lemma follows from a somewhat lengthy calculation and we defer it to a longer version of this paper.

Lemma 5. *Let e be a hyperedge of type (B). Let $p(e)$ be a random variable equal to the number of parts in which e is split in the partition constructed by Algorithm 3. We have*

$$\mathbf{E}[p(e)] \leq \frac{4}{3} \sum_{i=1}^{k} \Big(\max_{v \in e} x(v,i) - \min_{v \in e} x(v,i)\Big).$$

Finally, we consider hyperedges of type (C). The analysis is similar as for edges of type (B) and it is even simpler, since a hyperedge of type (C) cannot be split into 3 parts. Due to space constraints, we defer the analysis of this case to a longer version of this paper.

Lemma 6. *Let e be a hyperedge of type (C). Let $p(e)$ be a random variable equal to the number of parts in which e is split in the partition constructed by Algorithm 3. We have*

$$\mathbf{E}[p(e)] \leq \frac{4}{3}\sum_{i=1}^{k}\Big(\max_{v\in e} x(v,i) - \min_{v\in e} x(v,i)\Big).$$

Theorem 2 follows immediately from Lemmas 2, 5, and 6. This completes the analysis of the rounding algorithm.

References

1. Alpert, C.J., Kahng, A.B.: Recent directions in netlist partitioning: a survey. Integration, the VLSI Journal 19(1-2), 1–81 (1995)
2. Buchbinder, N., Naor, J.S., Schwartz, R.: Simplex partitioning via exponential clocks and the multiway cut problem. In: Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing, STOC 2013, pp. 535–544. ACM (2013)
3. Calinescu, G., Karloff, H.J., Rabani, Y.: An improved approximation algorithm for multiway cut. Journal of Computer and System Sciences 60(3), 564–574 (1998); Preliminary version in STOC 1998
4. Chekuri, C., Ene, A.: Approximation algorithms for submodular multiway partition. In: FOCS, pp. 807–816 (2011)
5. Chekuri, C., Ene, A.: Submodular cost allocation problem and applications. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part I. LNCS, vol. 6755, pp. 354–366. Springer, Heidelberg (2011)
6. Dahlhaus, E., Johnson, D.S., Papadimitriou, C.H., Seymour, P.D., Yannakakis, M.: The complexity of multiterminal cuts. SIAM Journal on Computing 23(4), 864–894 (1992); Preliminary version in STOC 1992
7. Karger, D.R., Klein, P.N., Stein, C., Thorup, M., Young, N.E.: Rounding algorithms for a geometric embedding of minimum multiway cut. Mathematics of Operations Research 29(3), 436–461 (2004); Preliminary version in STOC 1999
8. Kleinberg, J.M., Tardos, É.: Approximation algorithms for classification problems with pairwise relationships: Metric labeling and Markov random fields. Journal of the ACM (JACM) 49(5), 616–639 (1999)
9. Lawler, E.L.: Cutsets and partitions of hypergraphs. Networks 3(3), 275–285 (1973)
10. Sharma, A., Vondrák, J.: Multiway cut, pairwise realizable distributions, and descending thresholds. In: STOC (2014)
11. Zhao, L., Nagamochi, H., Ibaraki, T.: Greedy splitting algorithms for approximating multiway partition problems. Mathematical Programming 102(1), 167–183 (2005)

Deterministic Stateless Centralized Local Algorithms for Bounded Degree Graphs

Guy Even, Moti Medina, and Dana Ron

School of Electrical Engineering, Tel-Aviv Univ., Tel-Aviv 69978, Israel
{guy,medinamo,danar}@eng.tau.ac.il

Abstract. We design centralized local algorithms for: maximal independent set, maximal matching, and graph coloring. The improvement is threefold: the algorithms are deterministic, stateless, and the number of probes is $O(\log^* n)$, where n is the number of vertices of the input graph. Our algorithms for maximal independent set and maximal matching improves over previous randomized algorithms by Alon et al. (SODA 2012) and Mansour et al. (ICALP 2012). In these previous algorithms, the number of probes and the space required for storing the state between queries are poly$(\log n)$.

We also design the first centralized local algorithm for graph coloring. Our graph coloring algorithms are deterministic and stateless. Let Δ denote the maximum degree of a graph over n vertices. Our algorithm for coloring the vertices by $\Delta + 1$ colors requires $O(\log^* n)$ probes for constant degree graphs. Surprisingly, for the case where the number of colors is $O(\Delta^2 \log \Delta)$, the number of probes of our algorithm is $O(\Delta \cdot \log^* n + \Delta^2)$, that is, the number of probes is sublinear if $\Delta = o(\sqrt{n})$, i.e., our algorithm applies for graphs with unbounded degrees.

Keywords: Centralized Local Algorithms, Sublinear Approximation Algorithms, Graph Algorithms.

1 Introduction

Local Computation Algorithms, as defined by Rubinfeld et al. [17], are algorithms that answer queries regarding (global) solutions to computational problems by performing local (sublinear time) computations on the input. The answers to all queries must be consistent with a single solution regardless of the number of possible solutions. To make this notion concrete, consider the *Maximal Independent Set* problem, which we denote by MIS. Given a graph $G = (V, E)$ as input, the local algorithm ALG gives the illusion that it "holds" a specific maximal independent set $I \subseteq V$. Namely, given any vertex v as a query, ALG answers whether v belongs to I even though ALG cannot read all of G, cannot store the solution I, and cannot even remember all the answers to previous queries. In order to answer such queries, ALG can probe the graph G by asking about the neighbors of a vertex of its choice.

A local computation algorithm may be randomized, so that the solution according to which it answers queries may depend on its internal coin flips.

However, the solution should not depend on the sequence of the queries (this property is called query order obliviousness [17]). We measure the performance of a local computation algorithm by the following criteria: the maximum number of probes it makes to the input per query, the success probability over any sequence of queries, and the maximum space it uses between queries[1] . It is desired that both the probe complexity and the space complexity of the algorithm be sublinear in the size of the graph (e.g., polylog($|V|$)), and that the success probability be $1 - 1/\text{poly}(|V|)$. It is usually assumed that the maximum degree of the graph is upper-bounded by a constant, but our results are useful also for non-constant upper bounds (see also [16]). For a formal definition of local algorithms in the context of graph problems, which is the focus of this work, see Subsection 2.2.

The motivation for designing local computation algorithms is that local computation algorithms capture difficulties with very large inputs. A few examples include: (1) Reading the whole input is too costly if the input is very long. (2) In certain situations one is interested in a very small part of a complete solution. (3) Consider a setting in which different uncoordinated servers need to answer queries about a very long input stored in the cloud. The servers do not communicate with each other, do not store answers to previous queries, and want to minimize their accesses to the input.

Local computation algorithms have been designed for various graph (and hypergraph) problems, including the abovementioned MIS [17,1], hypergraph coloring [17,1], maximal matching [8] and (approximate) maximum matching [9]. Local computation algorithms also appear implicitly in works on sublinear approximation algorithms for various graph parameters, such as the size of a minimum vertex cover [14,10,19,11]. Some of these implicit results are very efficient in terms of their probe complexity (in particular, it depends on the maximum degree and not on $|V|$) but do not give the desired $1 - 1/\text{poly}(|V|)$ success probability. We compare our results to both the explicit and implicit relevant known results.

As can be gleaned from the definition in [17], local computation algorithms are closely related to *Local Distributed Algorithms* [14]. This connection is discussed in Section 2.3 (see also [4]).

In what follows we denote the aforementioned local computation model by CENTLOCAL (where the "CENT" stands for "centralized") and the distributed (local) model by DISTLOCAL (for a formal definition of the latter, see Subsection 2.3). We denote the number of vertices in the input graph by n and the maximum degree by Δ.

1.1 The Ranking Technique

The starting point for our results in the CENTLOCAL model is the *ranking* technique [10,19,1,8,9]. To exemplify this, consider, once again, the MIS problem.

[1] In our algorithms the running time per query in the RAM model is at most $\text{poly}(q) \cdot \log\log n$, where q is the maximum number of probes per query and $n = |V|$.

A very simple (global "greedy") algorithm for this problem works by selecting an arbitrary ranking of the vertices and initializing I to be empty. The algorithm then considers the vertices one after the other according to their ranks and adds a vertex to I if and only if it does not neighbor any vertex already in I. Such an algorithm can be "localized" as follows. For a fixed ranking of the vertices (say, according to their IDs), given a query on a vertex v, the local algorithm performs a *restricted* DFS starting from v. The restriction is that the search continues only on paths with monotonically decreasing ranks. The local algorithm then simulates the global one on the subgraph induced by this restricted DFS.

The main problem with the above local algorithm is that the number of probes it performs when running the DFS may be very large. Indeed, for some rankings (and queried vertices), the number of probes is linear in n. In order to circumvent this problem, *random* rankings were studied [10]. This brings up two questions, which were studied in previous works, both for the MIS algorithm described above and for other ranking-based algorithms [10,19,1,8,9]. The first is to bound the number of probes needed to answer a query with high probability. The second is how to efficiently store a random ranking between queries.

1.2 Our Contributions

Orientations with bounded reachability. Our first conceptual contribution is a simple but very useful observation. Rather than considering vertex rankings, we suggest to consider *acyclic orientations* of the edges in the graph. Such orientations induce partial orders over the vertices, and partial orders suffice for our purposes. The probe complexity induced by a given orientation translates into a combinatorial measure, which we refer to as the *reachability* of the orientation. Reachability of an acyclic orientation is the maximum number of vertices that can be reached from any start vertex by directed paths (induced by the orientation). This leads us to the quest for a CentLocal algorithm that computes an orientation with bounded reachability.

Orientations and colorings. Our second conceptual contribution is that an orientation algorithm with bounded reachability can be based on a CentLocal *coloring* algorithm. Indeed, every vertex-coloring with k colors induces an orientation with reachability $O(\Delta^k)$. Towards this end, we design a CentLocal coloring algorithm that applies techniques from DistLocal colorings algorithms [3,5,7,13]. Our CentLocal algorithm is deterministic, does not use any space between queries, performs $O(\Delta \cdot \log^* n + \Delta^2)$ probes per query, and computes a coloring with $O(\Delta^2 \log \Delta)$ colors. (We refer to the problem of coloring a graph by poly(Δ) colors as poly(Δ)-Color.) Our coloring algorithm yields an orientation whose reachability is $\Delta^{O(\Delta^2 \log \Delta)}$. For constant degree graphs, this implies $O(\log^* n)$ probes to obtain an orientation with constant reachability. As an application of this orientation algorithm, we also design a CentLocal algorithm for $(\Delta + 1)$-coloring.

Centralized local simulations of sequential algorithms. We apply a general transformation (similarly to what was shown in [1]) from global algorithms with certain properties to local algorithms. The transformation is based on our CentLocal orientation with bounded reachability algorithm. As a result we get deterministic CentLocal algorithms for MIS and maximal matching (MLM), which significantly improve over previous work [17,1,8], and the first CentLocal algorithm for coloring with $(\Delta + 1)$ colors (We refer to the problem of coloring a graph by $\Delta + 1$ colors as $(\Delta + 1)$-Color). Compared to previous work, for MIS and MLM the dependence on n in the probe complexity is reduced from polylog(n) to $\log^*(n)$ and the space needed to store the state between queries is reduced from polylog(n) to zero.

1.3 Comparison to Previous Work

Comparison to previous (explicit) CentLocal *algorithms.* A comparison of our results with previous CentLocal algorithms is summarized in Table 1. The dependence on Δ of previous algorithms is not explicit; the dependency in Table 1 is based on our understanding of these results.

Table 1. A comparison between CentLocal algorithms under the assumption that $\Delta = O(1)$. Our algorithms are deterministic and stateless (i.e., the space needed to store the state between queries is zero). MLM denotes a maximal matching, MM denotes maximum matching.

Problem	Previous work			Here (Deterministic, 0-Space)
	Space	# Probes	success prob.	# Probes
MIS	$\Delta^{O(\Delta \cdot \log \Delta)} \cdot \log^3 n$	$\Delta^{O(\Delta \cdot \log \Delta)} \cdot \log^2 n$	$1 - \frac{1}{\text{poly}(n)}$ [1]	$\Delta^{O(\Delta^2 \log \Delta)} \cdot \log^* n$ [Coro. 6]
MLM	$\Delta^{O(\Delta)} \cdot \log^3 n$	$\Delta^{O(\Delta)} \cdot \log^3 n$	$1 - \frac{1}{\text{poly}(n)}$ [8]	$\Delta^{O(\Delta^2 \log \Delta)} \cdot \log^* n$ [Coro. 6]
poly(Δ)-Color	none	none	none	$O(\Delta \cdot \log^* n + \Delta^2)$ [Thm. 3]
$(\Delta + 1)$-Color	none	none	none	$\Delta^{O(\Delta^2 \log \Delta)} \cdot \log^* n$ [Coro. 6]

Comparison to previous CentLocal *oracles in sublinear approximation algorithms.* A sublinear approximation algorithm for a certain graph parameter (e.g., the size of a minimum vertex cover) is given probe access to the input graph and is required to output an approximation of the graph parameter with high (constant) success probability. Many such algorithms work by designing an *oracle* that answers queries (e.g., a query can ask: does a given vertex belong to a fixed small vertex cover?). The sublinear approximation algorithm estimates the graph parameter by performing (a small number of) queries to the oracle. The oracles are essentially CentLocal algorithms but they tend to have constant error probability, and it is not clear how to reduce this error probability without significantly increasing their probe complexity. Furthermore, the question of bounded space needed to store the state between queries was not an issue in the design of these oracles, since only few queries are performed by the sublinear approximation algorithm. Hence, they are not usually considered to be "bona fide"

Table 2. A comparison between CentLocal oracles in sub-linear approximation algorithms and our CentLocal (deterministic) algorithms. The former algorithms were designed to work with constant success probability and a bound was given on their expected probe complexity. When presenting them as CentLocal algorithms we introduce a failure probability parameter, δ, and bound their probe complexity in terms of δ. Furthermore, the approximation ratios of the sublinear approximation algorithms were stated in additive terms, and we translate the results so as to get a multiplicative approximation.

Problem	Previous work			Here	
	# Probes	success prob.	apx. ratio	# Probes	apx. ratio
MIS	$O(\Delta^4) \cdot \text{poly}(\frac{1}{\delta}, \frac{1}{\varepsilon})$	$1-\delta$	$1-\varepsilon$ [19]	$\Delta^{O(\Delta^2 \log \Delta)} \cdot \log^* n$	1
MLM	$O(\Delta^4) \cdot \text{poly}(\frac{1}{\delta}, \frac{1}{\varepsilon})$	$1-\delta$	$1-\varepsilon$ [19]	$\Delta^{O(\Delta^2 \log \Delta)} \cdot \log^* n$	1
poly(Δ)-Color	none	none	-	$O(\Delta \cdot \log^* n + \Delta^2)$	-
($\Delta+1$)-Color	none	none	-	$\Delta^{O(\Delta^2 \log \Delta)} \cdot \log^* n$	-

CentLocal algorithms. A comparison of our results and these oracles appears in Table 2.

2 Preliminaries

2.1 Notations

Let $G = (V, E)$ denote an undirected graph. Let n denote the number of vertices and m denote the number of edges. We denote the degree of v by $deg(v)$. Let Δ denote the maximum degree, i.e., $\Delta \triangleq \max_{v \in V}\{deg(v)\}$. Let $\Gamma(v)$ denote the set of neighbors of $v \in V$. The length of a path equals the number of edges along the path. We denote the length of a path p by $|p|$. For $u, v \in V$ let $dist(u, v)$ denote the length of the shortest path between u and v. The ball of radius r centered at v is defined by

$$B_r(v) \triangleq \{u \in V \mid dist(v, u) \leq r\} .$$

For $k \in \mathbb{N}^+$ and $n > 0$, let $\log^{(k)} n$ denote the kth iterated logarithm of n. Note that $\log^{(0)} n \triangleq n$ and if $\log^{(i)} n = 0$, we define $\log^{(j)} n = 0$, for every $j > i$. For $n \geq 1$, define $\log^* n \triangleq \min\{i : \log^{(i)} n \leq 1\}$.

2.2 The CentLocal Model

The model of centralized local computations was defined in [17]. In this section we describe this model for problems over labeled graphs.

Labeled graphs. An undirected graph $G = (V, E)$ is labeled if: (1) The vertices have unique names. For simplicity, assume that the vertex names are in $\{1, \ldots, n\}$. We denote the vertex whose name is i by v_i. (2) Each vertex v holds a list of $deg(v)$ pointers, called *ports*, that point to the neighbors of v. The assignment of ports to neighbors is arbitrary and fixed.

Problems over labeled graphs. Let Π denote a computational problem over labeled graphs (e.g., maximum matching, maximal independent set, vertex coloring). A solution for problem Π over a labeled graph G is a function, the domain and range of which depend on Π and G. For example: (1) In the Maximal Matching problem, a solution is an indicator function $M : E \to \{0, 1\}$ of a maximal matching in G. (2) In the problem of coloring the vertices of a graph by $(\Delta+1)$ colors, a solution is a coloring $c : V \to \{1, \ldots, \Delta+1\}$. Let $sol(G, \Pi)$ denote the set of solutions of problem Π over the labeled graph G.

Probes. In the CentLocal model, access to the labeled graph is limited to probes. A *probe* is a pair (v, i) that asks "who is the ith neighbor of v?". The answer to a probe (v, i) is as follows. (1) If $deg(v) < i$, then the answer is "null". (2) If $deg(v) \geq i$, then the answer is the (ID of) vertex u that is pointed to by the ith port of v. For simplicity, we assume that the answer also contains the port number j such that v is the jth neighbor of u. (This assumption reduces the number of probes by at most a factor of Δ.)

Online algorithms in the CentLocal *model.* An online deterministic algorithm ALG for a problem Π over labeled graphs in the CentLocal model is defined as follows. The input for the algorithm consists of three parts: (1) access to a labeled graph G via probes, (2) the number of vertices n and the maximum degree Δ of the graph G, and (3) a sequence $\{q_i\}_{i=1}^N$ of queries. Each query q_i is a request for an evaluation of $f(q_i)$ where $f \in sol(G, \Pi)$. The algorithm is online because it must output an evaluation of $f(q_i)$ without any knowledge of subsequent queries.

We say that ALG is *consistent with* (G, Π) if

$$\exists f \in sol(G, \Pi) \text{ s.t. } \forall N \in \mathbb{N} \ \ \forall \{q_i\}_{i=1}^N \ \ \forall i \ : \ y_i = f(q_i)\,. \tag{1}$$

Consider, for example, the problem of computing a $(\Delta+1)$ vertex coloring. Consistency in this example means the following. The online algorithm is input a sequence of queries, each of which is a vertex. The algorithm must output the color of each queried vertex. If a vertex is queried twice, then the algorithm must return the same color. Moreover, queried vertices that are neighbors must be colored by different colors. Thus, if all vertices are queried, then the answers constitute a legal vertex coloring that uses $(\Delta+1)$ colors. We now describe two measures of performance that are used in the CentLocal model.

Performance measures. In the CentLocal model, two computational resources are considered: state-space and number of probes. The *state* of algorithm ALG is the information that ALG saves between queries. The *state-space* of algorithm ALG is the maximum number of bits required to encode the state of ALG. The state is used to ensure consistency. We note that the running time used to answer a query is not counted.

Definition 1. *An online algorithm is a* CentLocal$[q, s]$ *algorithm for* Π *if (1) it is consistent with* (G, Π)*, (2) it performs at most* q *probes, and (3) the state can be encoded by* s *bits.*

The goal in designing algorithms in the CentLocal model is to minimize the number of probes and the state-space (in particular $q, s = o(n)$). A CentLocal$[q, s]$ algorithm with $s = 0$ is called *stateless* or *zero-state-space.* In this case, we refer to the algorithm as a stateless CentLocal$[q]$-algorithm. Stateless algorithms are useful in the case of uncoordinated distributed servers that answer queries without communicating with each other.

Space vs. state-space. In [17] no distinction was made between the space needed to answer a query and the space needed to store the state between queries. Because the space needed to answer a query is freed after the query is answered, we only count the space needed to store the state between queries.

Randomized local algorithms. If ALG is a randomized algorithm, the consistency requirement is parameterized by the *failure probability* δ. We say that ALG is a CentLocal$[q, s, \delta]$ algorithm for Π with probability at least $1 - \delta$ if it is consistent with (G, Π), performs at most q probes, and has state-space s. The standard requirement is that $\delta = 1/\text{poly}(n)$.

Parallelizability and query order obliviousness. In [1,8,9] two requirements are introduced: *parallelizability* and *query order obliviousness.* These requirements are fully captured by the definition of a consistent, online, deterministic algorithm with zero state-space. That is, every online algorithm that is consistent, zero-state-space, and deterministic is both parallelizable and query order oblivious.

2.3 The DistLocal Model

The model of local distributed computation is a classical model (see [7,15,18]).

The distributed computation takes place in an undirected labeled graph $G = (V, E)$. Each vertex models a processor, and communication is possible between neighboring processors. All processors execute the same algorithm. Initially, every $v \in V$ is input a local input. The computation is done in $r \in \mathbb{N}$ synchronous rounds as follows. In every round: (1) every processor receives a message from each neighbor, (2) every processor performs a computation based on its local input and the messages received from its neighbors, (3) every processor sends a message to each neighbor. We assume that a message sent in the end of round i is received in the beginning of round $i + 1$. After the rth round, every processor computes a local output.

The following assumptions are made in the DistLocal model: (1) The local input to each vertex v includes the ID of v, the degree of the vertex v, the maximum degree Δ, the number of vertices n, and the ports of v to its neighbors. (2) The IDs are distinct. For simplicity, we assume that the IDs are in the set $\{1, \ldots, n\}$. (3) The length of the messages sent in each round is not bounded.

We say that a distributed algorithm is a DistLocal$[r]$-*algorithm* if the number of communication rounds is r. Strictly speaking, a distributed algorithm is considered *local* if r is bounded by a constant. We say that a DistLocal[r]-algorithm is *almost local* if $r = O(\log^* n)$. When it is obvious from the context we refer to an almost DistLocal algorithm simply by a DistLocal algorithm.

We remark that in the DistLocal model, efficiency of the algorithm executed locally by the processors is not important. Namely, one does not bound the running time required to complete each round.

Simulation of DistLocal *by* CentLocal *[14]:* Every deterministic DistLocal[r]-algorithm, can be simulated by a deterministic, stateless CentLocal[$O(\Delta^r)$]-algorithm. The simulation proceeds simply by probing all vertices in the ball of radius r centered at the query.

If $\Delta = 2$, then balls are simple paths (or cycles) and hence simulation of a DistLocal[r]-algorithm is possible by a CentLocal[$2r$]-algorithm.

3 Acyclic Orientation with Bounded Reachability

In this section we introduce the problem of *Acyclic Orientation with Bounded Reachability* (OBR). We then design a CentLocal algorithm for OBR.

Notations. Let $H = (V, A)$ denote a directed graph, where V is the set of vertices and $A \subseteq V \times V$. The *reachability set* of $v \in V$ is the set of vertices R such that there is a path from v to every vertex in R. We denote the reachability set of $v \in V$ in digraph H by $R_H(v)$. Let $r_H(v) \triangleq |R_H(v)|$ and $r_H^{\max} \triangleq \max_{v \in V} r_H(v)$. We simply write $R(v), r(v), r^{\max}$ when the digraph H is obvious from the context. We say that a digraph $H = (V, A)$ is an *orientation* of an undirected graph $G = (V, E)$ if G is an underlying graph of H.

In the problem of acyclic orientation with bounded reachability (OBR), the input is an undirected graph. The output is an orientation H of G that is acyclic. The goal is to minimize $r_H^{\max}$.

Previous works obtain an acyclic orientation by random vertex ranking [10,19,1,8,9]. We propose to obtain an acyclic orientation by vertex coloring.

Proposition 1 (Orientation via Coloring). *Every coloring by c colors induces an acyclic orientation with*

$$r^{\max} \leq \Delta \cdot \sum_{i=0}^{c-2} (\Delta - i)^i \leq \begin{cases} 2\Delta \cdot (\Delta - 1)^{c-2}, & \text{if } \Delta \geq 3, \\ 2c, & \text{if } \Delta = 2\,. \end{cases}$$

Proof. Direct each edge from a high color to a low color. By monotonicity the orientation is acyclic. Every directed path has at most c vertices, and hence the reachability is bounded as required.

3.1 A CentLocal Algorithm for OBR

In Theorem 3, we present a deterministic, stateless CentLocal[$O(\Delta \cdot \log^* n + \Delta^2)$]-algorithm that computes a vertex coloring that uses $c = O(\Delta^2 \log \Delta)$ colors. Orientation by this coloring yields an acyclic orientation with $r^{\max} \leq \Delta^c$.

Acyclic orientation can be also obtained by simulating DistLocal vertex coloring algorithms. Consider, for example, the $(\varDelta + 1)$ coloring using $r_1 = O(\varDelta) + \frac{1}{2} \cdot \log^* n$ rounds of [2] or the $O(\varDelta^2)$ coloring using $r_2 = O(\log^* n)$ rounds of [7]. CentLocal simulations of these algorithms require $O(\varDelta^{r_i})$ probes. Thus, in our algorithm, the number of probes grows (slightly) slower as a function of n and is polynomial in $\varDelta$.

Our algorithm relies on techniques from two previous DistLocal coloring algorithms.

Theorem 1 ([7, Corollary 4.1]). *A* $5\varDelta^2 \log c$ *coloring can be computed from a* c *coloring by a* DistLocal[1]*-algorithm.*

Theorem 2 ([12, Section 4]). *A* $(\varDelta + 1)$ *coloring can be computed by a* DistLocal$[O(\varDelta^2 + \log^* n)]$*-algorithm.*

Theorem 3. *An* $O(\varDelta^2 \log \varDelta)$ *coloring can be computed by a deterministic, stateless* CentLocal$[O(\varDelta \cdot \log^* n + \varDelta^2)]$*-algorithm.*

Proof. We begin by describing a two phased DistLocal$[O(\log^* n)]$-algorithm D that uses $O(\varDelta^2 \cdot \log \varDelta)$ colors. Algorithm D is especially designed so that it admits an "efficient" simulation by a CentLocal-algorithm.

Consider a graph $G = (V, E)$ with a maximum degree $\varDelta$. In the first phase, the edges are partitioned into $\varDelta^2$ parts, so that the maximum degree in each part is at most 2. Let $p_i(u)$ denote the neighbor of vertex u pointed to by the ith port of u. Following Kuhn [6] we partition the edge set E as follows. Let $E_{\{i,j\}} \subseteq E$ be defined by

$$E_{\{i,j\}} \triangleq \{\{u, v\} \mid p_i(u) = v, p_j(v) = u\}.$$

Each edge belongs to exactly one part $E_{\{i,j\}}$. For each part $E_{\{i,j\}}$ and vertex u, at most two edges in $E_{\{i,j\}}$ are incident to u. Hence, the maximum degree in each part is at most 2. Each vertex can determine in a single round how the edges incident to it are partitioned among the parts. Let $G_{\{i,j\}}$ denote the undirected graph over V with edge set $E_{\{i,j\}}$.

By Theorem 2, we 3-color each graph $G_{\{i,j\}}$ in $O(\log^* n)$ rounds. This induces a vector of $\varDelta^2$ colors per vertex, hence a $3^{(\varDelta^2)}$ vertex coloring of G.

In the second phase, Algorithm D applies Theorem 1 twice to reduce the number of colors to $O(\varDelta^2 \log \varDelta)$.

We now present an efficient simulation of algorithm D by a CentLocal-algorithm C. Given a query for the color of vertex v, Algorithm C simulates the first phase of D in which a 3-coloring algorithm is executed in each part $E_{\{i,j\}}$. Since the maximum degree of each $G_{\{i,j\}}$ is two, a ball of radius r in $G_{\{i,j\}}$ contains at most $2r$ edges. In fact, this ball can be recovered by at most $2r$ probes. It follows that a CentLocal simulation of the 3-coloring of $G_{\{i,j\}}$ requires only $O(\log^* n)$ probes. Observe that if vertex v is isolated in $G_{\{i,j\}}$, then it may be colored arbitrarily (say, by the first color). A vertex v is not isolated in at most $\varDelta$ parts. It follows that the simulation of the first phase requires $O(\varDelta \cdot \log^* n)$ probes.

The second phase of algorithm D requires an additional Δ^2 probes, and the theorem follows.

Corollary 4. *There is a deterministic, stateless* CENTLOCAL$[O(\Delta \cdot \log^* n + \Delta^2)]$*-algorithm for* OBR *that achieves* $r^{\max} \leq \Delta^{O(\Delta^2 \log \Delta)}$.

Proof. The CENTLOCAL$[\Delta \cdot \log^* n + \Delta^2]$-algorithm for OBR is given a query (v, i). The algorithm answers whether the edge $(v, p_i(v))$ is an incoming edge or an outgoing edge in the orientation. The algorithm proceeds by querying the colors of v and $p_i(v)$. The orientation of the edge $(v, p_i(v))$ is determined by comparing the colors of v and $p_i(v)$.

4 Deterministic Localization of Sequential Algorithms and Applications

A common theme in online algorithms and "greedy" algorithms is that the elements are scanned in query order or in an arbitrary order, and a decision is made for each element based on the decisions of the previous elements. Classical examples of such algorithms include the greedy algorithms for maximal matchings, $(\Delta+1)$ vertex coloring, and maximal independent set. We present a compact and axiomatic CENTLOCAL deterministic simulation of this family of algorithms, for which a randomized simulation appeared in [8]. Our deterministic simulation is based on an acyclic orientation that induces a partial order.

For simplicity, consider a graph problem Π, the solution of which is a function $g(v)$ defined over the vertices of the input graph. For example, $g(v)$ can be the color of v or a bit indicating if v belongs to a maximal independent set. (One can easily extend the definition to problems in which the solution is a function over the edges, e.g., maximal matching.)

We refer to an algorithm as a *sequential algorithm* if it fits the scheme listed as Algorithm 1. The algorithm ALG(G, σ) is input a graph $G = (V, E)$ and a bijection $\sigma : \{1, \ldots, n\} \to V$ of the vertices. Note that an element i in the domain of σ is a rank of a vertex. Hence, $\sigma(i)$ is the vertex whose rank is i, and $\sigma^{-1}(v)$ is the rank of v. The algorithm scans the vertices in the order induced by σ. It determines the value of $g(\sigma(i))$ based on the values of its neighbors whose value has already been determined. This decision is captured by the function f in Line 2. For example, in vertex coloring, f returns the smallest color that does not appear in a given a subset of colors.

Lemma 1. *Let $G = (V, E)$ be a graph, let $H = (V, A)$ be an acyclic orientation of G and let $P_{>} \subseteq V \times V$ denote the partial order defined by the transitive closure of H. Namely, $(u, v) \in P_{>}$ if and only if there exists a directed path from u to v in H. Let* ALG *denote a sequential algorithm. For every bijection $\sigma : \{1, \ldots, n\} \to V$ that is a linear extension of $P_{>}$ (i.e, for every $(u, v) \in P_{>}$ we have that $\sigma^{-1}(u) > \sigma^{-1}(v)$), the output of* ALG$(G, \sigma)$ *is the same.*

Proof. Consider two linear extensions σ and τ of $P_{>}$. Let g_σ denote the output of ALG(G, σ) and define g_τ analogously.

Algorithm 1. The sequential algorithm scheme

```
    Input: A graph $G = (V, E)$ and a bijection $\sigma : \{1, \ldots, n\} \to V$.
1:  for $i = 1$ to $n$ do
2:      $g(\sigma(i)) \leftarrow f\left(\{g(v) : v \in \Gamma(\sigma(i))\ \&\ \sigma^{-1}(v) < i\}\right)$    ▷ (Decide based on "previous" neighbors)
3:  end for
4:  Output: $g$.
```

Let

$$A_\sigma(u) \triangleq \{v \in \Gamma(u) \mid \sigma^{-1}(v) < \sigma^{-1}(u)\}\ .$$

We claim that $A_\sigma(u) = A_\tau(u)$ for every u. If v is a neighbor of u in G, then $(u, v) \in A$ or $(v, u) \in A$. We first consider the case $(u, v) \in A$. If $(u, v) \in A$, then $(u, v) \in P_>$. Hence $\sigma^{-1}(u) > \sigma^{-1}(v)$ and $\tau^{-1}(u) > \tau^{-1}(v)$ because σ and τ are linear extensions of $P_>$. We conclude that $v \in A_\sigma(u) \cap A_\tau(u)$. Similarly, if $(v, u) \in A$, then $\sigma^{-1}(u) < \sigma^{-1}(v)$ and $\tau^{-1}(u) < \tau^{-1}(v)$. This implies that $v \notin A_\sigma(u) \cup A_\tau(u)$, and hence $A_\sigma(u) = A_\tau(u)$, as required.

We prove, by induction on i, that $g_\sigma(\sigma(i)) = g_\tau(\sigma(i))$. The induction basis, for $i = 1$, holds because $\sigma(1)$ is a minimal element according to $P_>$ (a sink in H). Hence, $A_\sigma(u) = A_\tau(u) = \emptyset$. This implies that $g_\sigma(\sigma(1)) = f(\emptyset) = g_\tau(\sigma(1))$. Turning to the induction step, we prove that the claim holds for $i > 1$, assuming it holds for every $1 \le i' < i$. Let $u = \sigma(i)$.

Hence:

$$\begin{aligned} g_\sigma(u) &= f\left(\{g_\sigma(v)\}_{v \in A_\sigma(u)}\right) \\ &= f\left(\{g_\tau(v)\}_{v \in A_\sigma(u)}\right) \\ &= f\left(\{g_\tau(v)\}_{v \in A_\tau(u)}\right) = g_\tau(u)\ , \end{aligned}$$

where the second equality follows from the induction hypothesis. The third equality follows since $A_\sigma(u) = A_\tau(u)$ for every u.

Theorem 5. *For every sequential algorithm* ALG*, there exists a deterministic, stateless* CentLocal$[\Delta^{O(\Delta^2 \log \Delta)} \cdot \log^* n]$*-algorithm* ALG$_c$ *for which the following holds. For every graph G, there exists a bijection σ, such that* ALG$_c(G)$ *simulates* ALG(G, σ)*. That is, for every vertex v in G, the answer of* ALG$_c(G)$ *on query v is $g_\sigma(v)$, where g_σ denotes the output of* ALG(G, σ).

Proof. Consider the acyclic orientation H of G induced by the CentLocal$[\Delta \cdot \log^* n + \Delta^2]$-algorithm for OBR presented in Corollary 4. Let $P_>$ denote the partial order that is induced by H, and let σ be any linear extension of $P_>$ (as defined in Lemma 1). On query $v \in V$ the value $g_\sigma(v)$ is computed by performing a (directed) DFS on H that traverses the subgraph of H induced by $R_H(v)$. The DFS uses the CentLocal algorithm for OBR to determine the orientation of each incident edge and continues only along outward-directed edges[2]. The

[2] Given that the CentLocal algorithm for OBR works by running a CentLocal coloring algorithm, one can actually use the latter algorithm directly.

value of $g_\sigma(v)$ is determined when the DFS backtracks from v. Since $r_H^{\max} = \Delta^{O(\Delta^2 \cdot \log \Delta)}$, by multiplying the number of probes of the OBR algorithm and $r_H^{\max}$, we obtain that $\Delta^{O(\Delta^2 \log \Delta)} \cdot \log^* n$ probes suffice.

Corollary 6. *There are deterministic, stateless* CENTLOCAL$[\Delta^{O(\Delta^2 \log \Delta)} \cdot \log^* n]$ *algorithms for* $(\Delta + 1)$*-vertex coloring, maximal independent set, and maximal matching.*

References

1. Alon, N., Rubinfeld, R., Vardi, S., Xie, N.: Space-efficient local computation algorithms. In: SODA, pp. 1132–1139 (2012)
2. Barenboim, L., Elkin, M.: Distributed (Δ+1)-coloring in linear (in Δ) time. In: STOC, pp. 111–120 (2009)
3. Cole, R., Vishkin, U.: Deterministic coin tossing with applications to optimal parallel list ranking. Inf. and Cont. 70(1), 32–53 (1986)
4. Even, G., Medina, M., Ron, D.: Best of two local models: Local centralized and local distributed algorithms. CoRR, abs/1402.3796 (2014)
5. Goldberg, A.V., Plotkin, S.A., Shannon, G.E.: Parallel symmetry-breaking in sparse graphs. SIDMA 1(4), 434–446 (1988)
6. Kuhn, F.: Weak graph colorings: distributed algorithms and applications. In: SPAA, pp. 138–144. ACM (2009)
7. Linial, N.: Locality in distributed graph algorithms. SICOMP 21(1), 193–201 (1992)
8. Mansour, Y., Rubinstein, A., Vardi, S., Xie, N.: Converting online algorithms to local computation algorithms. In: Czumaj, A., Mehlhorn, K., Pitts, A., Wattenhofer, R. (eds.) ICALP 2012, Part I. LNCS, vol. 7391, pp. 653–664. Springer, Heidelberg (2012)
9. Mansour, Y., Vardi, S.: A local computation approximation scheme to maximum matching. In: Raghavendra, P., Raskhodnikova, S., Jansen, K., Rolim, J.D.P. (eds.) RANDOM 2013 and APPROX 2013. LNCS, vol. 8096, pp. 260–273. Springer, Heidelberg (2013)
10. H.: N Nguyen and K. Onak. Constant-time approximation algorithms via local improvements. In: FOCS, pp. 327–336 (2008)
11. Onak, K., Ron, D., Rosen, M., Rubinfeld, R.: A near-optimal sublinear-time algorithm for approximating the minimum vertex cover size. In: SODA, pp. 1123–1131 (2012)
12. Panconesi, A., Rizzi, R.: Some simple distributed algorithms for sparse networks. Dist. Comp. 14(2), 97–100 (2001)
13. Panconesi, A., Sozio, M.: Fast primal-dual distributed algorithms for scheduling and matching problems. Dist. Comp. 22(4), 269–283 (2010)
14. Parnas, M., Ron, D.: Approximating the minimum vertex cover in sublinear time and a connection to distributed algorithms. Theo. Comp. Sci. 381(1), 183–196 (2007)
15. Peleg, D.: Distributed computing: a locality-sensitive approach, vol. 5. SIAM (2000)
16. Reingold, O., Vardi, S.: New techniques and tighter bounds for local computation algorithms. CoRR, abs/1404.5398 (2014)
17. Rubinfeld, R., Tamir, G., Vardi, S., Xie, N.: Fast local computation algorithms. In: ICS, pp. 223–238 (2011)
18. Suomela, J.: Survey of local algorithms. ACM Comput. Surv. 45(2), 24:1–24:40 (2013)
19. Yoshida, Y., Yamamoto, M., Ito, H.: Improved constant-time approximation algorithms for maximum matchings and other optimization problems. SICOMP 41(4), 1074–1093 (2012)

Bicriteria Data Compression: Efficient and Usable[⋆]

Andrea Farruggia, Paolo Ferragina, and Rossano Venturini

Dipartimento di Informatica, University of Pisa, Pisa, Italy
{farruggi,ferragina,rossano}@di.unipi.it

Abstract. Lempel-Ziv's LZ77 algorithm is the *de facto* choice for compressing massive datasets (see e.g., Snappy in BigTable, Lz4 in Cassandra) because its algorithmic structure is flexible enough to guarantee very fast decompression speed at reasonable compressed-space occupancy. Recent theoretical results have shown how to design a bit-optimal LZ77-compressor which minimizes the compress size and how to deploy it in order to design a *bicriteria data compressor*, namely an LZ77-compressor which trades compressed-space occupancy versus its decompression time in a smoothed and principled way. Preliminary experiments were promising but raised many algorithmic and engineering questions which have to be addressed in order to turn these algorithmic results into an effective and practical tool. In this paper we address these issues by first designing a novel bit-optimal LZ77-compressor which is simple, cache-aware and asymptotically optimal. We benchmark our approach by investigating several algorithmic and implementation issues over many dataset types and sizes, and against an ample class of classic (LZ-based, PPM-based and BWT-based) as well as engineered compressors (Snappy, Lz4, and Lzma2). We conclude noticing how our novel bicriteria LZ77-compressor improves the state-of-the-art of fast (de)compressors Snappy and Lz4.

1 Introduction

The design of high-performing distributed storage systems — such as BigTable by Google [6], Cassandra by Facebook [3], Hadoop by Apache — requires the design of lossless data compressors which achieve effective compression ratio and very efficient decompression speed. The scientific literature abounds of solutions for this problem, named "compress once, decompress many times", but compressors running behind those large-scale storage systems are highly engineered solutions which only merely resemble the scientific results from which they are derived. The reason relies in the fact that theoretically efficient compressors are designed and analyzed in the RAM model, while their performance in practice is significantly conditioned by the numerous cache/IO misses induced by their decompression algorithms. This poor behavior is most prominent in the BWT-based compressors, such as Bzip2 and its derivatives [1,5], and it is not negligible in the LZ-based approaches (dating back to [19,20]).

⋆ This work was partially supported by MIUR of Italy under the project PRIN ARS Technomedia.

This motivated the software engineers to devise variants of Lempel-Ziv's original proposal (e.g., Snappy by Google, Lz4) which inject several software tricks having beneficial effects on memory-access locality at the cost of, however, increasing the compressed size. These compressors expanded further the known jungle of space/time trade-offs,[1] thus posing the software engineers in front of a choice: either achieve effective compression-ratios, possibly sacrificing the decompression speed (as it occurs in the theory-based results [8, 10, 11]); or try to trade compressed space by decompression time by adopting a plethora of programming tricks, which nonetheless waive any mathematical guarantees on their final performance (as it occurs in Snappy and Lz4).

Recently, it has been shown [7] that it is possible to design a bicriteria LZ77-compressor which allows to trade in a smoothed and principled way both the space occupancy (in bits) of the compressed file and the time cost of its decompression, by taking into account the underlying memory hierarchy. The key result was to design an algorithm that determines efficiently an LZ77-parsing of the input file $\mathcal{S}$ which minimizes the compressed-space occupancy (in bits), provided that its decompression time is bounded by a value T (in seconds) fixed in advance. Symmetrically, it is possible to exchange the role of the two computational resources. This problem has been solved by rephrasing the *bicriteria LZ77-parsing* problem into the well-known *Weight-constrained shortest path problem* (WCSPP) over a weighted DAG, where the goal is to search for a path whose decompression-time is at most T and whose compressed-space is minimized. This allowed to design an algorithm which solves the problem in $O(n \log^2 n)$ time and $O(n)$ working space, thus improving significantly all previously known results for the general version of WCSPP, which require $\Omega(n^2)$ time.

Very preliminary experiments [7] have shown the potential of the bicriteria LZ77-compressor (shortly, Bc-Zip) whose decompression speed is close to those one of Snappy and Lz4 (i.e., the fastest ones) and compression ratio is close to those of BWT-based and LZMA compressors (i.e., the most succinct ones). In this paper we address the following issues, which prevent the bicriteria strategy to be successful in practical settings:

– Bc-Zip deploys as a subroutine the bit-optimal LZ77-compressor devised in [11], which finds a LZ77-parsing that minimizes the compressed output (cfr. [12, 16]). Unfortunately the compressor implemented in [11], and used in [7], was slow and not optimal in asymptotic sense, though it was superior to the heuristics introduced in [2, 13, 17].
– the decompression efficiency of Bc-Zip relies heavily on the estimation of LZ77 decompression time. This was addressed in [11] by proposing an interpolation approach which required a "training" dataset and deployed many parameters, losing accuracy and generalization.

The main contributions of this paper are the following:

– we propose a novel bit-optimal LZ77-compressor which is simpler, cache-aware and asymptotically optimal, thus resulting faster in practice

[1] See e.g., `http://mattmahoney.net/dc/text.html`.

(see Section 3). This represents an important step in closing the compression time gap with the widely known compressors Gzip, Bzip2 and Lzma2.
- we introduce a new model for estimating the decompression time. This model is based on few measurable parameters that depend only on the underlying machine and, thus, result independent of the file to be compressed/decompressed. Given this model we design a *calibration tool* which automatically derives the model, achieving an average error of $\approx 5.6\%$; this is quite satisfactory according to [14]. Due to space limitations, its technical discussion is deferred to the journal version of this paper.
- we finally evaluate the novel bit-optimal LZ77-compressor and Bc-Zip by investigating many algorithmic and implementation issues (see Section 4): integer encoders, block lengths, dataset types, ample set of classic (LZ-based, PPM-based and BWT-based) as well as engineered compressors (Snappy, Lz4, Lzma2 and Ppmd). We perform many experiments aimed at measuring the impact of those features, so leading to the design of a compressor that surpasses the decompression performance of well engineered and widely used compressor Lz4 on three out of four datasets.

The ultimate result achieved by this paper is a deep and variegate understanding of the novel bicriteria compression technology both in terms of efficacy and efficiency issues under various experimental scenarios. We will make available to the scientific community this large implementation effort by providing the datasets, the whole experimental setting and the C++ code of Bc-Zip.

2 Background

The *bit-optimal LZ77-parsing problem* asks for a LZ77 parsing of a text $\mathcal{S}[1, n-1]$ whose compressed representation requires minimum space (in bits).

A LZ77-parsing of a text $\mathcal{S}$ is a decomposition of $\mathcal{S}$ in m substrings (phrases) of the form $p = \mathcal{S}[s, s+\ell-1]$ such that either $p = \mathcal{S}[s]$ is a single *character* (hence $\ell = 1$), or it is $\ell > 1$ and thus $\mathcal{S}[s, s+\ell-1] = \mathcal{S}[s-d, s-d+\ell-1]$ is a text substring of length ℓ *copied* from d positions before in $\mathcal{S}$. Clearly, many candidate copies might occur in $\mathcal{S}$, each having a different length and distance, so the possible LZ77-parsings of $\mathcal{S}$ may be numerous. Each of these LZ77-parsings induces a compressed version of $\mathcal{S}$ which is obtained by, first, substituting each phrase p with the pair $\langle 0, \mathcal{S}[s] \rangle$, if p is a single character, and with $\langle d, \ell \rangle$, otherwise; and then encoding each of those pairs with a pair of variable-length binary codewords which are computed by means of two (possibly different) integer encoders enc_d and enc_ℓ. For the sake of clarity, we drop the subscripts whenever the argument, either distance or length, allows us to disambiguate the encoder in use.

An important assumption of the bit-optimal approach, as of [9, 11], is that the integer encoders satisfy the so-called *non-decreasing cost property*, which is satisfied by most encoders adopted in modern compressors. An integer encoder enc satisfies the non-decreasing cost property if $|\mathsf{enc}(n)| \leq |\mathsf{enc}(n')|$ for all $n \leq n'$. Moreover, these encoders must be *stateless*, that is, they must always encode the same integer with the same bit-sequence.

More formally, given a text $\mathcal{S}$ and a pair of encoders enc_ℓ and enc_d, the bit-optimal LZ77-parsing problem asks thus for a LZ77 parsing of $\mathcal{S}$ which minimizes the compressed size when using enc_ℓ and enc_d as integer encoders. Authors of [9, 11] modeled the *bit-optimal LZ77-parsing problem* as a *single-source shortest path* problem over a graph $\mathcal{G}$, consisting of $n = |\mathcal{S}| + 1$ nodes (one per $\mathcal{S}$'s character, plus a sink node) labeled with the integers $\{1, \ldots, n\}$. In particular, there is (i) a node i associated to each character $\mathcal{S}[i]$; (ii) an edge $(i, i+1)$ for every $i < n$, and (iii) an edge $(i, j+1)$ iff the substring $\mathcal{S}[i, j]$ occurs earlier in the text. It follows that each edge (i, j) is in bijective correspondence with a candidate phrase of the LZ77-parsing of $\mathcal{S}$.

This graph has a number of properties: (i) it is directed and acyclic, and (ii) there is a bijection between LZ77-parsings of $\mathcal{S}$ and paths from 1 to n in $\mathcal{G}$. Since each edge is associated to a phrase, it can be weighted with the length, in bits, of its codeword. In particular, edges $(i, i+1)$ are assumed to have constant weight, since they correspond to the single-character phrase $\langle 0, \mathcal{S}[i]\rangle$; while edges $(i, j+1)$ are weighted with the value $|\mathsf{enc}(d)| + |\mathsf{enc}(\ell)|$ provided that $\langle d, \ell\rangle$ is the associated codeword. Given that $\mathcal{G}$ is a DAG, computing a shortest path from 1 to n is simple and takes $O(m)$ time and space. But there are strings for which $m = \Theta(n^2)$, so this algorithm is not practical even for files of a few tens of MiBs.

Starting from these premises, this problem was attacked in [9, 11] by introducing two main ideas: (i) prune $\mathcal{G}$ to a significantly smaller subgraph which preserves the shortest path of $\mathcal{G}$ from nodes 1 to n; (ii) generate on-the-fly this subgraph, thus minimizing the working space of the shortest-path computation.

The *pruning strategy* consists of retaining, for each node, only the *maximal edges*, that is, edges of maximum length among those with equal cost (in bits, according to enc). It has been shown [11] that the number of maximal edges depends on the structure of enc_d and enc_ℓ, but it is $O(n \log n)$ for the vast majority of encoders. The key algorithmic issue was then to show how to generate the maximal edges outgoing from a given node i, incrementally along with the shortest-path computation, taking $O(1)$ amortized time per edge and only $O(n)$ auxiliary space. This task is called *Forward Star Generation* (shortly, FSG).

The algorithm originally described in [11] involves the construction of suffix arrays and compact tries of several substrings of $\mathcal{S}$ (possibly transformed in proper ways) so that, although optimal asymptotically, this algorithm is not practical. In the next section we show a new algorithm which is optimal asymptotically and much simpler than the algorithm proposed in [11], since it is based solely on lists and their sequential scans.

3 Bit-Optimal Compression: Faster and Practical

The Forward Star Generation task asks to compute all maximal edges spurring from a node i only when needed, and discarded afterwards. An edge is *maximal* if it is either d-maximal or ℓ-maximal (or both). An edge spurring from vertex i and represented by a LZ77 phrase $\langle d, \ell\rangle$ is *d-maximal* if it is the longest LZ77 phrase taking at most $|\mathsf{enc}(d)|$ bits for representing its distance component; ℓ-maximality is defined similarly. Finding ℓ-maximal edges is easy once d-maximal edges are known, since the strategy consists on “splitting” d-maximal phrases according to the cost classes of enc_ℓ, so here we concentrate on finding those d-maximal edges.

Let us now consider a *cost class* of enc_d, that is, the maximal sub-range $[l, r]$ of $[1, n]$ such that each integer between l and r takes exactly c bits by using encoder enc_d, for some c. There is one d-maximal edge for each cost class.

Let us take the d-maximal edge, say $(i, i + \ell)$, for the cost class $[l, r]$. We can infer that the substring $\mathcal{S}[i, i+\ell]$ is the longest substring starting at i and having a copy at distance within $[l, r]$ because the subsequent edge $(i, i + \ell + 1)$ denotes a longer substring whose copy-distance must therefore occur in a farther back subrange (because of d-maximality).

Maximal edges leaving from i can be found by considering, for each cost class, the suffix array of $\mathcal{S}$ restricted to positions i and $[i - r, i - l]$, which we denote as Rsa. In fact, the d-maximal edge can be found by looking for the lexicographic *predecessor* and *successor* of $\mathcal{S}[i, n]$ in Rsa and taking the one with the longest common prefix to $\mathcal{S}[i, n]$. The selected suffix thus is the copy-reference of the corresponding d-maximal edge. This strategy, however, is inefficient, because Rsa cannot be computed in less than $\Omega(r-l) = \Omega(n/\log n)$ time when the number of d-maximal edges is $O(\log n)$. We overcome efficiency problems related to building indexing data structures (like the Rsa) by computing *en ensemble* the d-maximal edges of $O(r - l)$ vertexes. To do that, we extend the simple strategy outlined above by looking simultaneously for predecessors/successors of a set of suffixes.

More precisely, let us denote $B = [i, i + r - l]$ as the range of positions for which we would like to determine the d-maximal edges, and $W = [i-r, i+r-2l]$ the set of potential *back-references*. Notice that $|W| = 2(r - l)$, so $|B \cup W|$ is at most $3(r - l)$ (less if they overlap). Let us then denote Rsa as the suffix array restricted to positions in $B \cup W$. The main idea is to find all successors of suffixes starting in B with a left-to-right scan of Rsa, and all predecessors with a right-to-left scan. During the scan, we keep a queue $\mathcal{Q}$ of positions in B for which we did not have found yet their matching predecessor/successor. The queue is kept sorted in ascending order. So, let us assume the element of Rsa currently examined is in W but not in B. This means that it may be a successor for *some* of the positions in $\mathcal{Q}$, and so we have to determine those positions and remove them from the queue. We underline that not every position in $\mathcal{Q}$ may apply, because the distance between the current element in W and the element in the queue may be greater than r. However, since positions in $\mathcal{Q}$ are sorted by increasing position, those can be found in optimal $O(1)$ time per match by examining the queue starting from the first element, and stopping whenever the current element in the queue has distance greater than r.

Let us now consider the case when the currently examined element in Rsa is a position j in B. This implies that we have to insert j in $\mathcal{Q}$ while maintaining it sorted. This operation cannot be performed in constant time, since the element may be inserted in the middle of $\mathcal{Q}$. However, since distance between positions in B cannot be greater than r, positions in $\mathcal{Q}$ greater than j can be matched with j itself and removed from $\mathcal{Q}$, so j can be appended at the bottom of the queue in constant time. It is clear that the time complexity is proportional to the number of examined/discarded elements in/from $\mathcal{Q}$, which is $|B|$, plus the cost of scanning Rsa, which is at most $|W|+|B|$. This implies that each maximal edge is found in amortized $O(1)$ time.

An important part is the efficient and on-the-fly generation of the Rsa of suffixes starting in $W \cup B$. Due to space limitations, we only sketch two optimal solutions, which will be illustrated in the journal version of this paper. The first one, general but less practical, is based on the *Sorted Range Reporting* data structure [4]. The other solution makes some (generally satisfied) assumptions about the integer encoders in use, and it is yet optimal but more practical because it is based only on lists scanning. The last one is at the core of our implementation of the Bc-Zip compressor tested in Section 4.

4 Experiments

In this section we show the effectiveness of this novel "optimization approach" over LZ77-based compression in a throughout and conclusive way. Due to space limitations, here we will only highlight the most important results, omitting many figures and technical details. A more thorough illustration will be available in the journal version of this paper.

In the first part of this section we will evaluate the advantage of the bit-optimal parsing against Greedy, the most popular LZ77 parsing strategy, and many high-performance compressors. We will show that the bit-optimal parsing has a clear advantage over heuristically highly engineered compressors, thus justifying the interest in the technology. We will also show that the novel Fast-FSG algorithm exposed in Section 3 helps to bring down considerably the compression time, thus making bit-optimal parsing a solid and practical technology.

In the second part we will compare bicriteria data compression against the most common approach of trading decompression time for compression ratio. In fact, many practical LZ77 implementations (e.g., Gzip, Snappy, Lz4 etc.) employ the bucketing strategy (that is, splitting the file in blocks (buckets) of equal size which are individually compressed and then concatenated to produce the compressed output) or a *moving window* to (hopefully) lower decompression time by limiting the maximum distance at which a phrase may be copied, thus forcing spatial locality. Interestingly enough, we will show that this approach is not the best one to speed up the file decompression because basically it takes into account neither integer decoding time nor the length of the copied string, which may be relevant in some cases and could amortize the cost of long but far copies. We will validate this argument by also introducing a *decompression time model* which properly infers the decompression time of a LZ77-parsing from a small set of features (such as number of copies, distances distribution, etc). This model will be used in order to efficiently determine proper edge-costs in the graph over which the WCSPP is solved.

We will finally show the vast time/space trade-off achievable with the bicriteria strategy, which improves *simultaneously* both the most succinct (like Bzip2) and the fastest (like Lz4) compressors.

Experimental settings: We implemented the compressor in C++11, and we compiled it with Intel C++ Compiler 14 with flags `-O3 -DNDEBUG -march=native`. According to the applicative scenario we have in mind, we used two machines to carry out the experiments. The first machine, used in compression, is equipped with AMD Opteron 6276 processors, with 128GiB of memory; the second machine, used in decompression, is equipped with an Intel Core i5-2500, with 8GiB of DDR3 1333MHz memory. Both machines run Ubuntu 12.04.

Experiments were executed over 1GiB-long (2^{30} bytes) datasets of different types: (i) Census: U.S. demographic informations in tabular format (type: database); (ii) Dna: collection of families of genomes (type: highly repetitive biological data); (iii) Mingw: archive containing the whole mingw software distribution (type: mix of source codes and binaries)[2]; (iv) Wikipedia: dump of English Wikipedia (type: natural language). Each dataset have been obtained by taking a random chunk of 1GiB from the complete files. The whole experimental setting (datasets and C++ code) is available at `http://acube.di.unipi.it/bc-zip/`.

We experimented various integer encoders for the LZ77 phrases: Variable Byte (VByte), 4-Nibble (Nibble), and Elias' γ (Gamma) and δ (Delta) [15,18]. We also introduce two variants of those encoders, called VByte-Fast and T-Nibble, which perform particularly well on LZ77 phrases.

In the design of our compressor we modified the LZ77 scheme to allow the encoding of runs of literals in just one phrase. This twist has beneficial effects on both decompression speed and compression ratio on incompressible files when using the bit-optimal strategy, and introduces a very effective way of controlling the space/time trade-off when using the Bicriteria strategy.

Speed improvements over the novel bit-optimal compressor: In Figure 1 we compare the running time of the bit-optimal LZ77 algorithm when employing either the original subroutine for generating maximal edges (shortly FSG), as proposed in [11], or our novel Fast-FSG algorithm, described in Section 3. Figure 1 shows the results only for dataset Wikipedia, as the figures do not change significantly on the other datasets. We compared the compression ratios produced by two integer encoders — namely, Gamma and VByte-Fast — which are the ones that yield the lowest and highest performance gaps.

In the plots, Fast-FSG and FSG significantly diverge in running time, reflecting the different time complexities (constant vs $O(\log b)$ per edge, where b is the size of the bucket). In the Gamma case, the running time for the 1 MiB bucket-size are nearly the same for FSG and Fast-FSG, while the gap is already $\approx$ 4x for a 1GiB bucket-size. In the VByte-Fast case, gap ranges from $\approx$ 1.3x to $\approx$ 2.5x.

The improvements introduced by Fast-FSG make the bit-optimal LZ77-compressor much closer to the widely used and top performing compressors in compression time. In fact, our bit-optimal construction is on-par or faster than Lzma2.

Bit-optimal performance: According to our experiments, integer encoders T-Nibble and VByte-Fast are the most interesting in terms of compression ratio

[2] Thanks to Matt Mahoney – `http://mattmahoney.net/dc/mingw.html`.

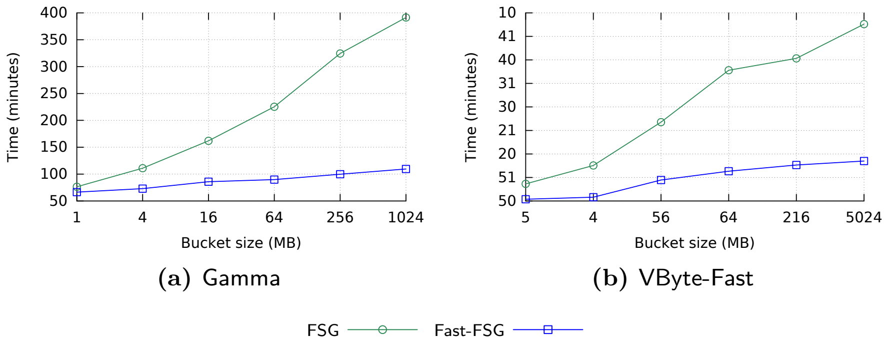


Figure 1. Comparison between the novel Fast-FSG and the previously known FSG in parsing the dataset Wikipedia by using VByte-Fast and Gamma as integer encoders. The construction time is reported by varying the bucket size.

and decompression speed (respectively, the most succinct and the fastest). For this reason we restrict our attention to these two encoders in the next experiments. In our tests, the bit-optimal strategy produces parsing which are more than 10% smaller on average than greedy ($\approx$ 11.5% with VByte-Fast, $\approx$ 14.6% with T-Nibble), in which the rightmost longest match is always selected, while being $\approx$ 15% faster at decompression. Working space was $\approx$ 59GiB of main memory.

Using bit-optimal in lieu of greedy means that we can use a faster encoder without sacrificing compression ratios. Bit-optimal achieves this result by "adapting" parsing choices to the ideal symbol probability distribution of the underlying integer encoders.

Table 1 compares the bit-optimal strategy (called `LzOpt`) against the best known compressors to date. With respect to the most space-efficient compressors (Lzma2, Bzip2, Ppmd, and BigBzip), compression ratio is, overall, only slightly worse: the gap with Lzma2, the most succinct compressor, ranges from 15% (Census) to 25% (Dna), with Mingw representing a situation in which the combination of bit-optimal parsing plus literal encodings let `LzOpt` be the most succinct compressor. Notice that Lzma2 reports better compression ratios than `LzOpt` due to its choice of a different, *statistical* encoder, whereas `LzOpt` is restricted to the use of *stateless* ones (see discussion on Section 2). On the other hand, `LzOpt` decompression time is order of magnitudes better than these approaches.

Comparing with the fastest compressors (Snappy and Lz4), parsings obtained with `LzOpt` are way more succinct: the relative gap of those compressors in compressed space ranges from $\approx$ 60% (Census) to over 1,300% (Dna). Decompression speed is already very competitive, especially if the slightly less succinct VByte-Fast encoder is taken into account. We will close the speed gap w.r.t. Snappy and Lz4 with the Bicriteria Data Compression scheme.

Table 1. Comparison between bit-optimal compressor (LzOpt), bicriteria compressor (Bc-Zip) and state-of-the-art data compressors. For each dataset we highlight the parsing having the closest decompression time to Lz4.

Dataset	Compressor	Compressed size (MBytes)	Decompression time (msecs)
Census	LzOpt (T-Nibble)	38.08	776
	LzOpt (VByte-Fast)	40.19	572
	Bc-Zip (VByte-Fast, 556 ms)	40.38	549
	Bc-Zip (VByte-Fast, 494 ms)	41.63	506
	Bc-Zip (VByte-Fast, 454 ms)	44.42	462
	Gzip	48.23	2,472
	Lzma2	33.03	2,652
	Snappy	123.68	634
	Lz4	61.82	454
	Bzip2	39.96	15,054
	BigBzip	33.28	71,000
	Ppmd	38.70	38,000
Mingw	LzOpt (T-Nibble)	179.01	1,586
	LzOpt (VByte-Fast)	192.34	954
	Bc-Zip (VByte-Fast, 920 ms)	193.77	845
	Bc-Zip (VByte-Fast, 726 ms)	205.56	695
	Bc-Zip (VByte-Fast, 461 ms)	293.62	472
	Gzip	344.47	5,534
	Lzma2	187.68	8,323
	Snappy	461.00	891
	Lz4	384.67	726
	Bzip2	317.96	32,469
	BigBzip	222.22	152,000
	Ppmd	245.54	414,000
Dna	LzOpt (T-Nibble)	23.78	598
	LzOpt (VByte-Fast)	25.14	482
	Bc-Zip (VByte-Fast, 455 ms)	27.97	468
	Bc-Zip (VByte-Fast, 418 ms)	47.59	432
	Bc-Zip (VByte-Fast, 381 ms)	75.08	395
	Gzip	245.25	5,815
	Lzma2	17.62	1,681
	Snappy	448.67	1,301
	Lz4	333.74	1,007
	Bzip2	45.79	34,157
	BigBzip	42.02	152,000
	Ppmd	196.36	129,000
Wikipedia	LzOpt (T-Nibble)	175.86	3,080
	LzOpt (VByte-Fast)	191.19	1,748
	Bc-Zip (VByte-Fast, 1306 ms)	205.89	1460
	Bc-Zip (VByte-Fast, 973 ms)	270.35	1106
	Bc-Zip (VByte-Fast, 862 ms)	316.18	986
	Gzip	269.36	6,154
	Lzma2	166.16	9,871
	Snappy	422.80	1,093
	Lz4	309.51	862
	Bzip2	214.65	29,037
	BigBzip	150.88	151,000
	Ppmd	148.27	283,000

Effectiveness of the bucketing strategy: Overall, experiments confirm that compression ratio does improve with a longer bucket size, but the exact improvement does depend on the peculiarities of the data being compressed. This implies that trading decompression time vs compression ratio via the choice of a proper bucket size requires a deep understanding of the data being compressed. In Figure 2 we show the decompression time when varying the bucket size. We only plot the results for Wikipedia and Dna because they suffice to capture the range of behaviors shown by the bit-optimal LZ77-compressor over our four datasets. In particular, we mention that the behavior on Census and Mingw is similar to that on Wikipedia, with the former reaching with VByte-Fast a decompression speed up to 2, 200 MiB/sec with 4-MiB buckets that decreases to 1, 800MiB/sec with 1-GiB buckets, while over Mingw the decompression speed is about 1, 100MiB/sec for any bucket size. For both datasets, the speed is roughly halved when using T-Nibble, the reason being the word-boundary alignment of VByte-Fast's codewords which removes the need of bit-shifting them when reading from memory.

Figure 2 also shows that the dependency between the bucket size and the decompression speed of the bit-optimal LZ77-output highly depends on the characteristics of the data being compressed, but (possibly) in a counter-intuitive way. In Wikipedia, it generally decreases with larger bucket sizes, with a peek somewhere near 4MiB, instead of 1MiB; in Dna the decompression speed *improves* with larger bucket sizes. In Dna, which is highly repetitive, there are far back-references which copy long portions of a genome which compensate the cache

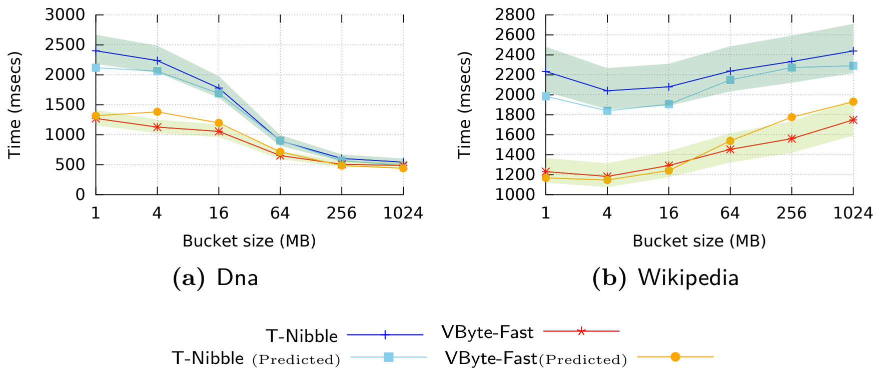


Figure 2. Decompression time by varying the bucket size on Dna and Wikipedia. The plot reports also the time predicted by our decompression-time model and a band around the decompression time capturing a relative error of 10%.

miss penalty induced by the copy (fewer phrases). On the other hand, Wikipedia is less repetitive and so far back-references added by larger windows are not much long, save little space, and thus they do not compensate the miss penalty incurred by their decompression. It is evident now that if we want to trade in a principled way decoding time *versus* compressed space, and thus ultimately improve the design of the bicriteria compressor Bc-Zip, we need to precisely explain and, thus, predict these phenomena. We designed a time model (full description in the journal version) which is capable of predicting decompression time with an average precision of $\approx 5.6\%$, which is a remarkable achievement as accurately predicting running times (that is, achieving an average precision of 10% or better) is notoriously an hard task [14]. In Figure 2 we plotted the predicted decompression time alongside actual decompression times. This model takes into account the cache miss latency to access distant substrings, the phrase decoding and the copying time. Thanks to this time model, Bc-Zip is capable of trading decompression time for compression ratio in a smooth and *consistent* way, as shown in the next paragraph.

Bicriteria compressor: Our implementation of the Bc-Zip compressor largely follows the scheme exposed in [7], using the Fast-FSG algorithm exposed in Section 3 and some minor algorithmic twists to accelerate compression. In our tests we compressed each dataset several times, for both VByte-Fast and T-Nibble, with time bounds ranging from the decompression time of the time-optimal parsing to the decompression time of the space-optimal one. In this way we can determine the whole range of trade-offs offered by Bc-Zip. Moreover, in order to directly compare Bc-Zip against the state-of-the-art compressors adopted in storage systems (such as Hadoop and BigTable), we compressed each dataset by setting its decompression-time bound (or compressed-space bound) as the decompression

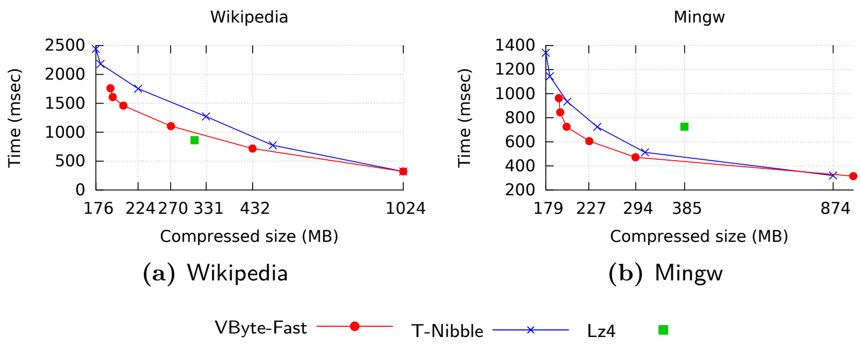


Figure 3. Space/time trade-off curve obtained with Bc-Zip, by varying the decompression time bound, and Lz4

time of the parsings generated by Lz4 (highlighted entries in Table 1). The average time model accuracy is ≈ 4.5% (VByte-Fast ≈ 5.4%, T-Nibble ≈ 3.7%).

Table 1 and Figure 3 show the large range of trade-offs obtained by Bc-Zip. For instance, in Mingw spans from ≈ 300 msec to ≈ 1, 400 msec time-wise, and from ≈ 976MB to ≈ 179MB space-wise. Another interesting aspect is that T-Nibble is competitive against VByte-Fast only when maximum compression is required, otherwise the latter delivers more succinct parsings for the same decompression time. This is due to T-Nibble's relatively slow decoding, which forces the compressor to trade LZ77 copies for literals in order to meet the decompression time budget. This is in contrast with VByte-Fast's fast decoder, which does not impact much on decompression time and thus the compressor only cares about the cache behavior by substituting cache miss-inducing copies for a sequence of miss-free ones, a more succinct time-saving strategy.

Another interesting observation is that varying the decompression time impacts little on compressed size when more succinct parsings are considered, while it may impact considerably when less space-efficient parsings are taken into account (with varying degree: more accentuate with Mingw, less with Wikipedia). This provides a quantitative explanation of the natural question that motivated the work in [7], namely *"who cares whether the compressed file is slightly longer if this allows to improve significantly the decompression speed?"*. The present paper shows that the space/time trade-offs do not change linearly but, instead, a small change in one resource may induce a significant change in the other, unpredictably.

Moreover, Bc-Zip is extremely competitive with Lz4, since it clearly dominates it in three out of four datasets (Census, Dna, Mingw), while in Wikipedia it is very close, being only ≈ 12% slower and ≈ 2% less succinct than Lz4. Overall, Bc-Zip performs more consistently thanks to its well-principled design, and surpasses the performance of well engineered, and widely used, compressor Lz4 on three out four datasets. We therefore believe that this is a nice success case of a win-win situation between algorithmic theory and engineering.

References

1. Adjeroh, D., Bell, T., Mukherjee, A.: The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching. Springer (2008)
2. Békési, J., Galambos, G., Pferschy, U., Woeginger, G.J.: Greedy algorithms for on-line data compression. J. Algorithms 25(2), 274–289 (1997)
3. Borthakur, D., et al.: Apache Hadoop goes realtime at Facebook. In: SIGMOD, pp. 1071–1080 (2011)
4. Brodal, G.S., Fagerberg, R., Greve, M., López-Ortiz, A.: Online sorted range reporting. In: Dong, Y., Du, D.-Z., Ibarra, O. (eds.) ISAAC 2009. LNCS, vol. 5878, pp. 173–182. Springer, Heidelberg (2009)
5. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. Tech. Rep. Digital (1994)
6. Chang, F., et al.: Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems 26(2) (2008)
7. Farruggia, A., Ferragina, P., Frangioni, A., Venturini, R.: Bicriteria data compression. In: SODA, pp. 1582–1595 (2014)
8. Ferragina, P., Giancarlo, R., Manzini, G., Sciortino, M.: Boosting textual compression in optimal linear time. Journal of the ACM 52, 688–713 (2005)
9. Ferragina, P., Nitto, I., Venturini, R.: On the bit-complexity of Lempel-Ziv compression. In: SODA, pp. 768–777 (2009)
10. Ferragina, P., Nitto, I., Venturini, R.: On optimally partitioning a text to improve its compression. Algorithmica 61(1), 51–74 (2011)
11. Ferragina, P., Nitto, I., Venturini, R.: On the bit-complexity of Lempel-Ziv compression. SIAM Journal on Computing (SICOMP) 42(4), 1521–1541 (2013)
12. Katajainen, J., Raita, T.: An analysis of the longest match and the greedy heuristics in text encoding. Journal of the ACM 39(2), 281–294 (1992)
13. Klein, S.T.: Efficient optimal recompression. Computer Journal 40(2/3), 117–126 (1997)
14. Huang, L., Jia, J., Yu, B., Chun, B., Maniatis, P., Naik, M.: Predicting execution time of computer programs using sparse polynomial regression. In: NIPS, pp. 883–891 (2010)
15. Salomon, D.: Data Compression: the Complete Reference, 4th edn. Springer (2006)
16. Schuegraf, E.J., Heaps, H.S.: A comparison of algorithms for data base compression by use of fragments as language elements. Information Storage and Retrieval 10(9-10), 309–319 (1974)
17. Smith, M.E.G., Storer, J.A.: Parallel algorithms for data compression. Journal of the ACM 32(2), 344–373 (1985)
18. Witten, I.H., Moffat, A., Bell, T.C.: Managing Gigabytes: Compressing and Indexing Documents and Images. Morgan Kaufmann Publishers (1999)
19. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transaction on Information Theory 23, 337–343 (1977)
20. Ziv, J., Lempel, A.: Compression of individual sequences via variable-rate coding. IEEE Transactions on Information Theory 24(5), 530–536 (1978)

Amortized $\tilde{O}(|V|)$-Delay Algorithm for Listing Chordless Cycles in Undirected Graphs[⋆]

Rui Ferreira[1], Roberto Grossi[2], Romeo Rizzi[3], Gustavo Sacomoto[4,5], and Marie-France Sagot[4,5]

[1] Microsoft Bing, UK
[2] Università di Pisa, Italy
[3] Università di Verona, Italy
[4] INRIA Grenoble Rhône-Alpes, France
[5] UMR CNRS 5558 - LBBE, Université Lyon 1, France

Abstract. Chordless cycles are very natural structures in undirected graphs, with an important history and distinguished role in graph theory. Motivated also by previous work on the classical problem of listing cycles, we study how to list chordless cycles. The best known solution to list all the C chordless cycles contained in an undirected graph $G = (V, E)$ takes $O(|E|^2 + |E| \cdot C)$ time. In this paper we provide an algorithm taking $\tilde{O}(|E| + |V| \cdot C)$ time. We also show how to obtain the same complexity for listing all the P chordless st-paths in G (where C is replaced by P).

1 Introduction

A *chordless (induced) cycle* c in an undirected graph G is a cycle such that the subgraph induced by its vertices contains exactly the edges of c. A chordless cycle is called a *hole* when its length is at least 4. Similarly, a *chordless (induced) path* π in G is such that the subgraph of G induced by π contains exactly the edges of π. Both chordless cycles and paths are very natural structures in undirected graphs with an important history, appearing in many papers in graph theory related to chordal graphs, perfect graphs and co-graphs (e.g. [11,6,3]), as well as many NP-complete problems involving them (e.g. [2,7,9]).

In this paper we consider algorithms for listing chordless cycles and st-paths in an undirected graph $G = (V, E)$, with $n = |V|$ vertices and $m = |E|$ edges, motivated by the algorithms for listing cycles and st-paths that have been produced by an active area of research since the early 70s [10,13,1].

In this paper we present an algorithm for listing all the C chordless cycles in an undirected graph $G = (V, E)$ in $\tilde{O}(m+n \cdot C)$ time, hence with an amortized $\tilde{O}(n)$ time delay, where $\tilde{O}(f(n, m))$ is used as a shorthand for $O(f(n, m) \operatorname{polylog} n)$. We also show that the same algorithm may be used to list all the P chordless st-paths in $\tilde{O}(m + n \cdot P)$ time, hence amortized $\tilde{O}(n)$ time delay.

[⋆] GS and MFS were partially supported by the ERC programme FP7/2007-2013 / ERC grant agreement no. [247073]10, and the French project ANR-12-BS02-0008 (Colib'read). RG was partially supported by Italian project PRIN 2012C4E3KT (AMANDA).

There are very few algorithms in the literature for listing chordless cycles and/or paths, where some of them have no guaranteed performance [12,16]. The most notable and elegant listing algorithm is by Uno [15], with a cost of $O(m^2 + m \cdot C)$ time for chordless cycles and $O(m^2 + m \cdot P)$ time for chordless st-paths, hence amortized $O(m)$ time delay.

2 Preliminaries

Our graphs are finite, undirected, and *simple*, *i.e.* without self-loops or parallel edges. Given a graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges, our task is to list out fast all its chordless cycles. We hence assume that G is connected. Given $V' \subseteq V$, we denote by $E\langle V'\rangle := \{uv \in E \mid u, v \in V'\}$ the set of those edges which are contained in V'. A graph $G' = (V', E')$ is called a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$. The subgraph G' is called *induced* (or *chordless*) if $E' = E\langle V'\rangle$. For any $V' \subseteq V$, we denote by $G[V'] := (V', E\langle V'\rangle)$ the *subgraph of* G *induced by* V'. Given $e \in E$, we denote by $G \setminus e := (V, E \setminus \{e\})$ the subgraph obtained from G by *deleting* the edge e. Given $v \in V$, we denote by $G \setminus v := G[V \setminus \{v\}]$ the subgraph obtained from G by first deleting all the edges incident to v, and then removing the isolated vertex v. Given a vertex $u \in V$, we denote by $N_G(u) := \{v \in V \mid uv \in E\}$ the *neighbourhood* of u, the subscript is omitted whenever the graph is clear from the context.

A *cycle* is a connected graph in which every vertex has degree 2. A *path* is a connected graph in which every vertex has degree 2 except for two degree-1 vertices, s and t, called the *endvertices* of the path. This is also called an *st-path* and denoted by π_{st}. Indeed, when building a path from s to t edge after edge, it will be most natural, and more precise, to think like we are orienting the traversed edges. For this reason, we will also write (u, v) for an edge that, when building a path, has been traversed from u to v.

A (chordless) path (or cycle) of G is a (chordless) subgraph of G which is a path (or cycle). We denote by $\mathcal{C}(G)$ the set of all chordless cycles in G. We denote by $\mathcal{P}(G)$ (by $\mathcal{P}_{st}(G)$) the set of all chordless paths (st-paths) in G. When $s = t$, we get those cycles visiting s. We refer to a path $\pi \in \mathcal{P}(G)$ by its natural sequence of vertices or edges. A *hole* is a chordless cycle of size at least 4. Thus $\mathcal{C}(G)$ comprises holes and triangles. Since there are at most mn triangles, our algorithm can be used to list the holes of G in $\tilde{O}(n)$ time each, with an overall $\tilde{O}(mn^2)$ additive time cost.

Uno [15] proposed an algorithm that lists each chordless cycle in an undirected graph $G = (V, E)$ in $O(m)$ time while using $O(m)$ space. The first step is the following reduction to the problem of enumerating the chordless st-path in a graph G. Based on the fact that for any vertex $s \in V$ the chordless cycles in $G \setminus s$ are also chordless cycles in G, the algorithm proceeds by listing all chordless cycles passing through s; and repeating the process in $G \setminus s$, until the graph is empty. Then, to list all chordless cycles passing through s in $G' = G$, the algorithm follows the approach of listing the chordless paths $s \rightsquigarrow t$ in $G' \setminus (s, t)$, for each $t \in N_{G'}(s)$; and to avoid duplications, at the end of each iteration the graph is updated to $G' = G' \setminus t$.

Given a previously computed chordless st-path $\pi = v_0 v_1 \dots v_l$, Uno's algorithm identifies the set of vertices $U \subseteq V$ such that each $u \in U$ is adjacent to some $v_j \in \pi$, and the edge (v_j, u) is contained in a chordless st-path $\pi' \neq \pi$ extending the prefix $\pi_j = v_0 v_1 \dots v_j$. The algorithm is kick-started by taking a shortest st-path (as a shortest path has the property of also being a chordless path) and employs a recursive strategy of vertex removal to avoid listing the same chordless path multiple times. This ensures that each chordless path is listed once. Uno's algorithm takes $O(m)$ time to compute U and prepare the recursive calls before it either outputs a new path or stops. The total time is therefore $O(m^2 + m \cdot |\mathcal{C}(G)|)$.

3 Our Approach and Key Ideas

We outline the main ideas which allow us to reduce the amortized cost for a chordless cycle from $O(m)$ to $\tilde{O}(n)$, giving a total $\tilde{O}(m + n \cdot |\mathcal{C}(G)|)$ time to list all the chordless cycles. Our approach relies on a variant of the *cleaning* operation introduced in [6] to recognize linear balanced matrices and even holes in graphs [4,5].

3.1 Certificates for Chordless st-path

A listing algorithm usually takes the form of a recursive procedure exploring the space of all solutions. A key idea employed since the first listing papers [10] is to check for the existence of at least one solution before branching, *i.e.* before partitioning the solution space in subspaces to be assigned to the children. This avoids unproductive recursive calls, *i.e.* calls that do not list any solution and whose overhead cost could completely dominate the cost of reporting the solutions (*e.g.* see [14]). In a previous work [1], we stressed the notion of certificate since, in a more refined recursive scheme, passing a certificate of existence as an extra parameter may facilitate the work of the children which may avoid running the existence check: if they have a single child, they could be done by just passing the certificate received as an input or a small adaptation of it. We also saw that more structural facts around the certificate could be useful. For the case of st-paths [1], the certificate is a DFS tree rooted in s and reaching t, which contained an st-path and also helped in other ways. Until now, the certificate was itself a solution or explicitely contained one.

Here we try out something new: what if our certificate guarantees the existence of a solution but is *not* itself a solution? The following fact suggests that the certificate for the existence of a chordless st-path might be just any st-path.

Fact 1. *Given two vertices s, t in G, there is a* chordless *st-path in G iff there is an st-path in G.*

Thus we allow for certificates which are somewhat less refined than actual solutions, in the same spirit that a binary heap demands a less strict and lazy notion of order. This is a new asset of the notion of certificate and opens up new possibilities.

3.2 From Chordless Cycles to Chordless *st*-paths

Uno [15] shows how to reduce listing chordless cycles in a graph to listing chordless st-paths for all edges (s,t) chosen in a specific order (see Section 2), which is necessary to avoid duplications in the output. The initialization step for each edge (s,t) takes $O(m)$ time as it requires to find one chordless st-path. This gives the m^2 term in the total cost of $O(m^2 + m \cdot |\mathcal{C}(G)|)$ for chordless cycles.

We observed in Section 3.1 that any st-path will suffice as a starter, as they are our certificates of choice. This makes a difference for the above reduction, since using dynamic graph connectivity algorithms [8], it is possible to maintain a spanning tree in $O(\text{polylog}\, n)$ time per edge deletion, perform connectivity queries in $O(\text{polylog}\, n)$, and more importantly obtain an st-path in $\tilde{O}(n)$. It is worth noting that it is not known how to obtain a chordless st-path faster than $O(m)$. Hence, we first build the dynamic connectivity structure as preprocessing step. Then, for each edge (s,t), in the same order as Uno's reduction, we list the chordless st-paths. Before calling our path listing algorithm for edge (s,t), we test if s and t are connected (Fact 1): if so, we call our path listing algorithm, paying $\tilde{O}(n)$ to find one initial st-path; otherwise, we skip the edge (s,t) and take the next in order. As a result, the total initialization cost is $\tilde{O}(m + kn)$ for all edges instead of $O(m^2)$, where k is the number of edges for which we find one initial st-path. Note that $k \leq |\mathcal{C}(G)|$ as each of them surely gives rise to a chordless st-path whence to a distinct chordless cycle. We obtain in this way an $\tilde{O}(m + n \cdot |\mathcal{C}(G)|)$ time algorithm to list chordless cycles, if we can list st-paths in amortized $\tilde{O}(n)$ time each.

3.3 Difficulty of Cleaning *st*-paths

Given any st-path, as stated in Fact 1 we can *clean* it to obtain a chordless st-path in a greedy fashion: start from $u = s$ and iteratively take a neighbour of u that is closest to t along the path. The process stops when $u = t$. The vertices taken in this way form a chordless path. The problem is that the cost of such a greedy traversal of the path is upper bounded by the sum of the degrees of the vertices along it. Unfortunately, this sum could be $\Theta(m)$ in the worst case.

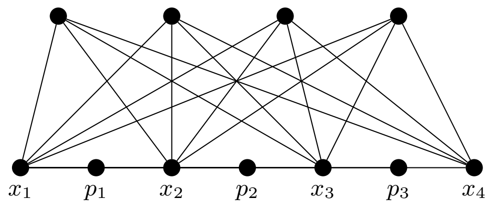


Fig. 1. Sum of degrees on chordless path $x_1, p_1, x_2, p_2, \ldots, x_{r-1}, p_{r-1}, x_r$ is $\Theta(m)$

Even worse, this is still true when the initial path is already chordless, as shown in Fig. 1. Consider the complete bipartite clique $K_{r,r} = (V_1 \cup V_2, E_{12})$, where $V_1 = \{x_1, x_2, \ldots, x_r\}$. Build a new graph $G = (V, E)$ where the vertex set is $V = V_1 \cup V_2 \cup \{p_1, \ldots, p_{r-1}\}$ for some new vertices $p_1, \ldots, p_{r-1}$, and the edge set is $E = E_{12} \cup \{(x_1, p_1), (p_1, x_2), (x_2, p_2), \ldots, (x_{r-1}, p_{r-1}), (p_{r-1}, x_r)\}$. Now, the path $x_1, p_1, x_2, p_2, \ldots, x_{r-1}, p_{r-1}, x_r$ is chordless but each edge is incident to at least one vertex in that path, so the sum of the degrees is $m = |E| = \Theta(r^2) = \Theta(|V|^2) = \Theta(n^2)$.

What we would like to do: recursively extend a given chordless path π_{su} into a chordless st-path, while maintaining as a certificate an st-path. The recursive extension can be seen as an implicit cleaning of our st-path certificate. Consider a vertex u along a given st-path (our certificate), where initially $u = s$. Our certificate guarantees that there is at least one chordless st-path going through a neighbour of u, say a. However, exploring all of u's neighbours would cost too much so we need to proceed more carefully: consider any neighbour $b \neq a$, the following two situations may occur. *(1)* a and b are both good, meaning that (u, a) and (u, b) are on two distinct chordless st-paths. In this case, the chordless st-paths traversing (u, a) cannot go through b too, as otherwise it would not be chordless (see Remark 1 below), so b should be removed. *(2)* b is not on any chordless st-path, so it is either disconnected from t or every st-path going through b passes through a. In this case, as it will be clear later, we need neither to explore nor to remove b.

In other words, we can treat the neighbours of u as described above, and they will not interfere when cleaning the st-path in the next recursive calls since their are either removed (as in case 1) or implicitly cut out (as in case 2). We make this statement more precise below.

3.4 Reduced Degree Property

We introduce a notion of reduced degree with a stronger property in mind. Consider a chordless st-path $\pi_{st} = v_0 v_1 \ldots v_\ell$ in the graph G, for some integer $\ell > 1$, where $v_0 = s$ and $v_\ell = t$. For a vertex v_i, a neighbour $v \in N(v_i)$ is *good* if there exists a chordless st-path in G with prefix $v_0 v_1 \ldots v_i v$ (*i.e.* it extends $v_0 v_1 \ldots v_i$ by adding the edge (v_i, v) as illustrated in Fig. 2). We denote by $N^{good}(v_i) \subseteq N(v_i)$ the set of *good* neighbours of v_i, noting that $v_{i+1} \in N^{good}(v_i)$. For each v_i, its *reduced degree* d_i is given by the number of non-good neighbours, namely, $d_i = |(N(v_i) \setminus \bigcup_{j \leq i} N^{good}(v_j)) \cup \{v_{i+1}\}|$.

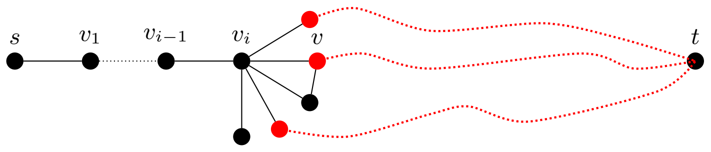


Fig. 2. Good neighbours (in red) of vertex v_i in G_i

The rationale is that exploring the good neighbours of v_i will list further chordless paths while examining its neighbours that are not good is a waste of computation. The reduced degree of v_i is actually an upper bound on the number of not-good vertices examined when exploring v_i to produce the chordless st-path π_{st} and gives an upper bound on the waste. Lemma 1 below shows that while examining the neighbours of the vertices along a chordless path still takes $O(m)$ time, only $O(n)$ neighbours are a waste while the remaining ones lead to further chordless paths (which is a good argument for amortization).

Lemma 1. *For a chordless path π_{st}, we have $\sum_{v_i \in \pi_{st}} d_i \leq 2n$, where d_i is the reduced degree of $v_i \in \pi_{st}$.*

Proof. We will show that each vertex x of G is a non-good neighbour of at most two vertices in π_{st}. To this purpose, we prove that if x is a non-good neighbour of both v_i and v_j then $|i - j| \leq 1$. We choose such three vertices v_i, v_j and x where the difference $j - i$ is the largest possible and assume by contradiction that $i < j - 1$. Thus v_i and v_j are not adjacent in π_{st}, whence (v_i, v_j) is not an edge of G since π_{st} is chordless. Also, being non-good, $x \notin \pi_{st}$. Consider the st-path $\pi^* = v_0 \ldots v_i x v_j \ldots v_l$. Clearly, π^* contains no repeated vertices and we will prove that π^* is a chordless st-path, contradicting the fact that x is not a good neighbour of v_i. The fact that π^* is chordless follows from the fact that there is no $v_k \in \pi^*$, $k \neq i$ and $k \neq j$, such that (v_k, x) is an edge of G, otherwise $j - k$ or $k - i$ would be strictly larger than $j - i$, contradicting our choice of v_i, v_j and x. $\square$

3.5 Cleanup of Current Vertex

Suppose we are extending the chordless path π_{su}, while cleaning the st-path certificate. We identify a good vertex $v \in N(u)$, which closest to t along the st-path. Ideally, we would clean the vertex u by throwing away all its other neighbours but this could cost $\Omega(m)$ per chordless path as illustrated in Fig. 1 and discussed in Section 3.3. We thus perform a partial cleaning, called *cleanup*, which consists in identifying and removing, among all neighbours of u (*i.e.* $|N(u)|$ elements) only its set $N^{good}(u)$ of good ones.

For a given u in a chordless st-path $\pi_{st} = v_0 \ldots v_i u \ldots v_l$, we let emerge the good neighbours in $N^{good}(u)$ one by one as follows. Consider the graph G' where the vertices $v_0 \ldots v_i$ and its good neighbours were removed. If u and t are not connected, then there cannot be further chordless paths from u and so there cannot be further good neighbours. Otherwise, if u and t are connected, we take any path from u to t, and select its neighbour v that appears along the path and is closest to t, as illustrated in Fig. 3. After that, we remove v and its incident edges, and iterate what described above until u is disconnected from t. The vertices v thus selected form the set $N^{good}(u)$ of good neighbours.

Lemma 2. *For a chordless path π_{st}, the cleanup of vertex $u \in \pi_{st}$ correctly produces the set $N^{good}(u)$ of its good neighbours.*

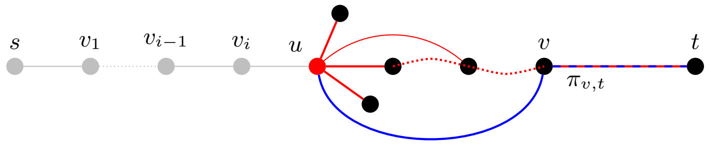


Fig. 3. Cleanup of the neighbours of vertex u

4 Listing Algorithm

We blend the key ideas discussed in Section 3 to get Algorithm 1, which has four parameters as input and lists all the chordless st-paths: the first parameter is the chordless path π_{su} partially built from s to the current vertex u (initially, $u = s$), which is the second parameter; the third parameter is a ut-path π_{ut} that plays the role of certificate by Fact 1; the fourth parameter is the reduced graph G, which changes with the recursive calls.

Algorithm 1. `list_induced_paths`$_{s,t}(\pi_{su}, u, \pi_{ut}, G)$

```
1  if $u = t$ then
2      output($\pi_{su}$)
3  else
4      $S := \emptyset$
5      while true do
6          $v :=$ the vertex in $\pi_{ut} \cap N(u)$ that is closest to $t$ in $\pi_{ut}$
7          $\pi_{vt} :=$ the subpath of $\pi_{ut}$ from $v$ to $t$
8          $S := S \cup \{(v, \pi_{vt})\}$
9          remove $v$ and its incident edges from $G$
10         if u and t are not connected then break
11         $\pi_{ut} :=$ any path from $u$ to $t$
12     end
13     foreach $(v, \pi_{vt}) \in S$ do
14         adds back $v$ and its incident edges to $G$
15         list_induced_paths$_{s,t}(\pi_{su} \cdot (u, v), v, \pi_{vt}, G)$
16         remove $v$ and its incident edges from $G$
17     end
18 end
```

The algorithm outputs a chordless st-path if $u = t$ (line 2). Otherwise, it performs a cleanup of u (the loop at lines 5–12). After that, it explores only the good neighbours recursively as they will surely lead to further chordless paths (the other loop at lines 13–17). Observe that S stores the good neighbours v of u and a vt-path for each of them: when performing the recursive call at line 15, only one of the vertices in S appears in the reduced graph G passed as a parameter to the recursive call (see lines 14 and 16 that guarantee this, and Remark 1 below). Hence, the recursive call now has as parameters the chordless sv-path $\pi_{su} \cdot (u, v)$ ending in v, and a vt-path that guarantees that a chordless st-path exists and has $\pi_{su} \cdot (u, v)$ as a prefix. This recursive call lists all the chordless st-paths that share this prefix.

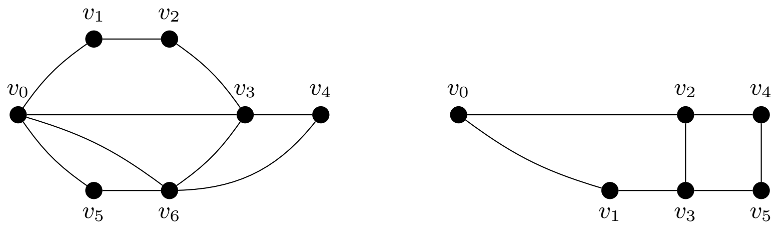


Fig. 4. Two example graphs where $s = v_0$ and $t = v_4$

For example, let us run Algorithm 1 on the input graph shown on the left of Fig. 4, with $u = s = v_0$ and the initial path $\pi_{ut} = v_0v_1v_2v_3v_4$. It computes the pairs (v, π_{vt}) in S as follows. First, (v_3, v_3v_4) is added to S as v_3 is a good neighbour for v_0 (the neighbour closest to t in the path), and the edges incident to v_3 are removed. After this removal, $s = v_0$ is still connected to $t = v_4$ through the path $v_0v_5v_6v_4$, which becomes the input for the next iteration of the while loop. Next, (v_6, v_6v_4) is added to S as v_6 is another good neighbour, and the edges incident to v_6 are removed disconnecting s from t, so the while loop ends. The recursive calls in the foreach loop give the two chordless paths $v_0v_3v_4$ and $v_0v_6v_4$ contained in the graph.

Remark 1. It is important to run the recursive calls with all good neighbours in S removed except one. If we left two or more good neighbours in the recursive call of line 15, they could interfere with each other and we might not obtain the chordless paths correctly. A very simple example is given in the graph shown on the right of Fig. 4. Consider for instance the case where Algorithm 1 would be given as input the path $v_0v_1v_3v_2v_4$. The pair (v_2, v_2v_4) is added to S and v_2 removed. After that, the path $v_0v_1v_3v_5v_4$ is found and the pair $(v_1, v_1v_3v_5v_4)$ is added to S and v_1 removed. Since v_0 and v_4 become disconnected, S contains all the good neighbours of v_0. Algorithm 1 executes the recursive calls with S. Suppose that we keep both good neighbours v_1 and v_2 in G during these calls, in particular for the call with the pair $(v_1, v_1v_3v_5v_4)$ from S. This call will extend in a nested call the chordless path to $\pi_{su} = v_0v_1v_3$ for $u = v_3$, and will claim that the good neighbours of v_3 are v_2 and v_5, which is incorrect since $v_0v_1v_3v_2$ is not chordless. This situation does not arise if v_2 is kept deleted in G when the recursive call on v_1 is performed as done in Algorithm 1.

The correctness of Algorithm 1 follows mostly from Lemma 2. Recall that, it guarantees that for a given path prefix π_{su} the set S contains the good neighbours of u, *i.e.* the neighbours of u that belong to at least one chordless st-path extending π_{su}. Clearly, we only have to recursively call the algorithm for these neighbours, the others certainly lead to no solution. This implies that Algorithm 1 tries all the possibilities to extend π_{su}, so all chordless st-paths are output. Moreover, since each good neighbour of u leads to a different extension, we have that no st-path is output more than once.

Certainly only st-paths are output by Algorithm 1, but at this point we have no guarantees that the paths are indeed chordless. In fact, after building S, the algorithm proceeds to recursively extend the prefix $\pi_{su} \cdot (u, v)$ for each $v \in S$ in the graph $G' = G \setminus (S \setminus \{v\})$. However, since u was included in current path none of its neighbours can be used later in the recursion. The algorithm removes the good neighbours of u from G, but the other neighbours, $N_G(u) \setminus S$, are still present in G'. They could thus be used to extend the path later in the recursion, resulting in a non-chordless st-path. Lemma 3 shows that this cannot happen.

Lemma 3. *The st-paths output by Algorithm 1 are chordless.*

The previous lemma leads to the following theorem.

Theorem 1. *The algorithm correctly outputs all chordless st-paths of G.*

Theorem 2. *The algorithm takes $O(m + |\mathcal{P}_{st}(G)|(t_p + nt_q + nt_u))$ time, where t_p is the cost of choosing any path from any two given vertices, t_q is the cost of checking if any given two vertices are connected or not, and t_u is the cost of removing/adding back any given edge.*

Proof. See Section 5.

There are several dynamic data structures in the literature [8] that maintain a spanning forest for a dynamic graph, supporting insertions and deletions of edges in polylogarithmic time. Consequently, $t_p = O(n \,\text{polylog}\,(n))$, $t_q = O(\text{polylog}\,(n))$, and $t_u = O(\text{polylog}\,(n))$, thus giving the following bound.

Corollary 1. *The algorithm takes $\tilde{O}(m + |\mathcal{P}_{st}(G)| \cdot n)$ time to report all the chordless st-paths.*

5 Amortized Analysis

Before starting our analysis, we observe some simple properties of the recursion tree generated by Algorithm 1.

Fact 2. *The recursion tree R of Algorithm 1 has the following properties:*

1. *There is a one-to-one correspondence between paths in $\mathcal{P}_{st}(G)$ and leaves in the recursion tree.*
2. *There is a one-to-one correspondence between proper prefixes of paths in $\mathcal{P}_{st}(G)$ and internal nodes in the recursion tree.*

3. The number of branching nodes is $|\mathcal{P}_{st}(G)| - 1$.
4. The length of a root-to-leaf path is equal to the length of the chordless st-path corresponding to the leaf. In particular, the height of the tree is $\leq n$.

Fact 2 suggests us to follow the following overall strategy.

1. We analyze the cost of each type (leaf, unary and branching) of node separately.
2. We consider all branching nodes together, and show that their amortized cost is $O(t_p + t_q + nt_u + n) = \tilde{O}(n)$ per solution.
3. We consider all unary nodes together, and show that their amortized cost is $O(|\pi_{st}|t_q + nt_u) = \tilde{O}(n)$ per solution.
4. We deduce that the cost of each solution is $O(t_p + nt_q + nt_u) = \tilde{O}(n)$.

Where the cost of a node is the time spent by the corresponding call without including the time spent by its nested recursive calls.

Lemma 4. *The cost of a leaf is $O(|\pi_{st}|)$.*

Let us now analyze the cost of the unary nodes. Let $r = \langle \pi_{su}, u, \pi_{ut}, G \rangle$ be a unary node. The vertex $v \in N(u)$ is the only neighbour of u that can extend the prefix π_{su} into a chordless st-path. Thus, removing v from G disconnects u from t, and the algorithm performs a single iteration of the loop in line 5, not executing line 11. In this case, the algorithm performs the following operations: (i) one connectivity query (line 10), (ii) $|N(v)|$ edge update operations on G (lines 9, 14 and 16), and (iii) a scan in the intersection of $N(u)$ and π_{ut} to find v (line 6). The cost of (i) and (ii) is $O(t_q + |N(v)|t_u)$.

A naive implementation of (iii) takes $O(|N(u)| + |\pi_{ut}|)$ time, which is too large to fit in our amortization strategy. In order to reduce this cost to $O(|N(u)|)$ we therefore maintain, as an extra invariant, for each vertex in the current graph its distance to t along the path π_{ut}. In this way, we can find v simply scanning $N(u)$. Thus, assuming the distance information is correctly maintained, we complete the proof of Lemma 5.

Lemma 5. *The cost of a unary node is $O(t_q + |N(v)|t_u + |N(u)|)$, where (u, v) is the edge added to the chordless path.*

It is not hard to maintain the distance information for $\langle \pi_{su} \cdot (u, v), v, \pi_{vt}, G' \rangle$, the only child of the unary node $\langle \pi_{su}, u, \pi_{ut}, G \rangle$. As the path π_{vt} is a suffix of π_{ut}, the distance of the vertices in π_{ut} does not change. On the other hand, the only vertices that the distances can change are the ones in π_{vt} but not in π_{ut}. These vertices can be identified when scanning $N(v)$ in the child node $\langle \pi_{su} \cdot (u, v), v, \pi_{vt}, G' \rangle$, since their distance is strictly larger than $|\pi_{vt}|$. It remains to show that the distance information can be maintained in the branching nodes.

Lemma 6. *The cost of a branching node $r \in R$ is $O(\beta(r)(t_p + t_q + nt_u))$, where $\beta(r)$ is the number of children of r.*

Let us now show that we can maintain the distance information in branching nodes in the same time bound of Lemma 6. This follows from the fact that in each iteration of the loop (line 5) we are already paying $O(|\pi_{ut}|)$, *i.e.* a full traversal of the path π_{ut}. Before each recursive call in line 15 we can traverse the path π_{ut} adding for each vertex the distance information, *i.e.* their position in the path.

At this point we have bounds for the cost of each node in the recursion tree. However, by directly applying them we cannot achieve our goal of $\tilde{O}(n)$ time per solution. For instance, consider the particular case where all internal nodes of the recursion tree are branching. The cost of each internal node is $O(\beta(r)(t_p + t_q + nt_u)) = \tilde{O}(n^2)$, since $\beta(r) = \Omega(n)$ in the worst case. Then, from item 3 of Fact 2, the number of branching nodes is $|\mathcal{P}_{st}(G)| - 1$. The total cost for the tree is thus $\tilde{O}(|\mathcal{P}_{st}(G)| \cdot n^2)$ or $\tilde{O}(n^2)$ per solution.

In order to get a tighter bound for the total cost of the branching nodes, we use the following amortization strategy. Let $r \in R$ be a branching node. We divide the cost $O(\beta(r)(t_p + t_q + nt_u))$ among the closest descendents that are branching nodes or leaves (no unary nodes), each being charged $O(t_p + t_q + nt_u)$. This can always be done since r has $\beta(r)$ children and the subtree of each child contains at least one leaf, *i.e.* the node r has at least $\beta(r)$ non-unary descendants. In this way, the original cost of node r is completely charged to its non-unary descendants, and the only cost that remains associated to r is the one received from its ancestors. Finally, each branching node can only be charged once, by its lowest non-unary ancestor. Each branching node and each leaf is therefore charged with $O(t_p + t_q + nt_u)$. Thus, the total cost of the branching nodes is $O(|\mathcal{P}_{st}(G)|(t_p + t_q + nt_u))$, completing the proof of Lemma 7.

Lemma 7. $\sum_{r:\text{branching}} T(r) = O(|\mathcal{P}_{st}(G)|(t_p + t_q + nt_u))$.

Let us now bound the total cost of the unary nodes. Similarly to the branching nodes case, a straightforward use of the bound given by Lemma 5 leads to an $\tilde{O}(n^2)$ cost per solution, since in the worst case the recursion tree can have $O(n)$ unary nodes for each leaf. The key idea to obtain a better amortized cost is to consider the bound on the reduced degrees given by Lemma 1.

We first observe that each unary node is contained in some root-to-leaf path $\Pi(l)$, where l is a leaf of the recursion tree. Thus,

$$\sum_{r:\text{unary}} T(r) \leq \sum_{l:\text{leaf}} \sum_{r \in \Pi(l)} T(r). \tag{1}$$

Fact 2 implies that there is a one-to-one correspondence between the prefixes of paths in $\mathcal{P}_{st}(G)$ and nodes in the recursion tree. That is, each leaf corresponds to a solution, and the root-to-leaf path $\Pi(l)$ corresponds to the chordless st-path associated to the leaf l. Moreover, the $O(t_q + |N(v)|t_u + |N(u)|)$ cost of an unary node can be amortized to $O(t_q + |N(v)|t_u)$, since we can always charge $|N(u)| = O(n)$ to its single child. We can thus rewrite the double sum as

$$\sum_{l:\text{leaf}} \sum_{r \in \Pi(l)} T(r) = \sum_{\pi \in \mathcal{P}_{st}(G)} \sum_{v_i \in \pi} (t_q + |N(v_i)|t_u). \tag{2}$$

For each chordless st-path π in the internal sum of Eq. 2, we have that the degrees are actually the reduced degrees of Section 3.4, since the good neighbours (*i.e.* the set S in Algorithm 1) are always removed. Using Lemma 1 we can thus bound the sum of the degrees by $2n$. Therefore,

$$\sum_{r:\text{unary}} T(r) \leq \sum_{\pi \in \mathcal{P}_{st}(G)} (|\pi| t_q + 2n t_u), \tag{3}$$

completing the proof of Lemma 8.

Lemma 8. *$\sum_{r:\text{unary}} T(r) = O(\sum_{\pi \in \mathcal{P}_{st}(G)} |\pi| t_q + n t_u)$.*

As a corollary of Lemmas 8 and 7, we obtain Theorem 2.

References

1. Birmelé, E., Ferreira, R.A., Grossi, R., Marino, A., Pisanti, N., Rizzi, R., Sacomoto, G.: Optimal listing of cycles and st-paths in undirected graphs. In: SODA 2013, pp. 1884–1896. ACM/SIAM (2013)
2. Chen, Y., Flum, J.: On parameterized path and chordless path problems. In: IEEE Conference on Computational Complexity, pp. 250–263 (2007)
3. Chudnovsky, M., Robertson, N., Seymour, P., Thomas, R.: The strong perfect graph theorem. Annals of Mathematics 164, 51–229 (2006)
4. Conforti, M., Cornuéjols, G., Kapoor, A., Vuskovic, K.: Recognizing balanced 0, +/- matrices. In: SODA 1994, pp. 103–111. ACM/SIAM (1994)
5. Conforti, M., Cornuéjols, G., Kapoor, A., Vuskovic, K.: Finding an even hole in a graph. In: FOCS 1997, pp. 480–485. IEEE Computer Society (1997)
6. Conforti, M., Rao, M.R.: Structural properties and decomposition of linear balanced matrices. Math. Program. 55, 129–168 (1992)
7. Haas, R., Hoffmann, M.: Chordless paths through three vertices. Theoretical Computer Science 351(3), 360–371 (2006)
8. Kapron, B.M., King, V., Mountjoy, B.: Dynamic graph connectivity in polylogarithmic worst case time. In: SODA, pp. 1131–1142 (2013)
9. Kawarabayashi, K.-I., Kobayashi, Y.: The induced disjoint paths problem. In: Lodi, A., Panconesi, A., Rinaldi, G. (eds.) IPCO 2008. LNCS, vol. 5035, pp. 47–61. Springer, Heidelberg (2008)
10. Read, C., Tarjan, R.E.: Bounds on backtrack algorithms for listing cycles, paths, and spanning trees. Networks 5(3), 237–252 (1975)
11. Seinsche, D.: On a property of the class of n-colorable graphs. Journal of Combinatorial Theory, Series B 16(2), 191–193 (1974)
12. Sokhn, N., Baltensperger, R., Bersier, L.-F., Hennebert, J., Ultes-Nitsche, U.: Identification of chordless cycles in ecological networks. In: Glass, K., Colbaugh, R., Ormerod, P., Tsao, J. (eds.) Complex 2012. LNICST, vol. 126, pp. 316–324. Springer, Heidelberg (2013)
13. Maciej, M.: Syslo. An efficient cycle vector space algorithm for listing all cycles of a planar graph. SIAM J. Comput. 10(4), 797–808 (1981)
14. Uno, T.: Algorithms for enumerating all perfect, maximum and maximal matchings in bipartite graphs. In: Leong, H.-V., Jain, S., Imai, H. (eds.) ISAAC 1997. LNCS, vol. 1350, pp. 92–101. Springer, Heidelberg (1997)
15. Uno, T.: An output linear time algorithm for enumerating chordless cycles. In: 92nd SIGAL of Information Processing Society Japan, pp. 47–53 (2003) (in Japanese)
16. Wild, M.: Generating all cycles, chordless cycles, and hamiltonian cycles with the principle of exclusion. J. of Discrete Algorithms 6(1), 93–102 (2008)

LP Approaches to Improved Approximation for Clique Transversal in Perfect Graphs [⋆]

Samuel Fiorini[1], R. Krithika[2], N.S. Narayanaswamy[2], and Venkatesh Raman[3]

[1] Département de Mathématique
Université libre de Bruxelles, Brussels, Belgium
`sfiorini@ulb.ac.be`

[2] Department of Computer Science and Engineering
Indian Institute of Technology Madras, Chennai, India
`{krithika,swamy}@cse.iitm.ac.in`

[3] The Institute of Mathematical Sciences, Chennai, India
`vraman@imsc.res.in`

Abstract. Given an undirected simple graph G, a subset T of vertices is an r-clique transversal if it has at least one vertex from every r-clique in G. I.e. T is an r-clique transversal if $G - S$ is K_r-free. r-clique transversals generalize vertex covers as a vertex cover is a set of vertices whose deletion results in a graph that is K_2-free. Perfect graphs are a well-studied class of graphs on which a minimum vertex cover can be obtained in polynomial time. However, the problem of finding a minimum r-clique transversal is NP-hard even for $r = 3$. As any induced odd length cycle in a perfect graph is a triangle, a triangle-free perfect graph is bipartite. I.e. in perfect graphs, a 3-clique transversal is an odd cycle transversal. In this work, we describe an $(\frac{r+1}{2})$-approximation algorithm for r-clique transversal on weighted perfect graphs improving on the straightforward r-approximation algorithm. We then show that 3-Clique Transversal is APX-hard on perfect graphs and it is NP-hard to approximate it within any constant factor better than $\frac{4}{3}$ assuming the unique games conjecture. We also show intractability results in the parameterized complexity framework.

1 Introduction

Given a graph G and a weighting $w : V(G) \mapsto \mathbb{Q}_+$ of the vertices of G, r-Clique Transversal is the problem of determining a minimum weight set of vertices that has at least one vertex from every r-clique in G.

r-Clique Transversal
Instance: A graph G and $w : V(G) \mapsto \mathbb{Q}_+$
Output: A minimum weight set $T \subset V(G)$ such that $G - T$ is K_r-free

[⋆] Supported by the Indo-German Max Planck Centre for Computer Science Programme in the area of *Algebraic and Parameterized Complexity*.

Note that r is fixed and not a part of the input in the problem definition. For each $r \geq 3$, it is NP-hard to approximate r-CLIQUE TRANSVERSAL within a factor better than r under the unique games conjecture [10]. Further, no approximation factor better than $r - 1$ is possible unless $NP \subseteq BPP$ [10]. Therefore, we restrict our study of r-CLIQUE TRANSVERSAL to perfect graphs which is one of the largest classes of graphs on which 2-CLIQUE TRANSVERSAL (VERTEX COVER) is polynomial-time solvable [9].

r-CLIQUE TRANSVERSAL has been recently studied on perfect graphs in the parameterized complexity framework [16]. In perfect graphs, a 3-clique transversal is an odd cycle transversal and the NP-hardness of the optimization problem is known from [3]. The study of ODD CYCLE TRANSVERSAL (OCT) has given rise to interesting techniques. The first fixed-parameter tractable algorithm for OCT [21] introduced an algorithmic paradigm called 'iterative compression' that was later used to prove several parameterized problems fixed-parameter tractable. The recent improvement to this bound gave interesting reductions and use of linear programming techniques in parameterized complexity [17]. Kernelization for OCT resulted in non trivial applications of matroid techniques [14].

In this work, we study the approximability of r-CLIQUE TRANSVERSAL on perfect graphs.

1.1 Results and Techniques Used

We study r-CLIQUE TRANSVERSAL on perfect graphs in two contexts.

1. We define a linear programming formulation that captures constraints for cliques of sizes r and $r + 1$ and use it to describe an $(\frac{r+1}{2})$-approximation algorithm. We employ the primal-dual method and an $\frac{r}{2}$-approximation algorithm for VERTEX COVER in r-uniform r-partite hypergraphs as a subroutine. As a consequence, we obtain a 2-approximation algorithm for OCT on perfect graphs.
2. Along the lines of finding a minimum vertex cover in perfect graph, we formulate a linear program for OCT using clique constraints. We show that an optimum solution to this linear program can be obtained in polynomial time for perfect graphs. Using this solution, we describe a rounding procedure that leads to a 2-approximation.

We then show intractability results in the approximation and parameterized complexity frameworks. We show that OCT on perfect graphs is APX-hard and hence has no polynomial-time approximation scheme unless $P = NP$. In the parameterized setting where the interest is in determining whether the graph has an odd cycle transversal of size at most k, we show that $O^*(2^{o(k)})$[1] algorithms or $O(k^{2-\epsilon})$ kernels are unlikely under standard complexity theoretic assumptions. From a recent result, an $O^*(2^k)$ algorithm is known for this problem [16]. The hardness results comprise of the following.

[1] O^* notation ignores polynomial terms.

- If there is a c-approximation algorithm for OCT on perfect graphs with $c < 1.12$, then $\mathsf{P} = \mathsf{NP}$.
- Assuming the unique games conjecture, if there is a c-approximation algorithm for OCT on perfect graphs with $c < \frac{4}{3}$, then $\mathsf{P} = \mathsf{NP}$.
- No $O^*(2^{o(k)})$ algorithm is likely for the problem of determining if a given perfect graph has an odd cycle transversal of at most k vertices under the Exponential Time Hypothesis.
- No $O(k^{2-\epsilon})$ edges kernel is possible for any $\epsilon > 0$ unless the polynomial hierarchy collapses to the third level.

Apart from giving an improved bound and showing hardness for r-CLIQUE TRANSVERSAL on perfect graphs, this work makes three important contributions. First, the approximation algorithms that we describe show the applicability of VERTEX COVER algorithms in solving its generalizations (OCT and r-CLIQUE TRANSVERSAL). They also exemplify two different techniques for designing 2-approximation algorithms for OCT. Second, our study of a clique-constrained odd cycle transversal polytope yields an approximation algorithm for OCT that uses the polyhedral characterization of perfect graphs based on the stable set polytope description. Third, our approach for showing hardness results demonstrates a way of turning an NP-hardness proof into a proof for hardness in the approximation and parameterized settings using linear kernelization.

Road Map: In Section 2, we define the necessary preliminaries on linear programming and perfect graphs. In Section 3, we describe a polynomial-time $(\frac{r+1}{2})$-approximation algorithm for r-CLIQUE TRANSVERSAL on perfect graphs using the primal-dual method. In Section 4, we describe a 2-approximation algorithm for OCT using the Lovász theta function. In Section 5, we show hardness results in the approximability and parameterized complexity frameworks.

2 Preliminaries

Let G denote a graph with vertex set $V(G) = \{1, \cdots, n\}$ and weighting $w : V(G) \mapsto \mathbb{Q}_+$. Here, w denotes the vector in $\mathbb{Q}_+^n$ associated with $V(G)$ where $w(i)$ is the weight of the vertex i. Unweighted graphs are treated as weighted graphs with unit weight on each of its vertices. Let $\mathbf{1}$ denote a vector which has all components equal to 1. The weight of a subset S of vertices is defined as $\sum_{i \in S} w(i)$. Following the notation used in [9], for graph G, we denote the chromatic number by $\chi(G)$, the weighted clique number by $\omega(G, w)$ and the weighted independence number by $\alpha(G, w)$. In this usage, $\omega(G)$ is $\omega(G, \mathbf{1})$. Extending this notation, $\text{oct}(G, w)$ and $r\text{-ct}(G, w)$ denote the minimum weights of an odd cycle transversal and an r-clique transversal, respectively, of G. Also, $\text{oct}(G)$ is $\text{oct}(G, \mathbf{1})$ and $r\text{-ct}(G)$ is $r\text{-ct}(G, \mathbf{1})$. G is said to be r-colorable (or r-partite) if $\chi(G) \leq r$. Graph theoretic notation and definitions not given explicitly here can be found in [8].

A graph G in which each induced subgraph H satisfies $\chi(H) = \omega(H)$ is called *perfect*. Perfect graphs were introduced by Claude Berge in the early 1960s [2].

The strong perfect graph theorem establishes that a graph is perfect if and only if neither the graph nor its complement contains an induced cycle of odd length at least 5 as an induced subgraph [5]. For more details on the structural and algorithmic aspects of perfect graphs, we refer the reader to the classical books by Golumbic [8] and Grötschel, Lovász and Schrijver [9].

Linear programming (LP) notation and definitions are as defined in [9]. For a vector $x = (x(1), x(2), \cdots, x(n))$, the vector denoted by $\mathbf{1} - x$ is defined as $(1 - x(1), 1 - x(2), \cdots, 1 - x(n))$. For a set $S \subseteq V(G)$, the incidence vector of S is the vector $\chi^S \in \mathbb{R}^n$ defined by $\chi^S(i) = 1$ if $i \in S$ and $\chi^S(i) = 0$ if $i \notin S$. The *convex hull* of a set $S \subseteq \mathbb{R}^n$ is denoted by $conv\ S$. A *separation oracle* for a polytope $P = \{x \in \mathbb{R}^n : Ax \leq b\}$ is a procedure which given a vector $y \in \mathbb{R}^n$, determines if y is in P, and if not, finds a violated inequality $A_i y > b_i$. The *optimization* problem for P is to find a vector $y \in P$ that maximizes/minimizes $w^T x$ on P for a given vector $w \in \mathbb{R}^n$, or assert that P is empty.

3 A Primal-Dual Approach to r-Clique Transversal

We describe a polynomial-time $(\frac{r+1}{2})$-approximation algorithm for r-CLIQUE TRANSVERSAL on perfect graphs. We employ the primal-dual method and also use the $\frac{r}{2}$-approximation for VERTEX COVER in r-uniform r-partite hypergraphs due to Lovász [18] as an oracle. We first mention the linear programming formulation and the approximation results for VERTEX COVER in r-uniform r-partite hypergraphs [18].

3.1 VERTEX COVER in r-uniform r-partite Hypergraphs

A hypergraph H is r-uniform if each edge in H is of size r and it is r-partite if there exists a partition of $V(H)$ into r subsets $V_1, \cdots, V_r$ such that for every edge e in $E(H)$, $|e \cap V_i| \leq 1$ for each $i \in \{1, \cdots, r\}$. A vertex cover S of H is a subset of vertices such that $S \cap e \neq \emptyset$ for every $e \in E(H)$. The standard linear programming relaxation for VERTEX COVER in hypergraphs is the following.

LP Relaxation of VERTEX COVER: LP-Vc(H)

$$\min \sum_{i \in V(H)} w(i)x(i) \text{ subject to } \sum_{i \in e} x(i) \geq 1 \text{ for each } e \in E(H) \text{ and } x \geq 0$$

Dual of LP-Vc(H)

$$\max \sum_{e \in E(H)} y(e) \text{ subject to } \sum_{e \in E(H): i \in e} y(e) \leq w(i) \text{ for each } i \in V(H) \text{ and } y \geq 0$$

In [18], Lovász proved the following.

Theorem 1. *Given an r-uniform r-partite hypergraph H with vertex weighting w, there exists a polynomial-time algorithm that returns an integer solution x^* of LP-Vc(H) and a solution y^* of its dual such that* $\sum_{i \in V(H)} w(i)x^*(i) \leq \frac{r}{2} \sum_{e \in E(H)} y^*(e)$.

3.2 r-Clique Transversal on Perfect Graphs

Let G be a perfect graph with positive rational vertex weighting w. Let $\mathcal{K}(G,r)$ denote the set of r-cliques in G. Let $\mathcal{Q}(G)$ denote $\mathcal{K}(G,r) \cup \mathcal{K}(G,r+1)$. We now describe a primal-dual approach for approximating r-Clique Transversal.

LP Formulation of r-Clique Transversal: LP-CT(G,r)

$$\begin{aligned} \min \sum_{i \in V(G)} w(i)x(i) & \\ \text{subject to } \sum_{i \in K} x(i) \geq 2 & \quad \text{for each } K \in \mathcal{K}(G,r+1) \\ \sum_{i \in K} x(i) \geq 1 & \quad \text{for each } K \in \mathcal{K}(G,r) \\ x(i) \geq 0 & \quad \text{for each } i \in \{1,\cdots,|V(G)|\} \end{aligned}$$

Dual of LP-CT(G,r): LP-CP(G,r)

$$\begin{aligned} \max \sum_{K \in \mathcal{K}(G,r)} y(K) + 2 \sum_{K \in \mathcal{K}(G,r+1)} y(K) & \\ \text{subject to } \sum_{Q \in \mathcal{Q}(G): i \in Q} y(Q) \leq w(i) & \quad \text{for each } i \in V(G) \\ y(Q) \geq 0 & \quad \text{for each } Q \in \mathcal{Q}(G) \end{aligned}$$

The idea is to construct a greedy weighted packing of $(r+1)$-cliques in polynomial time and add all vertices corresponding to tight constraints in the dual into the solution. Then, we use Theorem 1 on the residual graph with residual weights to obtain an r-clique transversal.

```
Algorithm Approx-r-CT($G, w$)
(1) Initialize primal solution $x^*$ to 0, dual solution $y^*$ to 0 and set $T$ to $\emptyset$.
(2) While $x^*$ violates the LP-CT(G, r) constraint for an $(r+1)$-clique $K$,
    2.1. Increase $y^*(K)$ to the maximum value possible ensuring that no
         constraint in LP-CP(G, r) is violated.
    2.2. Set $x^*(i) = 1$ for every vertex $i$ with $\sum_{K \in \mathcal{K}(G,r+1): i \in K} y^*(K) = w(i)$.
(3) Add $\{i \in V(G) : x^*(i) = 1\}$ to $T$.
(4) Define residual weighting $w'$ as $w'(i) = w(i) - \sum_{K \in \mathcal{K}(G,r+1): i \in K} y^*(K)$.
(5) Define residual graph $G' = G - T$ with vertex weighting $w'$.
(6) Define the hypergraph $H$ on the vertex set $V(G')$ (with weighting $w'$)
where hyperedges correspond to r-cliques in $G'$. That is, $V(H) = V(G')$
and $E(H) = \{e_K : K \in \mathcal{K}(G', r)\}$.
(7) Obtain a primal integer solution $x'$ and a dual solution $y'$ for LP-Vc(H)
using Theorem 1.
(8) For each $i \in V(H)$, set $x^*(i) = x'(i)$.
(9) For each $e_K \in E(H)$, set $y^*(K) = y^*(K) + y'(e_K)$.
(10) Add $\{i \in V(G) : x'(i) = 1\}$ to $T$. Return $T$.
```

Theorem 2. *Given a perfect graph* G *with vertex weighting* w*, Algorithm Approx-*r*-CT(*G, w*) returns an* r*-clique transversal* T *of* G *with* $w(T) \leq (\frac{r+1}{2}) \cdot r\text{-ct}(G, w)$ *in polynomial time.*

Proof. Let T denote the solution returned by the Algorithm Approx-r-CT(G, w). Let V_1 and V_2 be the sets of vertices of G that are added to T in steps (3) and (10), respectively. From step (5), it follows that V_1 and V_2 are disjoint. As χ^T is x^*, it respects the LP-CT(G, r) constraints corresponding to each $Q \in \mathcal{Q}(G)$. Thus, T is an r-clique transversal of G. We will now bound the weight of T.

$$w(T) = \sum_{i \in V_1} w(i) + \sum_{i \in V_2} w(i) = w(V_1) + w(V_2)$$

A vertex i is added to T in step (3) only if $w(i) = \sum_{K \in \mathcal{K}(G,r+1):i \in K} y^*(K)$. Also, by the definition of w' and due to the fact that a vertex i is added to T in step (10) only if $x'(i) = 1$, we have,

$$w(T) = \sum_{i \in V_1} \sum_{K \in \mathcal{K}(G,r+1):i \in K} y^*(K) + \sum_{i \in V_2} (w'(i) + \sum_{K \in \mathcal{K}(G,r+1):i \in K} y^*(K))$$

Each clique K in the above sum is counted by at most $r + 1$ vertices. Since G' is a K_{r+1}-free perfect graph, the r-uniform hypergraph H defined in step (6) is r-partite. From Theorem 1, step (7) can be performed in polynomial time. Rearranging the terms in the sum and using $\sum_{i \in V_2} w'(i) = \sum_{i \in V(H)} w'(i)x'(i)$, it follows that,

$$w(T) \leq (r+1) \sum_{K \in \mathcal{K}(G,r+1)} y^*(K) + \sum_{i \in V(H)} w'(i)x'(i)$$

Further, from Theorem 1, $\sum_{i \in V(H)} w'(i)x'(i) \leq \frac{r}{2} \sum_{e_K \in E(H)} y'(e_K)$. Note that for each $K \in \mathcal{K}(G, r)$, $y^*(K) = y'(e_K)$.

$$w(T) \leq \frac{(r+1)}{2} \sum_{K \in \mathcal{K}(G,r+1)} 2y^*(K) + \frac{r}{2} \sum_{K \in \mathcal{K}(G,r)} y^*(K)$$

Now, y^* defined in step (9) is a feasible solution for LP-CP(G, r). Consider $i \in V(G)$ corresponding to a constraint in LP-CP(G, r).

$$\begin{aligned}\sum_{Q \in \mathcal{Q}(G):i \in Q} y^*(Q) &= \sum_{K \in \mathcal{K}(G,r+1):i \in K} y^*(K) + \sum_{K \in \mathcal{K}(G,r):i \in K} y^*(K) \\ &= (w(i) - w'(i)) + \sum_{e_K \in E(H):i \in e_K} y'(e_K) \\ &\leq (w(i) - w'(i)) + w'(i) = w(i)\end{aligned}$$

Since the value of the LP-CP(G, r) feasible solution y^* is a lower bound for the value of an optimal feasible solution to LP-CT(G, r) which in turn is a lower bound for $r\text{-ct}(G, w)$, we get the following bound on $w(T)$.

$$w(T) \leq \frac{(r+1)}{2}\Big(\sum_{K \in \mathcal{K}(G,r+1)} 2y^*(K) + \sum_{K \in \mathcal{K}(G,r)} y^*(K)\Big) \leq \frac{(r+1)}{2} r\text{-ct}(G, w)$$

□

As a 3-clique transversal is an odd cycle transversal in perfect graphs, we obtain the following result.

Corollary 1. *Given a perfect graph* G *with vertex weighting* w*, for* $r = 3$*, Algorithm Approx-*r*-CT(*G, w*) returns an odd cycle transversal* T *of* G *with* $w(T) \leq 2 \cdot \text{oct}(G, w)$ *in polynomial time.*

Next, we present another 2-approximation algorithm for Oct using a linear program based on clique constraints and the polyhedral characterization of perfect graphs using the stable set polytope description.

4 Approximating Oct Using the Lovász Theta Function

Finding a minimum vertex cover in a perfect graph G is essentially to compute the Lovász theta function $\vartheta(G)$ [9]. In this section, we show that the theta function computation can be employed to obtain a 2-approximation for Oct. We generalize a linear programming formulation given by Grötschel, Lovász and Schrijver for independent sets [9] to induced bipartite graphs. For perfect graphs, we show that a fractional optimum solution to this linear program can be obtained in polynomial time (in the number of vertices). We round this solution to obtain a 2-approximation.

4.1 Perfect Graphs and the Vertex Cover Polytope

We now rephrase and state the known polyhedral characterization of perfect graphs in terms of vertex cover polytopes. Let G be a perfect graph with positive rational vertex weighting w. The independent set polytope (or stable set polytope) $\text{STAB}(G)$ is defined as the convex hull of the incidence vectors of all independent sets of G.

$$\text{STAB}(G) = conv\, \{\chi^S \in \{0,1\}^n \mid S \subseteq V(G) \text{ is an independent set in } G\}$$

$\alpha(G, w)$ is equal to the maximum value of $\sum_{i=1}^{n} w(i)x(i)$ for $x \in \text{STAB}(G)$. The clique-constrained independent set polytope $\text{QSTAB}(G)$ is defined as,

$$\text{QSTAB}(G) = \{x \in [0,1]^n : \textstyle\sum_{i \in Q} x(i) \leq 1,\ \forall Q \text{ a clique in } G\}.$$

As a clique and an independent set can intersect in at most one vertex, it follows that $\text{STAB}(G) \subseteq \text{QSTAB}(G)$, for any graph G. Further, a graph G is perfect if and only if $\mathrm{STAB}(G) = \mathrm{QSTAB}(G)$ [9]. In this work, we focus on the vertex cover polytope $\mathrm{VC}(G)$ which is the convex hull of the incidence vectors of all vertex covers of G. Let $\mathrm{QVC}(G) = \{x \in [0,1]^n : \sum_{i \in Q} x(i) \geq |Q| - 1, \ \ \forall Q$ a clique in G$\}$. Since $\mathrm{VC}(G) = \{1 - x : x \in \mathrm{STAB}(G)\}$ and $\mathrm{QVC}(G) = \{1 - x : x \in \mathrm{QSTAB}(G)\}$, a graph G is perfect if and only if $\mathrm{VC}(G) = \mathrm{QVC}(G)$. Further, on perfect graphs, VERTEX COVER is solvable in polynomial time [9]. For a detailed exposition on the polyhedral characterization, we refer to the article by Knuth [13].

4.2 The OCT Polytope and Its LP Relaxation

Let $\mathrm{OCT}(G)$ denote the convex hull of the incidence vectors of all the odd cycle transversals of G.

$$\mathrm{OCT}(G) = conv\,\{\chi^S \in \{0,1\}^n \mid S \subseteq V(G) \text{ is an odd cycle transversal of } G\}$$

Clearly, $\mathrm{oct}(G, w)$ is equal to $\min w^T x$ subject to $x \in \mathrm{OCT}(G)$. Define the convex set $\mathrm{QOCT}(G)$, the clique-constrained odd cycle transversal polytope insisting a lower bound on the variables sum for every clique.

$$\mathrm{QOCT}(G) = \{x \in [0,1]^n : \textstyle\sum_{i \in Q} x(i) \geq |Q| - 2, \ \ \forall Q \text{ a clique in } G\}$$

Now, $\mathrm{OCT}(G) \subseteq \mathrm{QOCT}(G)$ and $\mathrm{oct}(G, w)$ is at least the weight of an optimum point in $\mathrm{QOCT}(G)$. We then have the following linear programming relaxation for OCT.

$$\textbf{LP-Oct(}G\textbf{)}: \quad \min \quad w^T x \quad \text{subject to} \quad x \in \mathrm{QOCT}(G)$$

We next show that constraints corresponding to maximal cliques suffice to describe $\mathrm{QOCT}(G)$ and $\mathrm{QVC}(G)$. We use this property crucially in the approximation algorithm.

Lemma 1. *Let* $\mathcal{Q}(G)$ *denote the set of maximal cliques in* G. *Then, we have* $\mathrm{QVC}(G) = \{x \in [0,1]^n : \sum_{i \in Q} x(i) \geq |Q| - 1, \ \forall Q \in \mathcal{Q}(G)\}$ *and* $\mathrm{QOCT}(G) = \{x \in [0,1]^n : \sum_{i \in Q} x(i) \geq |Q| - 2, \ \forall Q \in \mathcal{Q}(G)\}$.

We now show that, if G is a perfect graph, then an optimum solution to LP-Oct(G) can be obtained in polynomial time.

Lemma 2. *If* G *is perfect, then there exists a polynomial-time separation oracle for* $\mathrm{QOCT}(G)$. *Thus, if* G *is perfect, then the optimization problem over* $\mathrm{QOCT}(G)$ *is polynomial-time solvable.*

Next, we show how to round a fractional optimum solution of LP-Oct(G) to obtain an odd cycle transversal that is a 2-approximation.

4.3 A 2-Approximation for OCT on Perfect Graphs

We now design a polynomial-time 2-approximation algorithm for OCT on perfect graphs using all the properties we have proved. Consider an optimum point x^* of $QOCT(G)$. Let $V(G)$ be partitioned into $V_0 = \{i \in V(G) : x^*(i) < \frac{1}{2}\}$ and $V_1 = \{i \in V(G) : x^*(i) \geq \frac{1}{2}\}$. We include the set V_1 into the odd cycle transversal T. Then, we obtain a set S that is a 2-approximation for OCT in $G[V_0]$. Finally, we show that $T = V_1 \cup S$ is a 2-approximation for OCT in G.

```
Algorithm Approx-Oct($G, w$)
1. Solve LP-Oct(G) to get a point $x^* = (x^*(1), x^*(2), \cdots, x^*(n))$.
2. Let $V_1 = \{i \in V(G) : x^*(i) \geq \frac{1}{2}\}$. If $G - V_1$ is bipartite, then return $V_1$.
3. Delete $V_1$ and edges of $G - V_1$ that are not in any triangle.
4. Obtain a minimum vertex cover S of the resulting graph $G_0$.
5. Return $V_1 \cup S$.
```

Lemma 3. *The size of a maximum clique in* $G[V_0]$ *is at most 3.*

Thus, it follows that $G[V_0]$ is 3-colorable as G is perfect. Note that OCT remains NP-hard on 3-colorable perfect graphs [3]. We first delete edges that are not a part of any triangle in $G[V_0]$. From [22], this preprocessing guarantees the perfectness of the resultant graph. Since the sets of triangles in both the graphs are same, to get a minimum odd cycle transversal for $G[V_0]$, it suffices to get a minimum odd cycle transversal of this preprocessed graph, denoted by G_0. Let n_0 denote the number of vertices in G_0.

Lemma 4. *If* x *is a point in* $QOCT(G_0) \cap [0, \frac{1}{2})^{n_0}$, *then* $2x$ *is a point in* $QVC(G_0)$.

Observe that the preprocessing rule on G_0 is crucial to the proof of Lemma 4. If G_0 had an edge $e = \{i, j\}$ that is not a part of any triangle, then $QOCT(G_0)$ will have a constraint $x(i) + x(j) \geq 2 - 2 = 0$ corresponding to e. In such a case, the point $2x$ defined above cannot be guaranteed to respect the $QVC(G)$ constraint $2x(i) + 2x(j) \geq 2 - 1 = 1$. Now, for a minimum weight vertex cover S of G_0, we have $w(S) \leq 2 \sum_{i \in V_0} w(i)x(i)$. By taking x as $x^*_{|V(G_0)}$, the vector x^* restricted to the coordinates corresponding to vertices in G_0, we have the following result.

Theorem 3. *Given a perfect graph* G *with vertex weighting* w, *Algorithm Approx-Oct(*G, w*) returns an odd cycle transversal* T *of* G *with* $w(T) \leq 2 \cdot \mathrm{oct}(G, w)$ *in polynomial time.*

5 Hardness of OCT on Perfect Graphs

We show various hardness results for OCT on perfect graphs in the approximation and parameterized complexity frameworks. The following result holds from the NP-hardness of OCT on perfect graphs shown in [3].

Lemma 5. *Given a graph* G *on* n *vertices, there is a polynomial-time algorithm that produces a perfect graph* H *such that* G *has a vertex cover of size* k *iff* H *has an odd cycle transversal of size* $n + k$.

Consider the following linear programming relaxation for VERTEX COVER.

LP-Vc(G): $\min w^T x$ subject to $x \in \{x \in [0,1]^n : x(i) + x(j) \geq 1, \forall \{i,j\} \in E(G)\}$

Let $x^* = (x^*(1), \ldots, x^*(n))$ be an optimum solution to LP-Vc(G). Let $vc^*(G) = \sum_{i \in V(G)} x^*(i)$. Let $\langle G, k \rangle$ be a VERTEX COVER instance (decision version). If $k < vc^*(G)$, then we can declare that G has no vertex cover of size k. Thus, without loss of generality, we can assume that $k \geq vc^*(G)$. The following theorem holds from the results of [19] and [20].

Theorem 4. *Given a graph G and a positive integer k, there is a polynomial-time algorithm that produces a graph G' and an integer k' such that G has a vertex cover of size k iff G' has a vertex cover of size k' where $k' \leq k$. Further, $(\frac{1}{2}, \ldots, \frac{1}{2})$ is the unique optimum solution to LP-Vc(G').*

It follows that VERTEX COVER (decision version) remains NP-complete on instances $\langle G, k \rangle$ satisfying the properties that $(\frac{1}{2}, \ldots, \frac{1}{2})$ is the optimum solution to LP-Vc(G) and $n = 2vc^*(G) \leq 2k$. Therefore, from this observation and Lemma 5, we have the following corollary.

Corollary 2. *Given a graph G, there is a polynomial-time algorithm that produces a perfect graph H such that G has a vertex cover of size k iff H has an odd cycle transversal of size at most $3k$.*

Using Corollary 2, we show that OCT on perfect graphs is APX-hard.

Theorem 5. *For any $\epsilon > 0$, if VERTEX COVER has no polynomial-time $(c - \epsilon)$-approximation algorithm, then OCT on perfect graphs has no polynomial-time $(c' - \epsilon)$-approximation algorithm where $c' = \frac{1}{3}(c + 2)$.*

It is known that if there exists a c-approximation algorithm for VERTEX COVER with $c < 1.36$, then $P = NP$ [7]. Moreover, assuming the unique games conjecture, if VERTEX COVER has an c-approximation algorithm with $c < 2$, then $P = NP$ [12]. From these results and Lemma 5, we have the following result.

Corollary 3. *If there is a c-approximation algorithm for OCT on perfect graphs with $c < 1.12$, then $P = NP$. Further, assuming the unique games conjecture, if there is a c-approximation algorithm for OCT on perfect graphs with $c < \frac{4}{3}$, then $P = NP$.*

In the parameterized setting, OCT on perfect graphs has an $O^*(2^k)$-time algorithm where k is the size of the odd cycle transversal that we seek [16]. A kernelization algorithm for a parameterized problem is a polynomial-time algorithm that produces an equivalent instance (called a kernel) whose size is just a function of the parameter (which in this case is k). A kernel consisting of $O(k^2)$ vertices and $O(k^3)$ edges exists for determining whether a family of sets of size 3 has a hitting set of size at most k [1], which can be used to obtain kernel of the same size for OCT on perfect graphs. For the problem of determining if a graph has a vertex cover of size k, no $O^*(2^{o(k)})$ time algorithm exists under the Exponential Time Hypothesis [4] and no $O(k^{2-\epsilon})$ (edges) kernel for any $\epsilon > 0$ exists unless the polynomial hierarchy collapses to the third level [6]. Thus, from Corollary 2, we obtain the following result.

Theorem 6. *The following results hold for the problem of determining if the given perfect graph has an odd cycle transversal of size* k.

1. *No* $O^*(2^{o(k)})$ *algorithm is possible under the Exponential Time Hypothesis.*
2. *No* $O(k^{2-\epsilon})$ *(edges) kernel is possible for any* $\epsilon > 0$ *unless the polynomial hierarchy collapses to the third level.*

Note that the proof of subquadratic kernelization hardness in Theorem 6 uses, apart from Lemma 5, also a *size preserving* reduction from OCT to VERTEX COVER described in [17].

6 Conclusion

Our approximation algorithms not only improve the known bounds for OCT and r-CLIQUE TRANSVERSAL on perfect graphs but also combines various properties of perfectness. Algorithm Approx-r-CT uses the fact that a K_{r+1}-free perfect graph is r-partite and that this property is hereditary. Algorithm Approx-Oct crucially uses the polynomial-time solvability of VERTEX COVER and the equivalence of the independent set and the clique-constrained independent set polytopes for perfect graphs. It also exploits the edge deletion properties that preserve perfectness. It would be interesting to explore if a similar combination of perfectness properties could improve the approximation ratio or prove tightness.

The relationship between VERTEX COVER and OCT has been witnessed to a significant extent in the parameterized complexity framework [11, 15–17, 23]. The first bound of $O^*(3^k)$ [21] for determining whether a given graph has an odd cycle transversal of size at most k was recently improved to $O^*(2.3146^k)$ by reducing OCT to a version of VERTEX COVER in polynomial time [17]. Using the same reduction as an oracle, a simpler $O^*(3^k)$ algorithm for OCT is described in [15]. An $O^*(2^k)$ bound was obtained for perfect graphs by transforming the problem into 2^{k+1} instances of bipartite vertex cover (which is polynomial-time solvable) [16]. In this paper, we have shown two ways of using VERTEX COVER algorithms (in graphs and hypergraphs) to obtain an approximation for OCT. It would be interesting to investigate whether the bounds for OCT or r-CLIQUE TRANSVERSAL in general can be improved using the ideas in this paper.

We contrast the optimization complexities of $VC(G)$ and $QVC(G)$ with that of $OCT(G)$ and $QOCT(G)$. From the theory of blockers/antiblockers, it is known that a polytope, its blocker and its antiblocker have same polynomial-time optimization complexities [9]. The optimization problems over $VC(G)$ and $QVC(G)$ are NP-hard in general and polynomial-time solvable on perfect graphs. This is due to the fact that $STAB(G)$ and $QSTAB(\overline{G})$ are antiblockers of each other. However, on perfect graphs, optimizing over $OCT(G)$ is known to be NP-hard while we have shown that $QOCT(G)$ is optimizable in polynomial time. We also observe that the definition of $QOCT(G)$ generalizes to provide another relaxation for the r-clique transversal polytope with similar computational results and structural properties. Exploring the relationship between $OCT(G)$ and $QOCT(G)$ is an interesting direction of research that could lead to an understanding of the r-clique transversal polytope of perfect graphs.

References

1. Abu-Khzama, F.N.: A kernelization algorithm for d-hitting set. Journal of Computer and System Sciences 76(7), 524–531 (2010)
2. Berge, C.: Färbung von graphen, deren sämtliche bzw. deren ungerade kreise starr sind (zusammenfassung). Wissenschaftliche Zeitschrift, Martin Luther Universität Halle-Wittenberg Mathematisch-Naturwissenschaftliche Reihe 10, 114–115 (1961)
3. Berry, L.A., Kennedy, W.S., King, A.D., Li, Z., Reed, B.A.: Finding a maximum-weight induced k-partite subgraph of an i-triangulated graph. Discrete Applied Mathematics 158(7), 765–770 (2010)
4. Cai, L., Juedes, D.: On the existence of subexponential parameterized algorithms. Journal of Computer and System Sciences 67, 789–807 (2003)
5. Chudnovsky, M., Robertson, N., Seymour, P.D., Thomas, R.: The strong perfect graph theorem. Annals of Mathematics 164(1), 51–229 (2006)
6. Dell, H., Melkebeek, D.: Satisfiability allows no nontrivial sparsification unless the polynomial-time hierarchy collapses. In: Proceedings of the ACM Symposium on Theory of Computing, pp. 251–260 (2010)
7. Dinur, I., Safra, S.: On the hardness of approximating minimum vertex cover. Annals of Mathematics 162(1), 439–485 (2005)
8. Golumbic, M.C.: Algorithmic Graph Theory and Perfect Graphs, 2nd edn. Elsevier Science B.V. (2004)
9. Grötschel, M., Lovász, L., Schrijver, A.: Geometric Algorithms and Combinatorial Optimization. Springer (1988)
10. Guruswami, V., Lee, E.: Inapproximability of feedback vertex set for bounded length cycles. Electronic Colloquium on Computational Complexity (ECCC) 21, 6 (2014)
11. Iwata, Y., Oka, K., Yuichi, Y.: Linear-time fpt algorithms via network flow. In: Proceedings of the ACM-SIAM Symposium on Discrete Algorithms, pp. 1749–1761 (2014)
12. Khot, S., Regev, O.: Vertex cover might be hard to approximate to within $2 - \epsilon$. Journal of Computer and System Sciences 74(3), 335–349 (2008)
13. Knuth, D.: The sandwich theorem. Electronic Journal of Combinatorics 1(A1), 1–48 (1994)
14. Kratsch, S., Wahlström, M.: Compression via matroids: A randomized polynomial kernel for odd cycle transversal. In: Proceedings of the ACM-SIAM Symposium on Discrete Algorithms, pp. 94–103 (2012)
15. Krithika, R., Narayanaswamy, N.S.: Another disjoint compression algorithm for odd cycle transversal. Information Processing Letters 113(22–24), 849–851 (2013)
16. Krithika, R., Narayanaswamy, N.S.: Parameterized algorithms for (r, l)-partization. Journal of Graph Algorithms and Applications 17(2), 129–146 (2013)
17. Lokshtanov, D., Narayanaswamy, N.S., Raman, V., Ramanujan, M.S., Saurabh, S.: Faster parameterized algorithms using linear programming. CoRR abs/1203.0833. A preliminary version appeared in the Proceedings of STACS (2012)
18. Lovász, L.: On minimax theorems of combinatorics. Ph.D thesis, Matemathikai Lapok 26, 209–264 (1975)

19. Nemhauser, G.L., Trotter, L.E.: Properties of vertex packing and independence system polyhedra. Mathematical Programming 6(1), 48–61 (1974)
20. Nemhauser, G.L., Trotter, L.E.: Vertex packings: Structural properties and algorithms. Mathematical Programming 8(1), 232–248 (1975)
21. Reed, B.A., Smith, K., Vetta, A.: Finding odd cycle transversals. Operations Research Letters 32, 299–301 (2004)
22. Wagler, A.: Critical and anticritical edges in perfect graphs. In: Brandstädt, A., Van Le, B. (eds.) WG 2001. LNCS, vol. 2204, pp. 317–327. Springer, Heidelberg (2001)
23. Wahlström, M.: Half-integrality, lp-branching and fpt algorithms. In: Proceedings of the ACM-SIAM Symposium on Discrete Algorithms, pp. 1762–1781 (2014)

Representative Sets of Product Families

Fedor V. Fomin[1], Daniel Lokshtanov[1], Fahad Panolan[2], and Saket Saurabh[1,2]

[1] University of Bergen, Norway
`{fomin,daniello}@ii.uib.no`
[2] The Institute of Mathematical Sciences, Chennai, India
`{fahad,saket}@imsc.res.in`

Abstract. A subfamily $\mathcal{F}'$ of a set family $\mathcal{F}$ is said to *q-represent* $\mathcal{F}$ if for every $A \in \mathcal{F}$ and B of size q such that $A \cap B = \emptyset$ there exists a set $A' \in \mathcal{F}'$ such that $A' \cap B = \emptyset$. In a recent paper [SODA 2014] three of the authors gave an algorithm that given as input a family $\mathcal{F}$ of sets of size p together with an integer q, efficiently computes a q-representative family $\mathcal{F}'$ of $\mathcal{F}$ of size approximately $\binom{p+q}{p}$, and demonstrated several applications of this algorithm. In this paper, we consider the efficient computation of q-representative sets for *product* families $\mathcal{F}$. A family $\mathcal{F}$ is a product family if there exist families $\mathcal{A}$ and $\mathcal{B}$ such that $\mathcal{F} = \{A \cup B \ : \ A \in \mathcal{A}, B \in \mathcal{B}, A \cap B = \emptyset\}$. Our main technical contribution is an algorithm which given $\mathcal{A}$, $\mathcal{B}$ and q computes a q-representative family $\mathcal{F}'$ of $\mathcal{F}$. The running time of our algorithm is *sublinear* in $|\mathcal{F}|$ for many choices of $\mathcal{A}$, $\mathcal{B}$ and q which occur naturally in several dynamic programming algorithms. We also give an algorithm for the computation of q-representative sets for product families $\mathcal{F}$ in the more general setting where q-representation also involves independence in a matroid in addition to disjointness. This algorithm considerably outperforms the naive approach where one first computes $\mathcal{F}$ from $\mathcal{A}$ and $\mathcal{B}$, and then computes the q-representative family $\mathcal{F}'$ from $\mathcal{F}$.

We give two applications of our new algorithms for computing q-representative sets for product families. The first is a $3.8408^k n^{\mathcal{O}(1)}$ deterministic algorithm for the Multilinear Monomial Detection (k-MlD) problem. The second is a significant improvement of deterministic dynamic programming algorithms for "connectivity problems" on graphs of bounded treewidth.

1 Introduction

Let $M = (E, \mathcal{I})$ be a matroid and let $\mathcal{S} = \{S_1, \ldots, S_t\}$ be a family of subsets of E of size p. A subfamily $\widehat{\mathcal{S}} \subseteq \mathcal{S}$ is *q-representative* for $\mathcal{S}$ if the following holds. For every set $Y \subseteq E$ of size at most q, if there is a set $X \in \mathcal{S}$ disjoint from Y with $X \cup Y \in \mathcal{I}$, then there is a set $\widehat{X} \in \widehat{\mathcal{S}}$ disjoint from Y with $\widehat{X} \cup Y \in \mathcal{I}$. By the classical result of Lovász [14], there exists a representative family $\widehat{\mathcal{S}} \subseteq^q_{rep} \mathcal{S}$ with at most $\binom{p+q}{p}$ sets. However, it is a very non-trivial question how to construct such a representative family efficiently. It appeared already in the 1980's that representative families can be extremely useful in dynamic programming algorithms and that faster computation of representative families leads to more efficient algorithms.

Recently, three of the authors in [10] showed that a q-representative family with at most $\binom{p+q}{p}$ sets can be found in $\mathcal{O}\left(\binom{p+q}{p}tp^{\omega} + t\binom{p+q}{q}^{\omega-1}\right)$ operations over the field representing the matroid. Here, $\omega < 2.373$ is the matrix multiplication exponent. For the special case of uniform matroids on n elements, a faster algorithm computing a representative family in time $\mathcal{O}((\frac{p+q}{q})^q \cdot 2^{o(p+q)} \cdot t \cdot \log n)$ was given. The results of Fomin et al. [10] improved over previous work by Monien [17] and Marx [15,16], and led to the fastest known deterministic parameterized algorithms for k-Path, k-Tree, and more generally, for k-Subgraph Isomorphism, where the k-vertex pattern graph is of constant treewidth [10].

All currently known algorithms that use fast computation of representative sets as a subroutine are based on dynamic programming. It is therefore very tempting to ask whether it is possible to compute representative sets faster for families that arise naturally in dynamic programs, than for general families. A class of families which often arises in dynamic programs is the class of *product* families; a family $\mathcal{F}$ is the *product* of $\mathcal{A}$ and $\mathcal{B}$ if $\mathcal{F} = \{A \cup B \ : \ A \in \mathcal{A}, B \in \mathcal{B} \wedge A \cap B = \emptyset\}$. Product families naturally appear in dynamic programs where sets represent partial solutions and two partial solutions can be combined if they are disjoint. For an example, in the k-Path problem partial solutions are vertex sets of paths starting at a particular root vertex v, and two such paths may be combined to a longer path if and only if they are disjoint (except for overlapping at v). Many other examples exist—essentially product families can be thought of as a *subset convolution* [2,3], and the wide applicability of the fast subset convolution technique of Bjorklund et al [4] is largely due to the frequent demand to compute product families in dynamic programs.

Our Results. Our main technical contributions are two algorithms for the computation of representative sets for product families, one for uniform, and one for linear matroids. For uniform matroids we give an algorithm which given an integer q and families $\mathcal{A}$, $\mathcal{B}$ of sets of sizes p_1 and p_2 over the ground set of size n, computes a q-representative family $\mathcal{F}'$ of $\mathcal{F}$. The running time of our algorithm is *sublinear* in $|\mathcal{F}|$ for many choices of $\mathcal{A}$, $\mathcal{B}$ and q which occur naturally in several dynamic programming algorithms. For example, let q, p_1, p_2 be integers. Let $k = q + p_1 + p_2$ and suppose that we have families $\mathcal{A}$ and $\mathcal{B}$, which are $(k - p_1)$ and $(k - p_2)$-representative families. Then the sizes of these families are roughly $|\mathcal{A}| = \binom{k}{p_1}$ and $|\mathcal{B}| = \binom{k}{p_2}$. In particular, when $p_1 = p_2 = \lceil k/2 \rceil$ both families are of size roughly 2^k, and thus the cardinality of $\mathcal{F}$ is approximately 4^k. On the other hand, for any choice of p_1, p_2, and k, our algorithm outputs a $(k - p_1 - p_2)$-representative family of $\mathcal{F}$ of size roughly $\binom{k}{p_1+p_2}$ in time $3.8408^k n^{\mathcal{O}(1)}$. For many choices of p_1, p_2 and q our algorithm runs significantly faster than $3.8408^k n^{\mathcal{O}(1)}$. The expression capturing the running time dependence on p_1, p_2 and q can be found in Theorem 1.

Our second algorithm is for computing representative families of product families, when the universe is also enriched with a linear matroid. More formally, let $M = (E, \mathcal{I})$ be a matroid and let $\mathcal{A}, \mathcal{B} \subseteq \mathcal{I}$. Then let $\mathcal{F} = \mathcal{A} \bullet \mathcal{B} = \{A \cup B : A \cup B \in \mathcal{I}, A \in \mathcal{A}, B \in \mathcal{B}$ and $A \cap B = \emptyset\}$. Just as for uniform matroids, a naive approach for computing a representative family of $\mathcal{F}$ would be to compute the product $\mathcal{A} \bullet \mathcal{B}$ first and then compute a representative family of the product. The fastest currently known algorithm for computing a representative family is by Fomin et al. [10] and has running time approximately $\binom{p+q}{p}^{\omega-1}|\mathcal{F}|$. We give an algorithm that significantly outperforms the naive approach. An appealing feature of our algorithm is that it works by reducing the computation of a representative family for $\mathcal{F}$ to the computation of representative families for many smaller families. Thus an improved algorithm for the computation of representative sets for general families will automatically accelerate our algorithm for product families as well. The expression of the running time of our algorithm can be found in Theorem 2.

Applications. Our first application is a deterministic algorithm for the following parameterized version of multilinear monomial testing.

Multilinear Monomial Detection (k-MlD) **Parameter:** k
Input: An arithmetic circuit C over $\mathbb{Z}^+$ representing a polynomial $P(X)$ over $\mathbb{Z}^+$.
Question: Does $P(X)$ construed as a sum of monomials contain a multilinear monomial of degree k?

This is the central problem in the algebraic approach of Koutis and Williams for designing fast parameterized algorithms [12,13,20]. The idea behind the approach is to translate a given problem into the language of algebra by reducing it to the problem of deciding whether a constructed polynomial has a multilinear monomial of degree k. As it is mentioned implicitly by Koutis in [12], k-MlD can be solved in time $(2e)^k n^{\mathcal{O}(1)}$, where n is the input length, by making use of color coding. The color coding technique of Alon, Yuster and Zwick [1] is a fundamental and widely used technique in the design of parameterized algorithms. It appeared that most of the problems solvable by making use of color coding can be reduced to a multilinear monomial testing. Williams [20] gave a *randomized* algorithm solving k-MlD in time $2^k n^{\mathcal{O}(1)}$. The algorithms based on the algebraic method of Koutis-Williams provide a dramatic improvement for a number of fundamental problems [6,5,9,11,12,13,20].

The advantage of the algebraic approach over color coding is that for a number of parameterized problems, the algorithms based on this approach have much better exponential dependence on the parameter. On the other hand color coding based algorithms admit direct derandomization [1] and are able to handle integer weights with running time overhead poly-logarithmic in the weights. Obtaining deterministic algorithms matching the running times of the algebraic methods, but sharing these nice features of color coding remains a challenging open problem. Our deterministic algorithm for k-MlD is the first non-trivial step towards resolving this problem. In fact, our algorithm solves a weighted version of k-MlD, where the elements of X are assigned weights and the task is to find a k-multilinear term with minimum weight. The running time of our deterministic algorithm is $\mathcal{O}(3.8408^k 2^{o(k)} s(C) n \log W \log^2 n)$, where $s(C)$ is the size of the circuit and W is the maximum weight of an element from X. We also provide an algorithm for a more general version of multilinear monomial testing, where variables of a monomial should form an independent set of a linear matroid.

The second application of our fast computation of representative families is for dynamic programming algorithms on graphs of bounded treewidth. It is well known that many intractable problems can be solved efficiently when the input graph has bounded treewidth. Moreover, many fundamental problems like Maximum Independent Set or Minimum Dominating Set can be solved in time $2^{\mathcal{O}(t)}n$. On the other hand, it was believed until very recently that for some "connectivity" problems such as Hamiltonian Cycle or Steiner Tree no such algorithm exists. In their breakthrough paper, Cygan et al. [8] introduced a new algorithmic framework called Cut&Count and used it to obtain $2^{\mathcal{O}(t)}n^{\mathcal{O}(1)}$ time Monte Carlo algorithms for a number of connectivity problems. Recently, Bodlaender et al. [7] obtained the first deterministic single-exponential algorithms for these problems using two novel approaches. One of the approaches of Bodlaender et al. is based on rank estimations in specific matrices and the second based on matrix-tree theorem and computation of determinants. In [10], Fomin et al. used efficient algorithms for computing representative families of linear matroids to provide yet another approach for single-exponential algorithms on graphs of bounded treewdith.

It is interesting to note that for a number of connectivity problems such as Steiner Tree or Feedback Vertex Set the "bottleneck" of treewidth based dynamic programming algorithms is the *join* operation. For example, as it was shown by Bodlaender et al. in [7], Feedback Vertex Set and Steiner Tree can be solved in time $\mathcal{O}\left((1+2^{\omega})^{\mathbf{pw}}\mathbf{pw}^{\mathcal{O}(1)}n\right)$ and $\mathcal{O}\left((1+2^{\omega+1})^{\mathbf{tw}}\mathbf{tw}^{\mathcal{O}(1)}n\right)$, where $\mathbf{pw}$ and $\mathbf{tw}$ are the pathwidth and the treewidth of the input graph. The reason for the difference in the exponents of these two algorithms is due to the cost of the join operation, which is required for treewidth and does not occur for pathwidth. For many computational problems on graphs of bounded treewidth in the join nodes of the decomposition, the family of partial solutions is the product of the families of its children, and we wish to store a representative set (for a graphic matroid) for this product family. Here our second algorithm comes into play. By making use of this algorithm one can obtain faster deterministic algorithms for many connectivity problems. We exemplify this by providing an algorithm with running time $\mathcal{O}\left((1+2^{\omega-1}\cdot 3)^{\mathbf{tw}}\mathbf{tw}^{O(1)}n\right)$ for Steiner Tree by changing the representative sets computation in the join node in the algorithm presented in [10].

Our Methods. The engine behind our algorithm for the computation of representative sets of product families is a new construction of pseudorandom coloring families. A *coloring* of a universe U is simply a function $f : U \to \{red, blue\}$. Consider a pair of disjoint sets A and B, with $|A| = p$ and $|B| = q$. A random coloring which colors each element in U red with probability $\frac{p}{p+q}$ and blue with probability $\frac{q}{p+q}$ will color A red and B blue with probability roughly $\frac{1}{\binom{p+q}{p}}$. Thus a family of slightly more than $\binom{p+q}{p}$ such random colorings will contain, with high probability, for each pair of disjoint sets A and B, with $|A| = p$ and $|B| = q$ a function which colors A red and B blue. The fast computation of representative sets of Fomin et al. [10] deterministically constructs a collection of colorings which mimics this property of random coloring families. The colorings in the family are used to *witness disjointedness*, since a coloring which colors A red and B blue certifies that A and B are disjoint. In our setting we can use such coloring families both for witnessing disjointedness in the computation of representative sets, and in the computation of $\mathcal{F} = \mathcal{A} \bullet \mathcal{B}$. After all, each set in $\mathcal{F}$ is the disjoint union of a set in $\mathcal{A}$ and a set in $\mathcal{B}$. In order to make this idea work we need to make a deterministic construction of coloring families which need to satisfy more properties of random colorings than the construction from [10]. We believe that the new construction of coloring families will find applications beyond our algorithm. The new construction can be used to speed-up the deterministic algorithm for k-PATH of Fomin et al. [10] from $\mathcal{O}(2.851^k n \log^2 n)$ to $\mathcal{O}(2.619^k n \log^2 n)$. Shachnai and Zehavi [19] also obtained the result for k-PATH independently by giving a time-size tradeoff version of coloring families used in [10]. However, this construction is not sufficient to give our results on product families.

For linear matroids, our algorithm computes a representative family $\mathcal{F}'$ of $\mathcal{F} = \mathcal{A} \bullet \mathcal{B}$ as follows. First the family $\mathcal{F}$ is broken up into many smaller families $\mathcal{F}_1, \ldots, \mathcal{F}_t$, then a representative family $\mathcal{F}'_i$ is computed for each $\mathcal{F}_i$. Finally $\mathcal{F}'$ is obtained by computing a representative family of $\bigcup_i \mathcal{F}'_i$ using the algorithm of Fomin et al [10] for computing representative families. The speedup over the naive method is due to the fact that (a) $\bigcup_i \mathcal{F}'_i$ is much smaller than $\mathcal{F}$ and (b) that each $\mathcal{F}_i$ has a certain structure which ensures better upper bounds on the size of $\mathcal{F}'_i$, and allows $\mathcal{F}'_i$ to be computed faster.

2 Preliminaries

In this section we give various definitions which we make use of in the paper.

Sets, Functions and Constants. Let $[n] = \{0, \ldots, n-1\}$ and $\binom{[n]}{i} = \{X \mid X \subseteq [n],\ |X| = i\}$. Furthermore for any ground set U, we use 2^U to denote the family of all subsets of U. We call a function $f : 2^U \to \mathbb{N}$, *additive* if for any subsets X and Y of U we have that $f(X) + f(Y) = f(X \cup Y) - f(X \cap Y)$.

A monomial $Z = x_1^{s_1} \cdots x_n^{s_n}$ of a polynomial $P(x_1, \ldots, x_n)$ is called *multilinear* if $s_i \in \{0, 1\}$ for all $i \in \{1, \ldots, n\}$. We say a monomial $Z = x_1^{s_1} \cdots x_n^{s_n}$ is a *k-multilinear term*, if Z is multilinear and $\sum_{i=1}^{n} s_i = k$. Throughout the paper we use ω to denote the matrix multiplication exponent. The current best known bound on $\omega < 2.373$ [21].

Now we give definitions related to matroids and representative families. For a broader overview on matroids we refer to [18].

Definition 1. *A pair $M = (E, \mathcal{I})$, where E is a ground set and $\mathcal{I}$ is a family of subsets (called independent sets) of E, is a* matroid *if it satisfies the following conditions: (I1) $\phi \in \mathcal{I}$, (I2) If $A' \subseteq A$ and $A \in \mathcal{I}$ then $A' \in \mathcal{I}$. (I3) If $A, B \in \mathcal{I}$ and $|A| < |B|$, then $\exists\ e \in (B \setminus A)$ such that $A \cup \{e\} \in \mathcal{I}$.*

An inclusion-wise maximal set of $\mathcal{I}$ is called a basis of the matroid. All bases of a matroid M have the same size, called the *rank* of the matroid M, and is denoted by $\mathsf{rank}(M)$. The *uniform matroids* are among the simplest examples of matroids. A pair $M = (E, \mathcal{I})$ over an n-element ground set E, is called a uniform matroid if the family of independent sets is given by $\mathcal{I} = \{A \subseteq E \mid |A| \leq k\}$, where k is some constant. This matroid is also denoted as $U_{n,k}$.

Let A be a matrix over an arbitrary field $\mathbb{F}$ and let E be the set of columns of A. Given A we define the matroid $M = (E, \mathcal{I})$ as follows. A set $X \subseteq E$ is independent (that is $X \in \mathcal{I}$) if the corresponding columns are linearly independent over $\mathbb{F}$. The matroids that can be defined by such a construction are called *linear matroids*, and if a matroid can be defined by a matrix A over a field $\mathbb{F}$, then we say that the matroid is representable over $\mathbb{F}$. That is, a matroid $M = (E, \mathcal{I})$ of rank d is representable over a field $\mathbb{F}$ if there exist vectors in $\mathbb{F}^d$ correspond to the elements such that linearly independent sets of vectors correspond to independent sets of the matroid. A matroid $M = (E, \mathcal{I})$ is called *representable* or *linear* if it is representable over some field $\mathbb{F}$.

Given two families of independent sets $\mathcal{A}$ and $\mathcal{B}$ of a matroid $M = (E, \mathcal{I})$, we define $\mathcal{A} \bullet \mathcal{B} = \{A \cup B \ : A \cup B \in \mathcal{I}, A \in \mathcal{A}, B \in \mathcal{B} \text{ and } A \cap B = \emptyset\}$.

We now define what it means for a family to be a q-representative family of a given family.

Definition 2 (Min/Max q-Representative Family [10,16]). *Given a matroid $M = (E, \mathcal{I})$, a family $\mathcal{S}$ of subsets of E and a non-negative weight function $w : \mathcal{S} \to \mathbb{N}$ we say that a subfamily $\widehat{\mathcal{S}} \subseteq \mathcal{S}$ is* min q-representative *(*max q-representative*) for $\mathcal{S}$ if the following holds: for every set $Y \subseteq E$ of size at most q, if there is a set $X \in \mathcal{S}$ disjoint from Y with $X \cup Y \in \mathcal{I}$, then there is a set $\widehat{X} \in \widehat{\mathcal{S}}$ disjoint from Y with (a) $\widehat{X} \cup Y \in \mathcal{I}$; and (b) $w(\widehat{X}) \leq w(X)$ ($w(\widehat{X}) \geq w(X)$). We use $\widehat{\mathcal{S}} \subseteq^q_{minrep} \mathcal{S}$ ($\widehat{\mathcal{S}} \subseteq^q_{maxrep} \mathcal{S}$) to denote a min q-representative (max q-representative) family for $\mathcal{S}$.*

3 Representative Set Computation for Product Families

In this section we design a faster algorithm to find q-representative family for product families. Our main technical tool is a generalization of *n-p-q-separating collections* defined in [10] to compute q-representative families of an arbitrary family. In fact we design a *family* of n-p-q-separating collections of various sizes governed by a parameter $0 < x < 1$. The construction of generalized n-p-q-separating collection goes along the lines of the proof given in [10] with a few non-trivial differences. Finally, we combine two n-p-q-separating collections obtained with different parameters to obtain the desired algorithm for product families.

3.1 Generalized n-p-q-Separating Collections

We start with the formal definition of a *generalized n-p-q-separating collection.*

Definition 3. *A generalized n-p-q-separating collection $\mathcal{C}$ is a tuple $(\mathcal{F}, \chi, \chi')$, where $\mathcal{F}$ is a family of sets over a universe U of size n, χ is a function from $\bigcup_{p' \leq p} \binom{U}{p'}$ to $2^{\mathcal{F}}$ and χ' is a function from $\bigcup_{q' \leq q} \binom{U}{q'}$ to $2^{\mathcal{F}}$ such that the following properties are satisfied*

1. *for every $A \in \bigcup_{p' \leq p} \binom{U}{p'}$ and $F \in \chi(A)$, $A \subseteq F$,*
2. *for every $B \in \bigcup_{q' \leq q} \binom{U}{q'}$ and $F \in \chi'(B)$, $F \cap B = \emptyset$,*
3. *for every pairwise disjoint sets $A_1 \in \binom{U}{p_1}, A_2 \in \binom{U}{p_2}, \ldots, A_r \in \binom{U}{p_r}$ and $B \in \binom{U}{q}$ such that $p_1 + \ldots + p_r = p$, $\exists F \in \chi(A_1) \cap \chi(A_2) \ldots \chi(A_r) \cap \chi'(B)$.*

The size of $(\mathcal{F}, \chi, \chi')$ is $|\mathcal{F}|$, the (χ, p')-degree of $(\mathcal{F}, \chi, \chi')$ for $p' \leq p$ is $\max_{A \in \binom{U}{p'}} |\chi(A)|$, *and the (χ', q')-degree of $(\mathcal{F}, \chi, \chi')$ for $q' \leq q$ is* $\max_{B \in \binom{U}{q'}} |\chi'(B)|$.

A *construction* of generalized separating collections is a data structure, that given n, p and q initializes and outputs a family $\mathcal{F}$ of sets over the universe U of size n. After the initialization one can query the data structure by giving it a set $A \in \bigcup_{p' \leq p} \binom{U}{p'}$ or $B \in \bigcup_{q' \leq q} \binom{U}{q'}$, the data structure then outputs a family $\chi(A) \subseteq 2^{\mathcal{F}}$ or $\chi'(B) \subseteq 2^{\mathcal{F}}$ respectively. Together the tuple $\mathcal{C} = (\mathcal{F}, \chi, \chi')$ computed by the data structure should form a *generalized n-p-q-separating collection.*

We call the time the data structure takes to initialize and output $\mathcal{F}$ the *initialization time.* The (χ, p')*-query time*, $p' \leq p$, of the data structure is the maximum time the data structure uses to compute $\chi(A)$ over all $A \in \binom{U}{p'}$. Similarly, the (χ', q')*-query time*, $q' \leq q$, of the data structure is the maximum time the data structure uses to compute $\chi'(B)$ over all $B \in \binom{U}{q'}$. The initialization time of the data structure and the size of $\mathcal{C}$ are functions of n, p and q. The initialization time is denoted by $\tau_I(n, p, q)$, size of $\mathcal{C}$ is denoted by $\zeta(n, p, q)$. The (χ, p')-query time and (χ, p')-degree of $\mathcal{C}$, $p' \leq p$, are functions of n, p', p, q and is denoted by $Q_{(\chi,p')}(n, p, q)$ and $\Delta_{(\chi,p')}(n, p, q)$ respectively. Similarly, the (χ', q')-query time and (χ', q')-degree of $\mathcal{C}$, $q' \leq q$, are functions of n, q', p, q and are denoted by $Q_{(\chi',q')}(n, p, q)$ and $\Delta_{(\chi',q')}(n, p, q)$ respectively. We are now ready to state a lemma regarding the computation of generalized n-p-q- separating collection and its proof is deferred to the full version of the paper.

Lemma 1 ($\star$).[1] *Given a constant x such that $0 < x < 1$, there is a construction of generalized n-p-q- separating collection with the following parameters*

- *size,* $\zeta(n, p, q) \leq 2^{\mathcal{O}(\frac{p+q}{\log\log\log(p+q)})} \cdot \frac{1}{x^p(1-x)^q} \cdot (p+q)^{\mathcal{O}(1)} \cdot \log n$
- *initialization time,* $\tau_I(n, p, q) \leq 2^{\mathcal{O}(\frac{p+q}{\log\log\log(p+q)})} \cdot \frac{1}{x^p(1-x)^q} \cdot (p+q)^{\mathcal{O}(1)} \cdot n \log n$
- (χ, p')*-degree,* $\Delta_{(\chi,p')}(n, p, q) \leq 2^{\mathcal{O}(\frac{p+q}{\log\log\log(p+q)})} \cdot \frac{1}{x^{p-p'}(1-x)^q} \cdot (p+q)^{\mathcal{O}(1)} \cdot \log n$
- (χ, p')*-query time,* $Q_{(\chi,p')}(n, p, q) \leq 2^{\mathcal{O}(\frac{p+q}{\log\log\log(p+q)})} \cdot \frac{1}{x^{p-p'}(1-x)^q} \cdot (p+q)^{\mathcal{O}(1)} \cdot \log n$

[1] Proofs of results labelled with $\star$ will be provided in the full version.

- (χ', q')*-degree,* $\Delta_{(\chi',q')}(n,p,q) \leq 2^{\mathcal{O}(\frac{p+q}{\log\log\log(p+q)})} \cdot \frac{1}{x^p(1-x)^{q-q'}} \cdot (p+q)^{\mathcal{O}(1)} \cdot \log n$
- (χ', q')*-query time,* $Q_{(\chi',q')}(n,p,q) \leq 2^{\mathcal{O}(\frac{p+q}{\log\log\log(p+q)})} \cdot \frac{1}{x^p(1-x)^{q-q'}} \cdot (p+q)^{\mathcal{O}(1)} \cdot \log n$

3.2 Representative Sets for Product Families

We are ready to give the main theorem about product families using the constructions of generalized n-p-q-separating collections.

Theorem 1 ($\star$)**.** *Let* $\mathcal{L}_1$ *be a* p_1*-family of sets and* $\mathcal{L}_2$ *be a* p_2*-family of sets over a universe* U *of size* n*. Let* $w : 2^U \to \mathbb{N}$ *be an additive weight function. Let* $\mathcal{L} = \mathcal{L}_1 \bullet \mathcal{L}_2$ *and* $p = p_1 + p_2$*. For any* $0 < x_1, x_2 < 1$*, there exist* $\widehat{\mathcal{L}} \subseteq^{k-p_1-p_2}_{minrep} \mathcal{L}$ *of size* $2^{\mathcal{O}(\frac{k}{\log\log\log(k)})} \cdot \frac{1}{x_1^p(1-x_1)^{k-p}} \cdot k^{\mathcal{O}(1)} \log n$ *and it can be computed in time* $\mathcal{O}\left(z(n,k,W) \cdot \left(\frac{1}{x_1^p(1-x_1)^q} + \frac{1}{x_2^{p_1}(1-x_2)^{p_2}} + \frac{|\mathcal{L}_1|}{x_1^{p_2}(1-x_1)^q(1-x_2)^{p_2}} + \frac{|\mathcal{L}_2|}{x_1^{p_1}(1-x_1)^q x_2^{p_1}}\right)\right)$*. Here* $z(n,k,W) = 2^{\mathcal{O}(\frac{k}{\log\log\log(k)})} k^{\mathcal{O}(1)} n \log n \log W$ *and* W *is the maximum weight defined by* w*.*

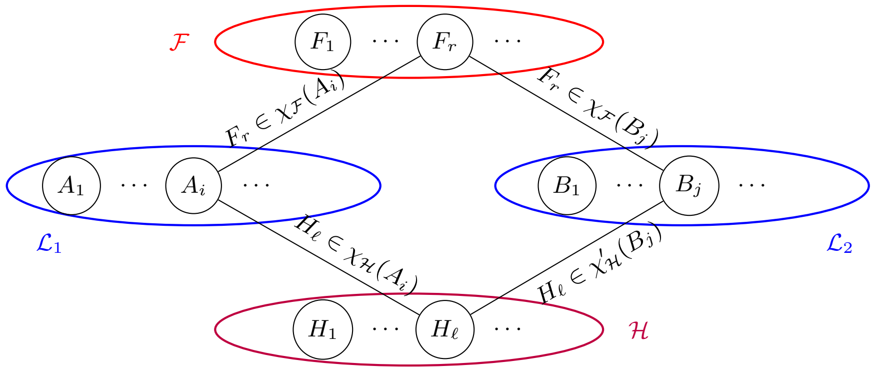
Proof. We set $p = p_1 + p_2$ and $q = k - p$. To obtain the desired construction we first define an auxiliary graph and then use it to obtain the q-representative for the product family $\mathcal{L}$. We first obtain two families of separating collections.

- Apply Lemma 1 for $0 < x_1 < 1$ and construct a n-p-q-separating collection $(\mathcal{F}, \chi_{\mathcal{F}}, \chi'_{\mathcal{F}})$ of size $2^{\mathcal{O}(\frac{p+q}{\log\log\log(p+q)})} \cdot \frac{1}{x_1^p(1-x_1)^q} \cdot (p+q)^{\mathcal{O}(1)} \log n$ in time linear in the size of $\mathcal{F}$.
- Apply Lemma 1 for $0 < x_2 < 1$ and construct a n-p_1-p_2-separating collection $(\mathcal{H}, \chi_{\mathcal{H}}, \chi'_{\mathcal{H}})$ of size $2^{\mathcal{O}(\frac{p_1+p_2}{\log\log\log(p_1+p_2)})} \cdot \frac{1}{x_2^{p_1}(1-x_2)^{p_2}} \cdot (p_1+p_2)^{\mathcal{O}(1)} \log n$ in time linear in the size of $\mathcal{H}$.

Now we construct a graph $G = (V, E)$ where the vertex set V contains a vertex each for sets in $\mathcal{F} \uplus \mathcal{H} \uplus \mathcal{L}_1 \uplus \mathcal{L}_2$. For clarity of presentation we name the vertices by the corresponding set. Thus, the vertex set $V = \mathcal{F} \uplus \mathcal{H} \uplus \mathcal{L}_1 \uplus \mathcal{L}_2$. The edge set $E = E_1 \uplus E_2 \uplus E_3 \uplus E_4$, where each E_i for $i \in \{1,2,3,4\}$ is defined as follows (see Figure 1). $E_1 = \Big\{(A,F) \;\Big|\; A \in \mathcal{L}_1,\ F \in \chi_{\mathcal{F}}(A)\Big\}$, $E_2 = \Big\{(B,F) \;\Big|\; B \in \mathcal{L}_2,\ F \in \chi_{\mathcal{F}}(B)\Big\}$, $E_3 = \Big\{(A,H) \;\Big|\; A \in \mathcal{L}_1,\ H \in \chi_{\mathcal{H}}(A)\Big\}$ and $E_4 = \Big\{(B,F) \;\Big|\; B \in \mathcal{L}_2,\ F \in \chi'_{\mathcal{H}}(B)\Big\}$. Thus G is essentially a 4-partite graph.

Algorithm. The construction of $\widehat{\mathcal{L}}$ is as follows. For a set $F \in \mathcal{F}$, we call a pair of sets (A, B) *cyclic*, if $A \in \mathcal{L}_1,\ B \in \mathcal{L}_2$ and there exists $H \in \mathcal{H}$ such that $FAHB$ forms a cycle of length four in G. Let $\mathcal{J}(F)$ denote the family of cyclic pairs for a set $F \in \mathcal{F}$ and $w_F = \min_{(A,B)\in\mathcal{J}(F)} w(A) + w(B)$.

We obtain the family $\widehat{\mathcal{L}}$ by adding $A \cup B$ for every set $F \in \mathcal{F}$ such that $(A,B) \in \mathcal{J}(F)$ and $w(A) + w(B) = w_F$. Indeed, if the family $\mathcal{J}(F)$ is empty then we do



$\mathcal{F}$
F_1
F_r
$F_r \in \chi_{\mathcal{F}}(A_i)$
$F_r \in \chi_{\mathcal{F}}(B_j)$
A_1
A_i
B_1
B_j
$\mathcal{L}_1$
$\mathcal{L}_2$
$H_\ell \in \chi_{\mathcal{H}}(A_i)$
$H_\ell \in \chi'_{\mathcal{H}}(B_j)$
H_1
H_ℓ
$\mathcal{H}$

Fig. 1. Graph constructed from $\mathcal{L}_1, \mathcal{L}_2, \mathcal{F}$ and $\mathcal{H}$

not add any set to $\widehat{\mathcal{L}}$ corresponding to F. The procedure to find the smallest weight $A \cup B$ for any F is as follows. We first mark the vertices of $N_G(F)$ (the neighbors of F). Now we mark the neighbors of $\mathcal{P} = (N_G(F) \cap \mathcal{L}_1)$ in $\mathcal{H}$. For every marked vertex $H \in \mathcal{H}$, we associate a set A of minimum weight such that $A \in (\mathcal{P} \cap N_G(H))$. This can be done sequentially as follows. Let $\mathcal{P} = \{S_1, \ldots, S_\ell\}$. Now iteratively visit the neighbors of S_i in $\mathcal{H}$, $i \in [\ell]$, and for each vertex of $\mathcal{H}$ store the smallest weight vertex $S \in \mathcal{P}$ it has seen so far. After this we have a marked set of vertices in $\mathcal{H}$ such that with each marked vertex H in $\mathcal{H}$ we stored a smallest weight marked vertex in $\mathcal{L}_1$ which is a neighbor of H. Now for each marked vertex B in $\mathcal{L}_2$, we go through the neighbors of B in the marked set of vertices in $\mathcal{H}$ and associate (if possible) a second vertex (which is a minimum weighted marked neighbor from $\mathcal{L}_2$) with each marked vertex in $\mathcal{H}$. We obtain a pair of sets $(A, B) \in \mathcal{J}(F)$ such that $w(A) + w(B) = w_F$. This can be easily done by keeping a variable that stores a minimum weighted $A \cup B$ seen after every step of marking procedure. Since for each $F \in \mathcal{F}$ we add at most one set to $\widehat{\mathcal{L}}$, the size of $\widehat{\mathcal{L}}$ follows.

Correctness. We first show that $\widehat{\mathcal{L}} \subseteq \mathcal{L}$. Towards this we only need to show that for every $A \cup B \in \widehat{\mathcal{L}}$ we have that $A \cap B = \emptyset$. Observe that if $A \cup B \in \widehat{\mathcal{L}}$ then there exists a $F \in \mathcal{F}$, $H \in \mathcal{H}$ such that $FAHB$ forms a cycle of length four in the graph G. So $H \in \chi_{\mathcal{H}}(A)$ and $H \in \chi'_{\mathcal{H}}(B)$. This means $A \subseteq H$ and $B \cap H = \emptyset$. So we conclude A and B are disjoint and hence $\widehat{\mathcal{L}} \subseteq \mathcal{L}$. We also need to show that if there exist pairwise disjoint sets $A \in \mathcal{L}_1, B \in \mathcal{L}_2, C \in \binom{U}{q}$, then there exist $\hat{A} \in \mathcal{L}_1, \hat{B} \in \mathcal{L}_2$ such that $\hat{A} \cup \hat{B} \in \widehat{\mathcal{L}}$, $\hat{A}, \hat{B}, C$ are pairwise disjoint and $w(\hat{A}) + w(\hat{B}) \leq w(A) + w(B)$. By the property of separating collections $(\mathcal{F}, \chi_{\mathcal{F}}, \chi'_{\mathcal{F}})$ and $(\mathcal{H}, \chi_{\mathcal{H}}, \chi'_{\mathcal{H}})$, we know that there exists $F \in \chi_{\mathcal{F}}(A) \cap \chi_{\mathcal{F}}(B) \cap \chi'_{\mathcal{F}}(C)$, $H \in \chi_{\mathcal{H}}(A) \cap \chi'_{H}(B)$. This implies that $FAHB$ forms a cycle of length four in the graph G. Hence in the construction of $\widehat{\mathcal{L}}$, we should have chosen $\hat{A} \in \mathcal{L}_1$ and $\hat{B} \in \mathcal{L}_2$ corresponding to F such that $w(\hat{A}) + w(\hat{B}) \leq w(A) + w(B)$ and added to $\widehat{\mathcal{L}}$. So we know that $F \in \chi_{\mathcal{F}}(\hat{A}) \cap \chi_{\mathcal{F}}(\hat{B})$. Now we claim that $\hat{A}, \hat{B}$ and C are pairwise disjoint. Since $\hat{A} \cup \hat{B} \in \widehat{\mathcal{L}}$, $\hat{A} \cap \hat{B} = \emptyset$. Finally, since $F \in \chi_{\mathcal{F}}(\hat{A}) \cap \chi_{\mathcal{F}}(\hat{B})$ and $F \in \chi'_{\mathcal{F}}(C)$, we get $\hat{A}, \hat{B} \subseteq F$ and $F \cap C = \emptyset$ which implies C is disjoint from $\hat{A}$ and $\hat{B}$. This completes the correctness proof. The running time analysis is deferred to the full version of the paper □

For the product families of a linear matroid we have the following theorem.

Theorem 2 (⋆). *Let $M = (E, \mathcal{I})$ be a linear matroid of rank k, $\mathcal{L}_1$ be a p_1-family of independent sets of M and $\mathcal{L}_2$ be a p_2-family of independent sets of M. Given a representation A_M of M over a field $\mathbb{F}$, we can find $\widehat{\mathcal{L}_1 \bullet \mathcal{L}_2} \subseteq^{k-p_1-p_2}_{minrep} \mathcal{L}_1 \bullet \mathcal{L}_2$ of size at most $\binom{k}{p_1+p_2}$ in $\mathcal{O}\left(|\mathcal{L}_2||\mathcal{L}_1|\binom{k-p_2}{p_1}^{\omega-1} p_1^\omega + |\mathcal{L}_2|\binom{k-p_2}{p_1}\binom{k}{p_1+p_2}^{\omega-1}(p_1+p_2)^\omega\right)$ operations over $\mathbb{F}$.*

4 Applications

In this subsection we mention some of the applications of Theorems 1 and 2 and its derivatives.

Multilinear Monomial Testing. In this section we first design a faster algorithm for a weighted version of k-MLD and then give an algorithm for an extension of this to a matroidal version. In the weighted version of k-MLD in addition to an arithmetic circuit C over variables $X = \{x_1, x_2, \ldots, x_n\}$ representing a polynomial $P(X)$ over $\mathbb{Z}^+$, we are also given an additive weight function $w : 2^X \to \mathbb{N}$. The task is that if there exists a k-multilinear term then find one with minimum weight. We call the weighted variant by k-wMLD. We start with the definition of an arithmetic circuit.

Definition 4. *An* arithmetic circuit C *over a commutative ring R is a simple labelled directed acyclic graph with its internal nodes are labeled by $+$ or $\times$ and leaves (in-degree zero nodes) are labeled from $X \cup R$, where $X = \{x_1, x_2, \ldots, x_n\}$, a set of variables. There is a node of out-degree zero, called the root node or the output gate. The size of C, $s(C)$ is the number of vertices in the graph.*

It is well known that we can replace any arithmetic circuit C with an equivalent circuit with fan-in two for all the internal nodes with quadratic blow up in the size. For an example, by replacing each node of in-degree greater than 2, with at most $s(C)$ many nodes of the same label and in-degree 2, we can convert a circuit C to a circuit C' of size $s(C') = s(C)^2$. So from now onwards we always assume that we are given a circuit of this form. We assume W be the maximum weight defined by w.

Theorem 3 (⋆). k-wMLD *can be solved in time $\mathcal{O}(3.8408^k 2^{o(k)} s(C) n \log^2 n \cdot \log W)$.*

Proof (Sketch of the proof). An arithmetic circuit C over $\mathbb{Z}^+$ with all leaves labelled from $X \cup \mathbb{Z}^+$ will represent sum of monomials with positive integer coefficients. With each multilinear term $\Pi_{j=1}^{\ell} x_{i_j}$ we associate a set $\{x_{i_1}, \ldots, x_{i_l}\} \subseteq X$. With any polynomial we can associate a family of subsets of X which corresponds to the set of multilinear terms in it. Since C is a directed acyclic graph, there exists a topological ordering $\pi = v_1, \ldots, v_n$, such that all the nodes corresponding to variables appear before any other gate and for every directed arc uv we have that $u <_\pi v$. For a node v_i of the circuit let $P_i(X)$ be the multivariate polynomial represented by the subcircuit containing all the nodes w such that $w \leq_\pi v_i$. At every node we keep a family $\mathcal{F}^j_{v_i}$ of *j-multilinear term*, where $j \in \{1, \ldots, k\}$. Let $\mathcal{F}_{v_i} = \cup_{x=1}^k \mathcal{F}^x_{v_i}$. Given a circuit C, if we compute associated family of subsets of X for each node we can answer the question of having a k-multilinear term of minimum weight in the polynomial computed by C. But the size of the family of subsets could be exponential in n, the number of variables. That is, the size of $\mathcal{F}^j_{v_i}$ could be $\binom{n}{j}$. So instead of storing all subsets, we store a representative family for the associated family of subsets of each node. That is, we store $\widehat{\mathcal{F}^j_{v_i}} \subseteq^{k-j}_{minrep} \mathcal{F}^j_{v_i}$. The correctness of this step follows from the definition of $k-j$-representative family.

We make a dynamic programming algorithm to detect a multilinear monomial of order k as follows. Our algorithm goes from left to right following the ordering given by π and computes $\mathcal{F}_{v_i}$ from the families previously computed. The algorithm computes an appropriate representative family corresponding to each node of C. We show that we can compute a representative family $\mathcal{F}_v$ associated with any node v, where the number of subsets with p elements in $\mathcal{F}_v$ is at most $\binom{k}{p} 2^{o(k)} \log n$. Detailed algorithm and correctness proof will be available in the full version of the paper. $\square$

We also provide an algorithm for matroidal version of k-wMLD where we are additionally given a linear matroid on the "variable set" and the objective is to find a multilinear monomial that is also an independent set of the given matroid. More precisely we give the following theorem.

Theorem 4 ($\star$). k-wMMLD *can be solved in time* $\mathcal{O}(7.7703^k k^\omega s(C))$.

Improved Treewidth Algorithms. Using Theorem 2 we give the fastest known deterministic algorithms for "connectivity problems" such as Steiner Tree, Feedback Vertex Set parameterized by the treewidth of the input graph.

Theorem 5 ($\star$). *Let G be an n-vertex graph given together with its tree decomposition of with* **tw**. *Then* Steiner Tree *(or* Feedback Vertex Set*) on G can be solved in time* $\mathcal{O}\left(\left(1 + 2^{\omega-1} \cdot 3\right)^{\mathbf{tw}} \mathbf{tw}^{\mathcal{O}(1)} n\right)$.

Finally, we have.

Theorem 6 ($\star$). k-Path *admits an algorithm with running time* $\mathcal{O}(2.619^k n \log^2 n)$.

References

1. Alon, N., Yuster, R., Zwick, U.: Color-coding. J. Assoc. Comput. Mach. 42(4), 844–856 (1995)
2. Bellman, R., Karush, W.: Mathematical programming and the maximum transform. J. Soc. Indust. Appl. Math. 10, 550–567 (1962)

3. Bellman, R., Karush, W.: On the maximum transform and semigroup of transformations. Bull. Amer. Math. Soc. 68, 516–518 (1962)
4. Björklund, A., Husfeldt, T., Kaski, P., Koivisto, M.: Fourier meets Möbious: Fast subset convolution. In: Proceedings of the 39th Annual ACM Symposium on Theory of Computing (STOC 2007). ACM Press, New York (2007) (page to appear)
5. Björklund, A., Husfeldt, T., Kaski, P., Koivisto, M.: Narrow sieves for parameterized paths and packings. CoRR, abs/1007.1161 (2010)
6. Björklund, A., Kaski, P., Kowalik, L.: Probably optimal graph motifs. In: STACS. LIPIcs, vol. 20, pp. 20–31 (2013)
7. Bodlaender, H.L., Cygan, M., Kratsch, S., Nederlof, J.: Deterministic single exponential time algorithms for connectivity problems parameterized by treewidth. In: Fomin, F.V., Freivalds, R., Kwiatkowska, M., Peleg, D. (eds.) ICALP 2013, Part I. LNCS, vol. 7965, pp. 196–207. Springer, Heidelberg (2013)
8. Cygan, M., Nederlof, J., Pilipczuk, M., Pilipczuk, M., van Rooij, J.M.M., Wojtaszczyk, J.O.: Solving connectivity problems parameterized by treewidth in single exponential time. In: Proceedings of the 52nd Annual Symposium on Foundations of Computer Science (FOCS 2011). IEEE (2011)
9. Fomin, F.V., Lokshtanov, D., Raman, V., Saurabh, S., Rao, B.V.R.: Faster algorithms for finding and counting subgraphs. J. Comput. System Sci. 78(3), 698–706 (2012)
10. Fomin, F.V., Lokshtanov, D., Saurabh, S.: Efficient computation of representative sets with applications in parameterized and exact algorithms. In: SODA, pp. 142–151 (2014)
11. Guillemot, S., Sikora, F.: Finding and counting vertex-colored subtrees. Algorithmica 65(4), 828–844 (2013)
12. Koutis, I.: Faster algebraic algorithms for path and packing problems. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part I. LNCS, vol. 5125, pp. 575–586. Springer, Heidelberg (2008)
13. Koutis, I.: Constrained multilinear detection for faster functional motif discovery. Inf. Process. Lett. 112(22), 889–892 (2012)
14. Lovász, L.: Flats in matroids and geometric graphs. In: Combinatorial surveys (Proc. Sixth British Combinatorial Conf., Royal Holloway Coll., Egham), pp. 45–86. Academic Press, London (1977)
15. Marx, D.: Parameterized coloring problems on chordal graphs. Theor. Comput. Sci. 351(3), 407–424 (2006)
16. Marx, D.: A parameterized view on matroid optimization problems. Theor. Comput. Sci. 410(44), 4471–4479 (2009)
17. Monien, B.: How to find long paths efficiently. In: Analysis and design of algorithms for combinatorial problems, Udine, 1982. North-Holland Math. Stud., vol. 109, pp. 239–254. North-Holland, Amsterdam (1985)
18. Oxley, J.G.: Matroid theory, vol. 3. Oxford University Press (2006)
19. Shachnai, H., Zehavi, M.: Faster computation of representative families for uniform matroids with applications. CoRR, abs/1402.3547 (2014)
20. Williams, R.: Finding paths of length k in $O^*(2^k)$ time. Inf. Process. Lett. 109(6), 315–318 (2009)
21. Williams, V.V.: Multiplying matrices faster than Coppersmith-Winograd. In: Proceedings of the 44th Symposium on Theory of Computing Conference (STOC 2012), pp. 887–898. ACM (2012)

Weighted Ancestors in Suffix Trees

Paweł Gawrychowski[1], Moshe Lewenstein[2], and Patrick K. Nicholson[1]

[1] Max-Planck-Institut für Informatik, Saarbrücken, Germany
[2] Bar-Ilan University, Israel

Abstract. The classical, ubiquitous, *predecessor problem* is to construct a data structure for a set of integers that supports fast predecessor queries. Its generalisation to weighted trees, a.k.a. *the weighted ancestor problem*, has been extensively explored and successfully reduced to the predecessor problem. It is known that any data structure solution for the weighted ancestor problem that occupies $\mathcal{O}(n\ \mathrm{polylog}(n))$ space must have $\Omega(\log\log n)$ query time, if the weights are drawn from a polynomially bounded universe. Perhaps the most important and frequent application of the weighted ancestors problem is for suffix trees. It has been a long-standing open question whether the weighted ancestors problem has better bounds for suffix trees. We answer this question positively: we show that a suffix tree built for a text $w[1..n]$ can be preprocessed using $\mathcal{O}(n)$ extra space, so that queries can be answered in $\mathcal{O}(1)$ time. Thus we improve the running times of several applications. Our improvement is based on a number of data structure tools and a periodicity-based insight into the combinatorial structure of a suffix tree.

1 Introduction

The well-known and widely-used *predecessor problem* is to preprocess a set of integers so that the predecessor of a given number can be located. Tight tradeoffs between construction space and query times for such a data structure are known (see [12] for a survey). The predecessor problem was generalised to trees by Farach and Muthukrishnan [6]. It is called the *weighted ancestor problem* and is defined as follows. We are given a rooted tree in which every node v has an associated integer *weight* $w(v)$ as input. The weights satisfy the min-heap property, that is the weight of every node is larger than the weight of its parent (the tree does not need to be binary). The goal of the problem is to preprocess the tree so that the predecessor of a given number, among the weights of all the ancestor nodes of a given leaf, can be located. They [6] described a randomised data structure, which can be constructed in $\mathcal{O}(n)$ time and space given an n-node tree with weights from $[1, n]$. The query time is $\mathcal{O}(\log\log n)$, see [1] for a *deterministic* version of the structure with the same bounds.

In the simpler unweighted version of the problem, called the *level ancestor problem*, we must preprocess a tree on n nodes, so that we can retrieve the k-th ancestor of a given node efficiently. Berkman and Vishkin showed that such a query can be answered in $\mathcal{O}(1)$ time, using $\mathcal{O}(n)$ preprocessing time and space [3].

Later, a much simpler solution was discovered [2]. However, the solutions for the level ancestor strongly utilise the fact that the difference in "weight" between levels is one, so they give no insight into the weighted ancestors problem.

The application for which the *weighted* ancestor problem was initially introduced [6] is *substring hashing.* In substring hashing one wants to preprocess a given string $w[1..n]$, to allow the efficient computation of the hash $h(i, j)$ of any of its substrings $w[i..j]$. The hashing should be perfect, i.e., $h(i, j) = h(i', j')$ if and only if $w[i..j] = w[i'..j']$. In [6] the substring hashing problem was reduced to weighted ancestor queries on a suffix tree. Since for suffix trees the universe size is $\mathcal{O}(n)$, one can use a predecessor data structure such as a y-fast trie [13] to obtain $\mathcal{O}(n)$ preprocessing time and space so that any hash can be computed in $\mathcal{O}(\log\log n)$ time. Their solution also gives the same bounds for the weighted ancestor problem in *any* tree where the weights are polynomial in n.

In the context of suffix trees the weighted ancestors problem can also be viewed as preprocessing a suffix tree built for a string $w[1..n]$, so as to allow the retrieval of the (implicit or explicit) node corresponding to any substring $w[i..j]$, given i and j. There are numerous applications and we will mention a few later.

Kopelowitz and Lewenstein [10] generalised the weighted ancestor problem to the dynamic setting and showed how to support leaf insertions and edge splitting operations (required to maintain a suffix tree for a growing text). They also showed that, up to an additive $\mathcal{O}(\log^\star n)$ term, the static problem is as easy as predecessor search: if one can implement a linear space static predecessor structure with a query time of $\text{pred}(n, U)$, where $\text{pred}(n, U)$ is the predecessor query time for a set of n integers drawn from the universe $[1, U]$, then a weighted ancestor query can be answered in $\mathcal{O}(\text{pred}(n, U) + \log^\star n)$ time after linear preprocessing.

Since the weighted ancestor is a generalisation of the predecessor problem, it cannot have better time/space bounds than the predecessor problem. Hence, by the known bounds for the weighted ancestor problem in which the universe size is polynomially bounded in n, any weighted ancestor data structure of size $\mathcal{O}(n \text{ polylog}(n))$ must have query time of $\Omega(\log\log n)$. Furthermore, this lower bound holds *even when* the the node weights are bounded by n. Nevertheless, weighted ancestors on suffix trees are a special case of the general weighted ancestors problem. Hence, it is plausible that one can do better. This was indirectly expressed in [6] where the question was raised whether batched substring hashing can be sped up. This led to the challenge of solving weighted ancestors on suffix trees in $\mathcal{O}(n)$ preprocessing time and space and $\mathcal{O}(1)$ query time which has been an open question for a long time now.

Contribution. All our results hold in the word-RAM model with logarithmic word size. We show that, for weighted ancestor in suffix trees, it is possible to achieve $\mathcal{O}(1)$ deterministic worst-case query time using $\mathcal{O}(n)$ additional space. To simplify the presentation in this shortened version of the paper, we describe a simpler solution that occupies $\mathcal{O}(n \text{ polylog}(n))$ space, and allows $\mathcal{O}(1)$ time queries. In the full version of this paper [8] we present the details to reduce the space usage to $\mathcal{O}(n)$, as well as the omitted proofs.

To sidestep the lower bound for the weighted ancestor problem, we look deeper into the structure of a suffix tree, and apply a periodicity-based argument. This argument allows us to decompose the tree into sufficiently simple subtrees, which are then preprocessed separately. To preprocess the subtrees, we develop an efficient solution for a variant of the predecessor problem, in which we are given multiple correlated sets of integers. The correlation allows us to circumvent the predecessor lower bound, which would be relevant if we were to consider each of the sets separately. As our solution contains many details, we provide a high level overview in Section 3. This yields improved query times to several problems.

Substring Search. Preprocess the suffix tree built for $w[1..n]$ to answer substring search queries, i.e., given a pair of indices i, j return the locus of $w[i..j]$ in the suffix tree (the node at the end of the partial path denoting $w[i..j]$). This is solved by a weighted ancestor query on a suffix tree: go to the node representing $w[i..n]$ and answer the predecessor query of $j - i + 1$ (in this case we prefer the analogous successor query). Since the weighted ancestors takes $\mathcal{O}(n)$ preprocessing space and $\mathcal{O}(1)$ query time, substring search has the same bounds.

Substring Hashing. We define $h(i, j) = \langle \text{locus of } w[i..j], i - j + 1 \rangle$, where the locus of $w[i..j]$ is found by substring search. It is easy to verify that $h(i, j) = h(i', j')$ iff $w[i..j] = w[i'..j']$. Hence, substring hashing can be improved to $\mathcal{O}(n)$ preprocessing space and $\mathcal{O}(1)$ query time. Consequently, batched substring hashing is also optimal. By not insisting that we return the corresponding node of the suffix tree, one can achieve an optimal $\mathcal{O}(1)$ query after $\mathcal{O}(n)$ preprocessing with a simpler method [7]. Nevertheless, the number of bits in the answer in [7] is $3 \log n$ while in our solution it is optimal $2 \log n$, and in some applications we want to access the suffix tree node, as it provides more information.

Cross-Document Pattern Matching. Index a collection of documents, so that given a substring $w[i..j]$ of the k-th document, we can search for its appearances in the k'-th document. This problem was introduced by Kopelowitz *et al.* [9], who also considered some extensions. Their linear space solution uses a generalised suffix tree with weighted ancestor queries. With our result the query time becomes $\mathcal{O}(1)$. The improvement can be also embedded in the extensions.

Searching Substrings Internally in the Suffix Tree. Cole *et al.* [5], when proposing data structures for indexing a text $w[1..n]$ with k mismatches, k errors and k wildcards, suggested the LCP data structure. The LCP data structure has two variants: *rooted* and *unrooted*. The former preprocesses an arbitrary collection of suffixes of $w[1..n]$ in $\mathcal{O}(n)$ space and allows a search from the root of the compressed trie of these suffixes in $\mathcal{O}(\log \log n)$ time by using weighted ancestor queries on a careful decomposition of the compressed trie. The latter preprocesses such a collection in $\mathcal{O}(n \log n)$ space to allow a search from an arbitrary node with an even more detailed decomposition. Both have query time $\mathcal{O}(\log \log n)$ due to weighted ancestors. Alas, reducing this to $\mathcal{O}(1)$ is problematic since the compressed trie is not a suffix tree, so required properties are lost. Nevertheless, we can support $\mathcal{O}(1)$ time LCP queries on the original suffix tree.

Indexing with k Wildcards. Cole *et al.* [5] also implicitly describe a solution to indexing with k wildcards in $\mathcal{O}(n \log n)$ space, that supports queries in time $\mathcal{O}(m + \sigma^k \log\log n + occ)$, using the LCP data structures mentioned above. The space was improved to $\mathcal{O}(n)$ by Bille *et al.* [4]. Recently, in [11] the running time was improved to $\mathcal{O}(m + \sigma^k \sqrt{\log\log\log n} + occ)$. Now this can be further improved to $\mathcal{O}(m + \sigma^k + occ)$ with unrooted LCP queries on the suffix tree itself.

Fragmented Pattern Matching. The problem of *Substring Concatenation*, defined in [1], requires preprocessing a text $w[1..n]$ so that given i, j and i', j'we can return a substring of w which is the concatenation of $w[i..j]$ and $w[i'..j']$. [1] solved this by using a suffix tree, a reversed suffix tree, weighted ancestor queries on both and a node intersection data structure, all in $O(\log\log n)$ time. However, this can also be solved with a couple of LCP data structures, one rooted and one unrooted. Combined with our new result this achieves $\mathcal{O}(1)$ query time. The more general Fragmented Pattern Matching requires proprocessing a text $w[1..n]$ so that after receiving a collection of k substrings as pairs of indices, one can answer whether there exists a substring $w[i_1..j_1]w[i_2..j_2] \ldots w[i_k..j_k]$ within the text. By extending the result for substring concatenation this takes $\mathcal{O}(k)$ time.

Weighted Ancestors in Arbitrary Trees. In the process of solving our problem, we remove the additive $\mathcal{O}(\log^\star n)$ term from [10], improving the cost of weighted ancestor queries in *any tree* to $\mathcal{O}(\text{pred}(n, U))$ after linear preprocessing.

2 Preliminaries

A *suffix tree* of a string w, denoted $ST(w)$, is a compacted trie containing all suffixes of $w\$$, where $\$$ is a unique character not occurring in w. A *generalised suffix tree* of a collection of strings $w_1, w_2, \ldots, w_k$, denoted $GST(w_1, w_2, \ldots, w_k)$, is a compacted trie containing all suffixes of $w_1\$_1$, $w_2\$_2$, …, $w_k\$_k$, where each $\$_i$ is a unique character not occurring in any of the strings. We will often use $w_i[j..]$ to denote the suffix of w_i starting at the j-th character. In a compacted trie we define the *depth* of a node to be its number of explicit ancestors, and the *string depth* to be the length of the string it represents. In a (generalised) suffix tree we define the *suffix link* of a node representing the string as to be a pointer to the node representing s. Every explicit node v stores such a link $\text{sl}(v)$. If v is implicit, then $\text{sl}(v)$ is not stored, but we will use this notion in some proofs.

We want to preprocess a suffix tree built for a string $w[1..n]$, so that, later, we can quickly retrieve the node corresponding to any substring $w[i..j]$. If the node is explicit, then we simply return a pointer to it, and if it is implicit, then we return a pointer to the corresponding edge of the suffix tree. We call this special case of the weighted level ancestor problem *substring retrieval*.

We say that a natural number p is a period of string w if $w[i] = w[i + p]$ for every i such that both sides are defined. The smallest such p is called the period of w, and if the period is at most $|w|/2$ we call w periodic. Otherwise it is aperiodic. The well-known property of periods is that if p and q are both periods

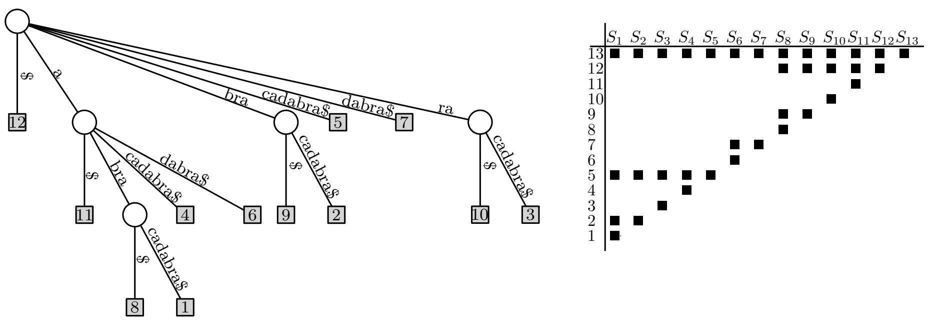

$
a
bra
cadabra$
dabra$
ra
12
11
4
6
9
2
5
7
10
3
8
1
S_1 S_2 S_3 S_4 S_5 S_6 S_7 S_8 S_9 S_{10} S_{11} S_{12} S_{13}
13 12 11 10 9 8 7 6 5 4 3 2 1

Fig. 1. A suffix tree built for `abracadabra` and the corresponding instance of PISNS

of w, and additionally $p + q \leq |w|$, then $\gcd(p, q)$ is a period of w, too. A *cyclic rotation* of a string $w[1..n]$ is a string $w[i+1..n]w[1..i]$. A *Lyndon word* has the property that it is lexicographically smallest among all its cyclic rotations. A string u is *primitive* if it cannot be represented as $u = v^i$ with $i > 1$.

All space bounds are measured in machine words, and all time bounds are deterministic worst-case. We make use of the solution to the level ancestor problem [2], so that we can retrieve a node of our compacted trie as soon as we know its depth.

3 Intuition and Overview

We start by presenting the intuition behind our solution and an overview of its formalisation, which is the main contribution of the paper.

Our goal is to preprocess the set of ancestors of every leaf. More precisely, if $D(v)$ is the set of string depths of all of the ancestors of v, we want to perform a predecessor search in any $D(v)$, where v is a leaf. We could preprocess every such $D(v)$ separately, but then the best query time that we can hope for is $\mathcal{O}(\log\log n)$, assuming that the allowed preprocessing space for every $D(v)$ is $\mathcal{O}(|D(v)| \text{ polylog}(|D(v)|))$ [12]. To overcome this, we observe that the sets corresponding to different leaves v are correlated. More precisely, if we consider two leaves v and $u = \text{sl}(v)$, then $x \in D(v)$ and $x > 0$ implies $x-1 \in D(u)$. If for every leaf corresponding to a suffix $w[i..n]$ we define a set $S_i = \{i + x : x \in D(v)\}$, then we get a collection of sets $S_1, S_2, \ldots, S_n \subseteq [1, N]$ such that $S_i \cap [n_{i+1}, N] \subseteq S_{i+1}$, where $N = n+1$ and $n_i = i$, see Figure 1. We call the problem of supporting predecessor queries on these types of sets *predecessor in shrinking nested sets*. In Section 4 we show that such collections can be processed using $\mathcal{O}(N \log^2 N + \sum_i |S_i|)$ space so that predecessor searching in any S_i takes just $\mathcal{O}(1)$ time (the space can be further improved). So, the correlation between different sets allows us to circumvent the known lower bound for near-linear space predecessor structures. Now if it were the case that $\mathcal{O}(\sum_i |S_i|) \in \mathcal{O}(n \text{ polylog}(n))$, we would be done.

Unfortunately, it can happen that a single explicit node contributes to multiple $D(v)$'s, hence the sum might be substantially larger. However, when we try to construct a string with such a large sum, it seems that the most natural candidates are very repetitive, for example $\mathtt{a}^{n-1}\mathtt{b}$. This is not a coincidence. If the same explicit node u contributes to two different sets $D(v)$ and $D(v')$, and the string depth of u is at least $\frac{3}{4}n$, then there are two different suffixes of the whole string such that their longest common prefix is of length at least $\frac{3}{4}n$. This means that the period of the *middle part* of the string, i.e., $w[\frac{1}{4}n..\frac{3}{4}n]$, is at most $\frac{1}{4}n$, or in other words the middle part is periodic. So the intuition is that the larger $\mathcal{O}(\sum_i |S_i|)$ is, the more periodic the string—or at least its middle part—is.

If the period of the whole string is p, then (by the periodicity lemma) any two suffixes $w[i..]$ and $w[j..]$ either branch out at string depth less than $2p$, or the shorter suffix is a prefix of the longer one and $j = i + \alpha p$. Moreover, any explicit node at string depth less than $2p$ is the lowest common ancestor of two leaves corresponding to suffixes of length less than $3p$. Hence the whole suffix tree can be decomposed into the top part, which is the suffix tree built for the length $3p$ suffix of w, and p long paths corresponding to the longer suffixes starting at different offsets modulo p (with one leaf attached to every explicit node on such path). On the i-th path, all explicit nodes are at string depths $i + \alpha p$, so it is trivial to answer a predecessor query in $\mathcal{O}(1)$ time there. If we additionally preprocess the top part, which is easy if p is small, we can answer any predecessor query. Hence the intuition is that the larger the sum becomes, the less interesting the tree is. Unfortunately, formalising this intuition is quite technical, as we need more control on how we measure the repetitiveness of our string.

To make the formalisation easier, in Section 5 we reduce the substring retrieval problem to a more structured variant. In *long substring retrieval* we must preprocess a generalised suffix tree T built for a collection of strings $w_1, w_2, \ldots, w_\beta$, where $\beta = \mathcal{O}(n/\ell)$, all of the same length ℓ, so that we can retrieve the node corresponding to any substring $w_k[i..j]$ of length at least $\frac{3}{4}\ell$. We call each w_i a *document*. All documents will be substrings of w, hence we specify them by giving their starting and ending positions. We show that if we can preprocess such a collection in $S(n)$ space achieving $\mathcal{O}(1)$ query time for the *long* substring retrieval problem, then we can solve the original substring retrieval in $\mathcal{O}((S(n) + n)\log n)$ space and the same query time. The idea is to decompose the string into fragments of length roughly 2^i for $i = 0, 1, \ldots, \log n$.

In Section 6 we solve long substring retrieval. We partition the substrings of length at least $\frac{3}{4}\ell$ of all documents into two types depending on whether their period is at least or at most $\frac{1}{4}\ell$. Informally, both types are easy to deal with, but for different reasons. Observe that if a substring of some w_i has period at most $\frac{1}{4}\ell$, then the middle part of w_i of length $\frac{1}{2}\ell$ is periodic. This allows us to quickly detect if the period of $w_i[j..k]$ that we query with is at least $\frac{1}{4}\ell$.

The simple case is when no w_i has a periodic middle part, i.e., all substrings of length at least $\frac{3}{4}\ell$ have periods at most $\frac{1}{4}\ell$. This implies that no w_i has two suffixes of length at least $\frac{3}{4}\ell$ such that their longest common prefix is of length at least $\frac{3}{4}\ell$. We define T' to be the *bottom part* of T consisting of all nodes at string depth at least $\frac{3}{4}\ell$. The number of leaves in any subtree of T' is exactly the number of *different* documents with suffixes in that subtree. Additionally, we partition the nodes of T' into *levels* according to the rounded logarithm of the number of documents in their subtree. Since this number is equal to the number of document ending in the subtree, the nodes at the same level constitute a collection of disjoint paths. Also, by looking at the suffix links we observe that the explicit nodes on these paths are, in a certain sense, nested. We exploit this nesting to retrieve the node lying on any of these paths in constant time. This is done by reducing the problem to predecessor in shrinking nested sets, allowing us to sidestep predecessor lower bounds.

In the general case some w_i might have a periodic middle part. Then T' is also the bottom part of T, but we additionally prune it to contain only the nodes such that their subtree does not contain the same document twice. We preprocess the pruned tree T' as in the simple case, which allows us to retrieve the node if the period of $w_i[j..k]$ is larger than $\frac{1}{4}\ell$. To process $w_i[j..k]$ with period at most $\frac{1}{4}\ell$, we group all substrings of length at least $\frac{3}{4}\ell$ with period at most $\frac{1}{4}\ell$ according to their periods. More precisely, for such a $w_i[j..k]$ with period p we find the (unique) Lyndon word r such that $|r| = p$ and $w_i[j..k]$ is a substring of r^∞. For every possible r we build a separate structure allowing us to locate the node corresponding to any $w_i[j..k]$ of length at least $\frac{3}{4}\ell$ being a substring of r^∞. The structure is again based on the observation that the explicit nodes in the corresponding part of T are in a certain sense nested.

4 Predecessor in Nested Sets

In this section we develop an efficient solution for a certain variant of the predecessor problem, where we want to preprocess a collection of sets of integers as to allow predecessor searching in any of them. By predecessor searching we mean returning the rank of the element which is the predecessor of a given value. We start with a version where the sets are $S_1, S_2, \ldots, S_k \subseteq [1, N]$ and $S_i \subseteq S_{i+1}$, which we call *predecessor in nested sets* or *PINS* .

Lemma 1. *PINS can be solved in $\mathcal{O}(N \log N + \sum_i |S_i|)$ space and $\mathcal{O}(1)$ time.*

Proof. We partition the collection of sets into $\log N$ groups. The k-th group contains all S_i with $|S_i| \in [2^k, 2^{k+1})$. Because $|S_i| \leq |S_{i+1}|$, we have that the k-th group contains the sets $S_{g_{k-1}+1}, \ldots, S_{g_k-1}, S_{g_k}$, where $0 = g_0 \leq g_1 \leq \ldots \leq g_{\log N}$. For every such group we allocate a table of length N, where we explicitly store the predecessor of every $x \in [1, N]$ in S_{g_k}. These tables allow us to locate the predecessor of any $x \in [1, N]$ in the last set of any group in $\mathcal{O}(1)$ time. Additionally, for every set S_i belonging to the k-th group we allocate a table of length $|S_{g_k}|$, where we store the predecessor of every $x \in S_{g_k}$ in S_i. To locate the predecessor of $x \in [1, N]$ in S_i, we first locate its predecessor y in S_{g_k}. Then we locate the predecessor of y in S_i. Both steps take $\mathcal{O}(1)$ time using the precomputed tables. Furthermore, the table allocated for every S_i is of length $|S_{g_k}| \leq 2|S_i|$, making the total space usage $\mathcal{O}(N \log N + \sum_i |S_i|)$. □

Now we discuss the more involved version of the problem, where we relax the requirement that $S_i \subseteq S_{i+1}$. In *predecessor in shrinking nested sets* (or *PISNS*)

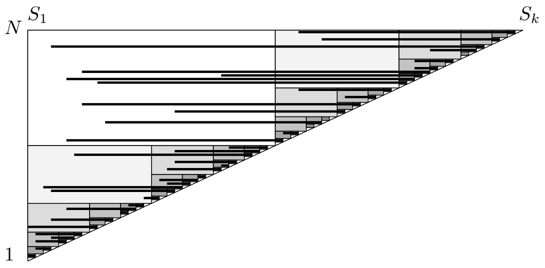


Fig. 2. An instance of PISNS. Each rectangle is an instance of PINS. Each horizontal line represents an element. If $x \in S_i$ then $x \in S_j$ as long as $j > i$ and $n_j \leq x$.

the sets have the additional property that one can choose $N = n_1 \geq n_2 \geq \ldots \geq n_k$ such that $S_i \subseteq [n_i, N]$ and $S_i \cap [n_{i+1}, N] \subseteq S_{i+1}$. We reduce this problem to a number of carefully chosen instances of PINS, as illustrated in Figure 2.

Lemma 2. *PISNS can be solved in $\mathcal{O}(N \log^2 N + \sum_i |S_i|)$ space and $\mathcal{O}(1)$ time.*

5 Reduction to Long Substring Retrieval

Lemma 3. *Suppose that any instance of long substring retrieval can be preprocessed using $S(n)$ space so that a query can be answered in $\mathcal{O}(1)$ time. Then, the general substring retrieval can be preprocessed using $\mathcal{O}((S(n)+n)\log n)$ space so that a query can be answered in $\mathcal{O}(1)$ time.*

Proof. (Sketch) To preprocess w for the general substring retrieval we construct a constant number of instances of long substring retrieval for each $k = 0, 1, \ldots, \log n$. For every such k, the instances roughly correspond to a decomposition of w into documents of length around $\ell = 2^k$. For every $k = 0, 1, \ldots, \log n$ we first split w into disjoint substrings of length 2^k, i.e., $b_1 = w[1..2^k], b_2 = w[2^k + 1..2^{k+1}], \ldots$, padding the last substring if necessary. Then for every $\alpha = 8, 9, \ldots, 15$ we create an instance of long substring retrieval with $\ell = \alpha 2^k$ by taking the documents to be of the form $w'_i = b_i b_{i+1} .. b_{i+\alpha-1}$ for $i = 1, 2, \ldots$, i.e., every possible contiguous sequence of α full blocks. Note that these documents are *not* disjoint substrings of w. There are $\mathcal{O}(n/\ell)$ such documents.

Now consider a query concerning a substring s. We want to select k and $\alpha \in \{8, 9, \ldots, 15\}$ such that $(\alpha - 2)2^k \leq |s| < (\alpha - 1)2^k$ and access the corresponding instance. This is always possible, as we can compute k such that $2^{k+3} \leq |s| < 2^{k+4}$, then $|s| - 2^{k+3} < 2^{k+3}$, so we can choose $\alpha' < 8$ such that $\alpha' 2^k \leq |s| - 2^{k+3} < (\alpha' + 1)2^k$, and finally take $\alpha = 8 + \alpha'$. Let b_i be the block where s starts, then s is fully within $b_i b_{i+1} .. b_{i+\alpha-1}$, so we can query the instance with the substring of w'_i-th document equal to s. For the answer to be correct, we must guarantee that $|s| \geq \frac{3}{4}\alpha 2^k$, but this follows from $\alpha \geq 8$. Hence using long substring retrieval we get the node v corresponding to s in the the generalised suffix tree built for all w'_i. $\square$

6 Solving Long Substring Retrieval

Recall that the goal in long substring retrieval is to preprocess a generalised suffix tree built for documents $w_1, w_2, \ldots, w_\beta$, where $\beta = \mathcal{O}(n/\ell)$ and $|w_i| = \ell$, as to retrieve the node corresponding to any $w_k[i..j]$ of length at least $\frac{3}{4}\ell$.

6.1 Handling Active Nodes

Let T be the generalised suffix tree built for $w_1, w_2, \ldots, w_\beta$, where $\beta = \mathcal{O}(n/\ell)$. While the goal is to preprocess the whole bottom part of T, i.e., all nodes at string depth at least $\frac{3}{4}\ell$, we will first show how to preprocess just some of these nodes. A node of T is *active* if its string depth is at least $\frac{3}{4}\ell$ and additionally there are no two different leaves corresponding to the suffixes of the same document in its subtree. Notice that if v is not active, neither is its parent, hence we can find a collection of nodes $v_1, v_2, \ldots, v_s$ such that a node is active iff it is a (not necessarily proper) descendant of some v_i. The active part of T, i.e., the forest consisting of all subtrees rooted at $v_1, v_2, \ldots, v_s$, will be called T'.

Lemma 4. *If a non-root node v is not active, then* $\mathrm{sl}(v)$ *is not active either.*

We will preprocess T so that we can retrieve the node corresponding to a substring $w_k[i..j]$ assuming that it is active. First we observe that it is not difficult to detect that the corresponding node is not active: for every leaf of T we can compute and store the string depth of its active ancestor that has the smallest string depth. Then we can take the leaf corresponding to $w_k[i..]$ and check if it has an active ancestor with a sufficiently large string depth.

We partition T' into disjoint paths using a variant of the centroid path decomposition. First define the *level* of a node $v \in T$ to be the unique integer k such that the number of leaves in the subtree of v belongs to $[2^k, 2^{k+1})$. From the definition, the level of any ancestor of v is at least as large as the level of v, and any node has at most one child at the same level. We also need the following properties of the levels, which are specific to the tree T. While we can afford to store the level only at the explicit nodes, all properties hold also for implicit nodes, and clearly the level of an implicit node can be determined by looking at its first explicit descendant. Based on these definitions, we prove the following.

Lemma 5. *The level of* $\mathrm{sl}(v)$ *is at least as large as the level of v.*

Lemma 6. *Suppose u and v are two nodes at the same level, such that u is neither an ancestor or descendant of v. If* $\mathrm{sl}(u)$ *is an ancestor of* $\mathrm{sl}(v)$*, then its level is larger than the levels of u and v.*

From now on we focus on a fixed level k. Since no node has two children at the same level, the active nodes at level k create a set of disjoint paths, $p_1, p_2, \ldots, p_s$, such that no node in p_i is an ancestor of a node in p_j if $i \neq j$. Every such path starts at an explicit node which has no child at level k and continues up, terminating either just before another explicit node at a level larger than k or an implicit node at string depth exactly $\frac{3}{4}\ell$. We say that path p_i *points to path* p_j, denoted $p_i \rightarrow p_j$, if there is a node $u \in p_i$ and a node $v \in p_j$ such that $\mathrm{sl}(u) = v$. This is a valid definition, and furthermore any path is pointed to by at most one other path, as shown in the following lemma.

Lemma 7. *Relation $\rightarrow$ has the following properties (a) if $p_i \rightarrow p_j$ then $i \neq j$, (b) if $p_i \rightarrow p_j$ and $p_i \rightarrow p_{j'}$ then $j = j'$, (c) if $p_i \rightarrow p_j$ and $p_{i'} \rightarrow p_j$ then $i = i'$.*

Hence we can partition the whole set of paths of active nodes at level k into:

1. *cycles of paths*, which are of the form $p_{i_1} \rightarrow p_{i_2} \rightarrow \ldots \rightarrow p_{i_z} \rightarrow p_{i_1}$, $z \geq 2$;
2. *chains of paths*, which are of the form $p_{i_1} \rightarrow p_{i_2} \rightarrow \ldots \rightarrow p_{i_z}$, where p_{i_z} doesn't point to any path and no path points to p_{i_1}.

We will preprocess every such cycle and chain separately using the solution for predecessor in shrinking nested sets from the previous section, which allow us to answer a predecessor query on any path in $\mathcal{O}(1)$ time, and bound the total space used by all the instances of the solution.

Consider a single cycle or chain of paths, where for a cycle of paths we additionally define $i_{z+1} = i_1$. If $v \in p_{i_j}$ is an explicit node, then $\mathrm{sl}(v)$ is either an explicit node on $p_{i_{j+1}}$, or its level is larger. Hence the sets of explicit nodes on subsequent paths are, in a certain sense, nested. To formalise this intuition, for every path p_{i_j} we denote the smallest and largest string depth of an (implicit or explicit) node by ℓ_j and r_j, respectively. Then the range of this path is an interval $U_j = [j + \ell_j, j + r_j]$. Furthermore, we construct a set $S_j \subseteq U_j$ corresponding to the path by including the depth of every explicit node (increased by j for technical reasons). Now the ranges and the sets are nested in the following sense.

Lemma 8. *The following properties of U_j and S_j hold (a) $j + \ell_j \leq j + 1 + \ell_{j+1}$, (b) $j + r_j \leq j + 1 + r_{j+1}$, (c) $S_j \cap U_{j+1} \subseteq S_{j+1}$.*

To execute a predecessor query on p_{i_j}, it is enough to perform such a query on the corresponding set S_j, so we focus on preprocessing all these sets. It is clear that their total size is small, as every element of S_j corresponds to a different explicit node of T', but this is not enough to beat the $\mathcal{O}(\log\log n)$ bound on the query time. We need an insight into the structure of all S_j based on Lemma 8.

Suppose we extend every range to the right by defining $U'_j = [j + \ell_j, z + r_z]$. Then it still holds that $S_j \cap U'_{j+1} \subseteq S_{j+1}$, but additionally all U'_j end with the same number. We will preprocess all U'_j using the data structure of Lemma 2. Its space usage depends on the total size of all S_j, which as already observed is small, but also on the size of the largest extended range U'_1. Even though a single $|U'_1|$ might be big, the sum of all such values over all cycles and chains of paths is at most $n/2^k$ by the following lemmas based on charging arguments. We define the *cost* $\mathrm{c}(p_{i_j})$ of a path p_{i_j} as follows: (1) $\mathrm{c}(p_{i_j}) = r_j - r_{j-1} + 1$ if $j > 1$; (2) for a cycle of paths $\mathrm{c}(p_{i_j}) = r_1 - r_z + 1$ if $j = 1$; (3) for a chain of paths $\mathrm{c}(p_{i_j}) = r_1 - \ell_1 + 1$ if $j = 1$. Note that for a cycle of paths we arbitrarily fix one of the paths to be p_{i_1}. The following two lemmas bound the costs of individual chains (or cycles) of paths, and the cost of all paths at level k, respectively.

Lemma 9. *For any chain of paths we have that* $|U'_1| = \sum_j \mathrm{c}(p_{i_j})$*, and for any cycle of paths* $|U'_1| \leq 2\sum_j \mathrm{c}(p_{i_j})$*.*

Lemma 10. *The sum of costs of all paths at level* k *is at most* $3n/2^k$*.*

To locate the node corresponding to $w_k[i..j]$, we first retrieve the leaf of T corresponding to the whole $w_k[i..]$. Then we must compute the level of the node corresponding to $w_k[i..j]$. More precisely, we must find an ancestor u of v at level k such that the string depth of u is at least $|w_k[i..j]|$, and furthermore the level of the node corresponding to $w_k[i..j]$ is the same as the level of u. This is enough to reduce the query to a weighted predecessor search on a single path in one of our collections. Computing u can be done in $\mathcal{O}(1)$ using the following lemma, which also removes the $\mathcal{O}(\log^* n)$ additive term from the query complexity of [10].

Lemma 11. *A weighted tree on* n *nodes, where some of the nodes are marked, but any path from a leaf to the root contains at most* $\mathcal{O}(\log n)$ *marked nodes, can be preprocessed in* $\mathcal{O}(n)$ *space so that predecessor search can be performed among the marked ancestors of any node in* $\mathcal{O}(1)$ *time.*

To apply the above lemma, we mark the explicit nodes of T such that the level of their parent is strictly larger. As the maximum level is $\log n$, the maximum number of marked nodes on any path from the leaf is also $\log n$. Hence we have reduced the query to performing a predecessor search among all ancestors on the same level of an explicit node u. At every explicit node we store a pointer to its path, and for every path we store a pointer to its cycle or chain of paths.

6.2 Handling the Remaining Nodes

The method from last subsection allows us to retrieve the node v corresponding to $w_i[j..k]$ if it belongs to T', or detect that we need to look at the non-active part. If v does not belong to T', even though $|w_i[j..k]| \geq \frac{3}{4}\ell$, then its subtree contains two different leaves originating from the same document. But then these leaves correspond to some $w_{i'}[j'..]$ and $w_{i'}[j''..]$ with $j' \neq j''$, and furthermore $w_i[j..k]$ is a prefix of both these suffixes. It follows that the period of $w_i[j..k]$ is at most $\frac{1}{4}\ell$. We preprocess all such $w_i[j..k]$ separately.

As discussed in Section 3, if the period of $w_i[j..k]$ of length at least $\frac{3}{4}\ell$ is at most $\frac{1}{4}\ell$, then the middle part of w_i, namely $w_i[\frac{1}{4}\ell..\frac{3}{4}\ell]$, is periodic. For every w_i we compute the period p of its middle part, and if $p \leq \frac{1}{4}\ell$ we also find the lexicographically smallest cyclic rotation of the corresponding string r of length p such that the middle part is a substring of r^∞. We group together all w_i with the same r and preprocess the subtree of T corresponding to their substrings fully contained in the periodic part separately.

For a string r, let T_r be the subtree of T corresponding to all substrings of r^∞ of length at least $\frac{1}{2}\ell$. First we show that any such T_r can be efficiently preprocessed for weighted level ancestor queries. In this case the input to a query is a substring of r^∞ specified by its length and starting position. Without loss of generality the starting position is less than $|r|$.

Lemma 12. *Let r be any primitive string of length at most $\frac{1}{4}\ell$, and s be the number of explicit nodes in T_r at string depth at least $\frac{1}{2}\ell$. T_r can be preprocessed using $\mathcal{O}(|r|\log|r| + s)$ space, so that, in $\mathcal{O}(1)$ time, the node corresponding to any substring of r^∞ of length at least $\frac{3}{4}\ell$ can be retrieved.*

Now if r and r' are two different Lyndon words of length at most $\frac{1}{4}\ell$, the sets of explicit nodes in T_r and $T_{r'}$ at string depth at least $\frac{1}{2}\ell$ are disjoint, as otherwise from the periodicity lemma we would get that r and r' are cyclic shifts of the same string. Hence if we apply the above lemma for every different Lyndon word r such that some w_i has the middle part which is a substring of r^∞, all explicit nodes contributing to the s added in the space complexity will sum up to n. Also, all $|r|$ will sum up to at most $\sum_i \frac{1}{4}|w_i| = \mathcal{O}(n)$, making the total space complexity $\mathcal{O}(n\log n)$.

References

1. Amir, A., Landau, G.M., Lewenstein, M., Sokol, D.: Dynamic text and static pattern matching. ACM Transactions on Algorithms 3(2) (2007)
2. Bender, M.A., Farach-Colton, M.: The level ancestor problem simplified. Theor. Comput. Sci. 321(1), 5–12 (2004)
3. Berkman, O., Vishkin, U.: Finding level-ancestors in trees. J. Comput. Syst. Sci. 48(2), 214–230 (1994)
4. Bille, P., Gørtz, I.L., Vildhøj, H.W., Vind, S.: String indexing for patterns with wildcards. In: Fomin, F.V., Kaski, P. (eds.) SWAT 2012. LNCS, vol. 7357, pp. 283–294. Springer, Heidelberg (2012)
5. Cole, R., Gottlieb, L.A., Lewenstein, M.: Dictionary matching and indexing with errors and don't cares. In: STOC, pp. 91–100 (2004)
6. Farach, M., Muthukrishnan, S.: Perfect hashing for strings: Formalization and algorithms. In: Hirschberg, D.S., Meyers, G. (eds.) CPM 1996. LNCS, vol. 1075, pp. 130–140. Springer, Heidelberg (1996)
7. Gawrychowski, P.: Pattern Matching in Lempel-Ziv Compressed Strings: Fast, Simple, and Deterministic. In: Demetrescu, C., Halldórsson, M.M. (eds.) ESA 2011. LNCS, vol. 6942, pp. 421–432. Springer, Heidelberg (2011)
8. Gawrychowski, P., Lewenstein, M., Nicholson, P.K.: Weighted ancestors in suffix trees. CoRR abs/1406.7716 (2014)
9. Kopelowitz, T., Kucherov, G., Nekrich, Y., Starikovskaya, T.A.: Cross-document pattern matching. J. Discrete Algorithms 24, 40–47 (2014)
10. Kopelowitz, T., Lewenstein, M.: Dynamic weighted ancestors. In: SODA, pp. 565–574 (2007)
11. Lewenstein, M., Nekrich, Y., Vitter, J.S.: Space-efficient string indexing for wildcard pattern matching. In: STACS, pp. 506–517 (2014)
12. Pătraşcu, M.: Predecessor search. In: Encyclopedia of Algorithms (2008)
13. Willard, D.E.: Log-logarithmic worst-case range queries are possible in space $\theta(n)$. Inf. Process. Lett. 17(2), 81–84 (1983)

Improved Practical Matrix Sketching with Guarantees

Mina Ghashami, Amey Desai, and Jeff M. Phillips

University of Utah, Utah, USA

Abstract. Matrices have become essential data representations for many large-scale problems in data analytics, and hence matrix sketching is a critical task. Although much research has focused on improving the error/size tradeoff under various sketching paradigms, we find a simple heuristic iSVD, with no guarantees, tends to outperform all known approaches. In this paper we adapt the best performing guaranteed algorithm, FrequentDirections, in a way that preserves the guarantees, and nearly matches iSVD in practice. We also demonstrate an adversarial dataset for which iSVD performs quite poorly, but our new technique has almost no error. Finally, we provide easy replication of our studies on APT, a new testbed which makes available not only code and datasets, but also a computing platform with fixed environmental settings.

1 Introduction

Matrix sketching has become a central challenge [3, 15, 18, 26, 31] in large-scale data analysis as many large data sets including customer recommendations, image databases, social graphs, document feature vectors can be modeled as a matrix, and sketching is either a necessary first step in data reduction or has direct relationships to core techniques including PCA, LDA, and clustering.

There are several variants of this problem, but in general the goal it to process an $n \times d$ matrix A to somehow represent a matrix B so $\|A - B\|_F$ or (examining the covariance) $\|A^T A - B^T B\|_2$ is small.

In both cases, the best rank-k approximation A_k can be computed using the singular value decomposition (svd); however this takes $O(nd \min(n, d))$ time and $O(nd)$ memory. This is prohibitive for modern applications which usually desire a small space streaming approach, or even an approach that works in parallel. For instance diverse applications receive data in a potentially unbounded and time-varying stream and want to maintain some sketch B. Examples of these applications include data feeds from sensor networks [6], financial tickers [9,39], on-line auctions [5], network traffic [21,36], and telecom call records [12].

In recent years, extensive work has taken place to improve theoretical bounds in B. Random projection [3, 35] and hashing [10, 38] approximate A in B as a random linear combination of rows and/or columns of A. Column sampling methods [7,14–17,28,34] choose a set of columns (and/or rows) from A to represent B; the best bounds require multiple passes over the data. We refer readers to recent work [10, 20] for extensive discussion of various models and error bounds.

A.S. Schulz and D. Wagner (Eds.): ESA 2014, LNCS 8737, pp. 467–479, 2014.

Very recently Liberty [26] introduced a new technique FrequentDirections (abbreviated FD) which is deterministic, achieves the best bounds on the covariance $\|A^TA - B^TB\|_2$ error, the direct error $\|A - B\|_F^2$ [20] (using B as a projection), and moreover, it seems to greatly outperform the projection, hashing, and column sampling techniques in practice.

However, there is a family of heuristic techniques [8, 22, 23, 25, 33] (which we refer to as iSVD, described relative to FD in Section 2), which are used in many practical settings, but are not known to have any error guarantees. In fact, we observe (see Section 3) on many real and synthetic data sets that iSVD noticeably outperforms FD, yet there are adversarial examples where it fails dramatically.

Thus in this paper we ask (and answer in the affirmative), *can one achieve a matrix sketching algorithm that matches the usual-case performance of iSVD, and the adversarial-case performance of FD, and error guarantees of FD?*

1.1 Notation and Problem Formalization

We denote an $n \times d$ matrix A as a set of n rows as $[a_1; a_2; \ldots, a_n]$ where each a_i is a row of length d. Alternatively a matrix V can be written as a set of columns $[v_1, v_2, \ldots, v_d]$. We assume $d \ll n$. We will consider streaming algorithms where each element of the stream is a row a_i of A.

The squared Frobenius norm of a matrix A is defined $\|A\|_F^2 = \sum_{i=1} \|a_i\|^2$ where $\|a_i\|$ is Euclidean norm of row a_i, and it intuitively represents the total size of A. The spectral norm $\|A\|_2 = \max_{x:\|x\|=1} \|Ax\|$, and represents the maximum influence along any unit direction x. It follows that $\|A^TA - B^TB\|_2 = \max_{x:\|x\|=1} |\|Ax\|^2 - \|Bx\|^2|$.

Given a matrix A and a low-rank matrix X let $\pi_X(A) = AX^T(XX^T)^+X$ be a *projection* operation of A onto the rowspace spanned by X; that is if X is rank r, then it projects to the r-dimensional subspace of points (e.g. rows) in X. Here X^+ indicates taking the Moore-Penrose pseudoinverse of X.

The singular value decomposition of A, written $\mathsf{svd}(A)$, produces three matrices $[U, S, V]$ so that $A = USV^T$. Matrix U is $n \times n$ and orthogonal. Matrix V is $d \times d$ and orthogonal; its columns $[v_1, v_2, \ldots, v_d]$ are the right singular vectors, describing directions of most covariance in A^TA. S is $n \times d$ and is all 0s except for the diagonal entries $\{\sigma_1, \sigma_2, \ldots, \sigma_r\}$, the *singular values*, where $r \leq d$ is the rank. Note that $\sigma_j \geq \sigma_{j+1}$, $\|A\|_2 = \sigma_1$, and $\sigma_j = \|Av_j\|$ describes the norm along directions v_j.

Frequent Directions Bounds. We describe FD in detail in Section 2, here we state the error bounds precisely. From personal communication [19], in a forth-coming extension of the analysis by Liberty [26] and Ghashami and Phillips [20], it is shown that there exists a value Δ that FD, run with parameter ℓ, satisfies three facts (for $\alpha = 1$):

- Fact 1: For any unit vector x we have $\|Ax\|^2 - \|Bx\|^2 \geq 0$.
- Fact 2: For any unit vector x we have $\|Ax\|^2 - \|Bx\|^2 \leq \Delta$.
- Fact 3: $\|A\|_F^2 - \|B\|_F^2 \geq \alpha\Delta\ell$.

Their analysis shows that *any* algorithm that follows these facts (for any $\alpha \in (0,1]$, and any $k \leq \alpha\ell$ including $k = 0$ where A_0 is the all zeros matrix) satisfies $\varDelta \leq \|A - A_k\|_F^2/(\alpha\ell - k)$; hence for any unit vector x we have $0 \leq \|Ax\|^2 - \|Bx\|^2 \leq \varDelta \leq \|A - A_k\|_F^2/(\alpha\ell - k)$. Furthermore *any* such algorithm also satisfies $\|A - \pi_{B_k}(A)\| \leq \alpha\ell\varDelta \leq \|A - A_k\|_F^2 \alpha\ell/(\alpha\ell - k)$, where $\pi_{B_k}(\cdot)$ projections onto B_k, the top k singular vectors of B. We reprove these results in the full version.

FD maintains an $\ell \times d$ matrix B (i.e. using $O(\ell d)$ space), with $\alpha = 1$. Thus setting $\ell = k + 1/\varepsilon$ achieves $\|A^T A - B^T B\|_2 \leq \varepsilon\|A - A_k\|_F^2$, and setting $\ell = k + k/\varepsilon$ achieves $\|A - \pi_{B_k}(A)\|_F^2 \leq (1+\varepsilon)\|A - A_k\|_F^2$.

1.2 Frequency Approximation, Intuition, and Results

FD is inspired by an algorithm by Misra and Gries [30] for the streaming frequent items problem. That is, given a stream $S = \langle s_1, s_2, \ldots, s_n \rangle$ of items $s_i \in [u] = \{1, 2, \ldots, u\}$, represent $f_j = |\{s_i \in S \mid s_i = j\}|$; the frequency of each item $j \in [u]$. The MG sketch uses $O(1/\varepsilon)$ space to construct an estimate $\hat{f}_j$ (for *all* $j \in [u]$) so that $0 \leq f_j - \hat{f}_j \leq \varepsilon n$. In brief it keeps $\ell - 1 = 1/\varepsilon$ counters, each labeled by some $j \in [u]$: it increments a counter if the new item matches the associated label or for an empty counter, and it decrements all counters if there is no empty counter and none match the stream element. $\hat{f}_j$ is the associated counter value for j, or 0 if there is no associated counter.

Intuitively (as we will see in Section 2), FD works similarly treating the singular vectors of B as labels and the squared singular values as counters.

This MG algorithm and variants have been rediscovered several times [13, 24, 29] and can be shown to process elements in $O(1)$ time. In particular, the SpaceSaving (SS) algorithm [29] has similar guarantees ($0 \leq \hat{f}_j - f_j \leq \varepsilon n$ for all $j \in [u]$) and also uses ℓ counters/labels. But when no counter is available, it does the unintuitive step of replacing the label of the least counter with the current stream element, and incrementing it by one. $\hat{f}_j$ is the associated counter, or otherwise the value of the minimum counter. A masterful empirical study by Cormode and Hadjieleftheriou [11] demonstrates that the SpaceSaving approach can outperform the standard MG step, and Agarwal *et al.* [4] shows that one can isomorphically convert between them by adding the number of decrements to each estimate from the MG sketch.

Main Results. To improve on FD empirically, it is natural to ask if variants mimicking SpaceSaving can be applied. We present two approaches SpaceSaving Directions (abbreviated SSD, which directly mimics the SpaceSaving algorithm) and Compensative FrequentDirections (abbreviated CFD, which mimics the conversion from MG to SS sketch). These are not isomorphic in the matrix setting as is the case in the items setting, but we are able to show strong error guarantees for each, asymptotically equivalent to FD. However, while these sometimes improve empirically on FD, they do not match iSVD.

Rather, we achieve our ultimate goal with another approach Parametrized FrequentDirections; it has parameter α and is abbreviated α-FD. It smoothly translates between FD and iSVD (FD = 1-FD and iSVD = 0-FD), and for $\alpha = 0.2$

we empirically demonstrate that it nearly matches the performance of iSVD on several synthetic and real data sets. It also has the same asymptotic guarantees as FD. Furthermore, we construct an adversarial data set where iSVD performs dramatically worse than FD and all proposed algorithms, including α-FD.

Finally, to ensure our results are easily and readily *reproducible*, we implement all experiments on a new extension of Emulab [37] called APT [32]. It allows one to check out a virtual machine with the same specs as we run our experiments, load our precise environments and code and data sets, and directly reproduce all experiments.

2 Algorithms

The main structure of the algorithm we will study is presented in Algorithm 2.1, where $S' \leftarrow$ ReduceRank(S) is a subroutine that differs for each variant we consider. It sets at least one non-zero in S to 0 in S'; this leads to a reduced rank for B_i, in particular with one row as all 0s. Notationally we use σ_j as the jth singular value in S, and σ'_j as the jth singular value in S'.

Algorithm 2.1. (Generic) FD Algorithm

```
**Input:** $\ell, \alpha \in (0,1], A \in \mathbb{R}^{n \times d}$
$B_0 \leftarrow$ all zeros matrix $\in \mathbb{R}^{\ell \times d}$
**for** $i \in [n]$ **do**
    Insert $a_i$ into a zero valued rows of $B_{i-1}$; result is $B_i$
    **if** ($B_i$ has no zero valued rows) **then**
        $[U, S, V] \leftarrow \mathsf{svd}(B_i)$
        $C_i = SV^T$                                # Only needed for proof notation
        $S' \leftarrow$ ReduceRank$(S)$
        $B_i \leftarrow S'V^T$
**return** $B = B_n$
```

For FD, ReduceRank sets each $\sigma'_j = \sqrt{\sigma_j^2 - \delta_i}$ where $\delta_i = \sigma_\ell^2$.

For iSVD, ReduceRank keeps $\sigma'_j = \sigma_j$ for $j < \ell$ and sets $\sigma'_\ell = 0$.

The runtime of FD can be improved [26] by doubling the space, and batching the svd call. A similar approach is possible for variants we consider.

2.1 Parameterized FD

Parameterized FD uses the following subroutine (Algorithm 2.2) to reduce the rank of the sketch; it zeros out row ℓ. This method has an extra parameter $\alpha \in [0,1]$ that describes the fraction of singular values which will get affected in the ReduceRank subroutine. Note iSVD has $\alpha = 0$ and FD has $\alpha = 1$. The intuition is that the smaller singular values are more likely associated with noise terms and the larger ones with signals, so we should avoid altering the signal terms in the ReduceRank step.

Here we show error bounds asymptotically matching FD for α-FD (for constant $\alpha > 0$), by showing the three Facts hold. We use $\Delta = \sum_{i=1}^{n} \delta_i$.

Algorithm 2.2. REDUCERANK-PFD(S, α)

```
$\delta_i \leftarrow \sigma_\ell^2$
return $\mathsf{diag}(\sigma_1, \ldots, \sigma_{\ell(1-\alpha)}, \sqrt{\sigma^2_{\ell(1-\alpha)+1} - \delta_i}, \ldots, \sqrt{\sigma_\ell^2 - \delta_i})$
```

Lemma 1. *For any unit vector x and any $\alpha \geq 0$: $0 \leq \|C_i x\|^2 - \|B_i x\|^2 \leq \delta_i$.*

Proof. The right hand side is shown by just expanding $\|C_i x\|^2 - \|B_i x\|^2$.

$$\|C_i x\|^2 - \|B_i x\|^2 = \sum_{j=1}^{\ell} \sigma_j^2 \langle v_j, x\rangle^2 - \sum_{j=1}^{\ell} {\sigma'}_j^2 \langle v_j, x\rangle^2 = \sum_{j=1}^{\ell} (\sigma_j^2 - {\sigma'}_j^2)\langle v_j, x\rangle^2$$

$$= \delta_i \sum_{j=(1-\alpha)\ell+1}^{\ell} \langle v_j, x\rangle^2 \leq \delta_i \|x\|^2 = \delta_i$$

To see the left side of the inequality $\delta_i \sum_{j=(1-\alpha)\ell+1}^{\ell} \langle v_j, x\rangle^2 \geq 0$. □

Then summing over all steps of the algorithm (using $\|a_i x\|^2 = \|C_i x\|^2 - \|B_{i-1} x\|^2$) it follows (see Lemma 2.3 in [20]) that

$$0 \leq \|Ax\|^2 - \|Bx\|^2 \leq \sum_{i=1}^{n} \delta_i = \varDelta,$$

proving Fact 1 and Fact 2 about α-FD for any $\alpha \in [0, 1]$.

Lemma 2. *For any $\alpha \in (0, 1]$, $\|A\|_F^2 - \|B\|_F^2 = \alpha \varDelta \ell$, proving Fact 3.*

Proof. We expand that $\|C_i\|_F^2 = \sum_{j=1}^{\ell} \sigma_j^2$ to get

$$\|C_i\|_F^2 = \sum_{j=1}^{(1-\alpha)\ell} \sigma_j^2 + \sum_{j=(1-\alpha)\ell+1}^{\ell} \sigma_j^2$$
$$= \sum_{j=1}^{(1-\alpha)\ell} {\sigma'}_j^2 + \sum_{j=(1-\alpha)\ell+1}^{\ell} ({\sigma'}_j^2 + \delta_i) = \|B_i\|_F^2 + \alpha \ell \delta_i.$$

By using $\|a_i\|^2 = \|C_i\|_F^2 - \|B_{i-1}\|_F^2 = (\|B_i\|_F^2 + \alpha \ell \delta_i) - \|B_{i-1}\|_F^2$, and summing over i we get

$$\|A\|_F^2 = \sum_{i=1}^{n} \|a_i\|^2 = \sum_{i=1}^{n} \|B_i\|_F^2 - \|B_{i-1}\|_F^2 + \alpha \ell \delta_i = \|B\|_F^2 + \alpha \ell \varDelta.$$

Subtracting $\|B\|_F^2$ from both sides, completes the proof. □

The combination of the three Facts, provides the following results.

Theorem 1. *Given an input matrix $A \in \mathbb{R}^{n \times d}$, α-FD with parameter ℓ returns a sketch $B \in \mathbb{R}^{\ell \times d}$ that satisfies*

$$0 \leq \|Ax\|^2 - \|Bx\|^2 \leq \|A - A_k\|_F^2/(\alpha\ell - k)$$

and projection of A onto B_k, the top k rows of B satisfies

$$\|A - \pi_{B_k}(A)\|_F^2 \leq \frac{\alpha\ell}{\alpha\ell - k}\|A - A_k\|_F^2.$$

2.2 SpaceSaving Directions

SpaceSaving Directions (abbreviated SSD) uses Algorithm 2.3 for ReduceRank. Like the SS algorithm for frequent items, it assigns the counts for the smallest counter (in this case squared singular value $\sigma_{\ell-1}^2$) to the incoming direction. Unlike the SS algorithm, we do not use $\sigma_{\ell-1}^2$ as the squared norm along each direction orthogonal to B, as that gives a consistent over-estimate.

Algorithm 2.3. ReduceRank-SS(S)

```
$\delta_i \leftarrow \sigma_{\ell-1}^2$
return diag($\sigma_1, \ldots, \sigma_{\ell-2}, 0, \sqrt{\sigma_\ell^2 + \delta_i}$).
```

Then to understand the error bounds for ReduceRank-SS, we will consider an arbitrary unit vector x. We can decompose $x = \sum_{j=1}^d \beta_j v_j$ where $\beta_j^2 = \langle x, v_j\rangle^2 > 0$ and $\sum_{j=1}^d \beta_j^2 = 1$. For notational convenience, without loss of generality, we assume that $\beta_j = 0$ for $j > \ell$. Thus $v_{\ell-1}$ represents the entire component of x in the null space of B (or B_i after processing row i).

To analyze this algorithm, at iteration $i \geq \ell$, we consider a $d \times d$ matrix $\bar{B}_i$ that has the following properties: $\|B_i v_j\|^2 = \|\bar{B}_i v_j\|^2$ for $j < \ell - 1$ and $j = \ell$, and $\|\bar{B}_i v_j\|^2 = \delta_i$ for $j = \ell - 1$ and $j > \ell$. Also let $A_i = [a_1; a_2; \ldots; a_i]$.

Lemma 3. *For any unit vector x we have* $0 \leq \|\bar{B}_i x\|^2 - \|A_i x\|^2 \leq 2\delta_i$

Proof. We prove the first inequality by induction on i. It holds for $i = \ell - 1$, since $B_{\ell-1} = A_{\ell-1}$, and $\|\bar{B}_i x\|^2 \geq \|B_i x\|^2$. We now consider the inductive step at i. Before the reduce-rank call, the property holds, since adding row a_i to both A_i (from A_{i-1}) and C_i (from B_{i-1}) increases both squared norms equally (by $\langle a_i, x\rangle^2$) and the left rotation by U^T also does not change norms on the right. On the reduce-rank, norms only change in directions v_ℓ and $v_{\ell-1}$. Direction v_ℓ increases by δ_i, and in $\bar{B}_i$ the directions $v_{\ell-1}$ also does not change, since it is set back to δ_i, which it was before the reduce-rank.

We prove the second inequality also by induction, where it also trivially holds for the base case $i = \ell - 1$. Now we consider the inductive step, given it holds for $i - 1$. First obverse that $\delta_i \geq \delta_{i-1}$ since δ_i is at least the $(\ell - 1)$st squared singular value of B_{i-1}, which is at least δ_{i-1}. Thus, the property holds up to the reduce rank step, since again, adding row a_i and left-rotating does not affect the difference in norms. After the reduce rank, we again only need to consider the two directions changed $v_{\ell-1}$ and v_ℓ. By definition $\|A_i v_{\ell-1}\|^2 + 2\delta_i \geq \|C_i v_{\ell-1}\|^2 = \delta_i = \|\bar{B}_i v_{\ell-1}\|^2$, so direction $v_{\ell-1}$ is satisfied. Then $\|\bar{B}_i v_\ell\|^2 = \|B_i v_\ell\|^2 = \delta_i + \|C_i v_\ell\|^2 \leq 2\delta_i$ and $0 \leq \|A_i v_\ell\|^2 \leq \|\bar{B}_i v_\ell\|^2$. Hence $\|\bar{B}_i v_\ell\|^2 - \|A_i v_\ell\|^2 \leq 2\delta_i - 0$, satisfying the property for direction v_ℓ, and completing the proof. □

Now we would like to prove the three Facts needed for relative error bounds for $B = B_n$. But this does not hold since $\|B\|_F^2 = \|A\|_F^2$ (an otherwise nice property), and $\|\bar{B}\|_F^2 \gg \|A\|_F^2$. Instead, we first consider yet another matrix $\hat{B}$ defined as follows with respect to B. B and $\hat{B}$ have the same right singular values V. Let $\delta = \delta_n$, and for each singular value σ_j of B, adjust the corresponding singular values of $\hat{B}$ to be $\hat{\sigma}_j = \max\{0, \sqrt{\sigma_j^2 - 2\delta}\}$. Now from Lemma 3 it immediately follows that:

Lemma 4. *For any unit vector x we have $0 \leq \|Ax\|^2 - \|\hat{B}x\|^2 \leq 2\delta$ and $\|A\|_F^2 - \|\hat{B}\|_F^2 \geq \delta(\ell - 1)$.*

Thus $\hat{B}$ satisfies the three Facts. We can now state the following property about B directly, setting $\alpha = (1/2)$, adjusting ℓ to $\ell - 1$, then adding back the at most $2\delta = \Delta \leq \|A - A_k\|_F^2/(\alpha\ell - \alpha - k)$ to each directional norm.

Theorem 2. *After obtaining a matrix B from SSD on a matrix A with parameter ℓ, the following properties hold:*
- $\|A\|_F^2 = \|B\|_F^2$.
- *for any unit vector x and for $k < \ell/2 - 1/2$, we have $|\|Ax\|^2 - \|Bx\|^2| \leq \|A - A_k\|_F^2/(\ell/2 - 1/2 - k)$.*
- *for $k < \ell/2 - 1$ we have $\|A - \pi_B^k(A)\|_F^2 \leq \|A - A_k\|_F^2(\ell - 1)/(\ell - 1 - 2k)$.*

2.3 Compensative Frequent Directions

In original FD, the computed sketch B underestimates Frobenius norm of stream [20]. In Compensative FrequentDirections (abbreviated CFD), we keep track of the total mass $\Delta = \sum_{i=1}^n \delta_i$ subtracted from squared singular values (this requires only an extra counter). Then we slightly modify the FD algorithm. In the final step where $B = S'V^T$, we modify S' to $\hat{S}$ by setting each singular value $\hat{\sigma}_j = \sqrt{{\sigma'}_j^2 + \Delta}$, then we instead return $B = \hat{S}V^T$.

It now follows that for any $k \leq \ell$, including $k = 0$, that $\|A\|_F^2 = \|B\|_F^2$, that for any unit vector x we have $|\|Ax\|_F^2 - \|Bx\|_F^2| \leq \Delta \leq \|A - A_k\|_F^2/(\ell - k)$ for any $k < \ell$, and since V is unchanged that $\|A - \pi_B^k(A)\|_F^2 \leq \|A - A_k\|_F^2 \ell/(\ell - k)$.

3 Experiments

Herein we describe an extensive set of experiments on a wide variety of large input data sets. We show improvements over FD (and in one instance iSVD) by

Table 1. Datasets; numeric rank is defined $\|A\|_F^2/\|A\|_2^2$

DataSet	# Datapoints	# Attributes	Rank	Numeric Rank
Random Noisy	10000	500	500	$m{=}50$ $m{=}30$ $m{=}20$ $m{=}10$ 21.62, 15.39, 11.82, 8.79
Adversarial	10000	500	500	1.69
Birds [2]	11788	312	312	12.50
Spam [1]	9324	499	499	3.25

our proposed algorithm 0.2-FD. All experiments are easily reproducible through a configuration we have prepared on the APT [32] system.

Each data set is an $n \times d$ matrix A, and the n rows are processed one-by-one in a stream. We also compare against some additional baseline streaming techniques, exemplifying the three alternatives to techniques based on FD. We perform a separate set of experiments to compare accuracy of our algorithms against each other and against exemplar algorithms in hashing, random projection and column sampling line of works.

Competing Algorithms. For randomized algorithms, we average over 5 trials.

Random Projection: In this method [27,35], sketch B is constructed by multiplying a projection matrix R into the input matrix A. R is a $\ell \times n$ matrix where each entry $R_{i,j} \in \{-1/\sqrt{\ell}, 1/\sqrt{\ell}\}$ uniformly. In fact matrix R randomly projects columns of A from dimension n to dimension ℓ. This method needs $O(\ell d)$ space and update time per row is $O(\ell d)$.

Hashing: In this method [38], there are two hash functions $h : [n] \to [\ell]$ and $s : [n] \to \{-1, +1\}$ which map each row of A to a row of sketch B and to either $+1$ or -1, respectively. More precisely, B is initialized to be a $\ell \times d$ zero matrix, then when we process row a_i, we change B as $B_{h(i)} = B_{h(i)} + s(i)a_i$.

Sampling: Column Sampling [15,16,34] (which translates to row sampling in our setting), samples ℓ rows a_i of matrix A with replacement proportional to $\|a_i\|^2$ and rescales each chosen rows to have norm $\|A\|_F/\sqrt{\ell}$. This method requires $O(d\ell)$ space and the update time per row is $O(d)$ when implemented as ℓ independent reservoir sampler. We do not consider other column sampling techniques with better error guarantees since they cannot operate in a stream.

FD and **iSVD** are described in detail in Section 2.

Datasets. We compare performance of our algorithms on both synthetic and real datasets; see a summary in Table 1. We also generate adversarial data to show that iSVD performs poorly under specific circumstances, this explains why there is no theoretical guarantee for them.

For Random Noisy, we generate the input $n \times d$ matrix A synthetically, mimicking the approach by Liberty [26]. We compose $A = SDU + F/\zeta$, where SDU is the m-dimensional signal (for $m < d$) and G/ζ is the (full) d-dimensional noise

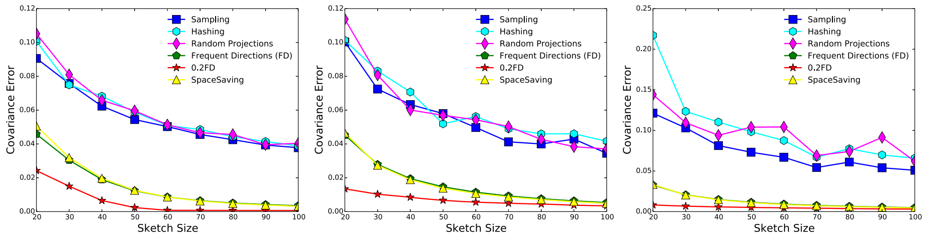


Fig. 1. Covariance Error: Random Noisy(50) (left), Birds (middle), and Spam (right)

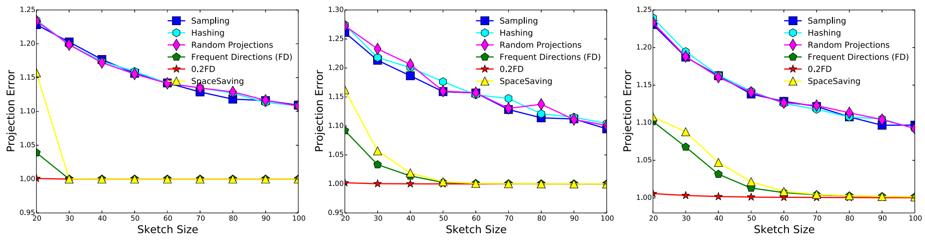


Fig. 2. Projection Error: Random Noisy(50) (left), Birds (middle), and Spam (right)

with ζ controlling the signal to noise ratio. Each entry $F_{i,j}$ of F is generated i.i.d. from a normal distribution $N(0,1)$, and we set $\zeta = 10$. For the signal, $S \in \mathbb{R}^{n\times m}$ again with each $S_{i,j} \sim N(0,1)$ i.i.d; D is diagonal with entries $D_{i,i} = 1-(i-1)/d$ linearly decreasing; and $U \in \mathbb{R}^{m\times d}$ is just a random rotation. We use $n = 10000$, $d = 500$, and consider $m \in \{10, 20, 30, 50\}$ (the default is $m = 50$).

In order to create Adversarial data, we constructed two orthogonal subspaces $S_1 = \mathbb{R}^{m_1}$ and $S_2 = \mathbb{R}^{m_2}$ ($m_1 = 400$ and $m_2 = 4$). Then we picked two separate sets of random vectors Y and Z and projected them on S_1 and S_2, respectively. Normalizing the projected vectors and concatenating them gives us the input matrix A. All vectors in $\pi_{S_1}(Y)$ appear in the stream before $\pi_{S_2}(Z)$; this represents a very sudden and orthogonal shift. As the theorems predict, FD and our proposed algorithms adjust to this change and properly compensate for it. However, since $m_1 \geq \ell$, then iSVD cannot adjust and always discards all new rows in S_2 since they always represent the smallest singular value of B_i.

We consider two real-world datasets. Birds [2] has each row represent an image of a bird, and each column a feature. PCA is a common first approach in analyzing this data, so we center the matrix. Spam [1] has each row represent a spam message, and each column some feature; it has dramatic and abrupt feature drift over the stream, but not as much as Adversarial.

Approximation Error vs. Sketch Size. We measure error for all algorithms as we change the parameter ℓ (Sketch Size) determining the number of rows in matrix B. We measure covariance error as err $= \|A^TA - B^TB\|_2/\|A\|_F^2$ (Covariance Error); this indicates for instance for FD, that err should be at most $1/\ell$, but could be

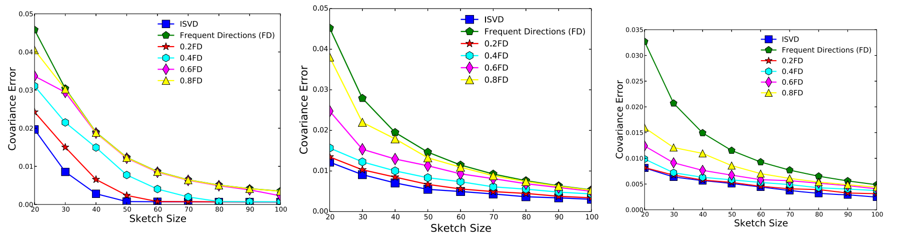


Fig. 3. Parametrized FD on Random Noisy(50) (left), Birds (middle), and Spam (right)

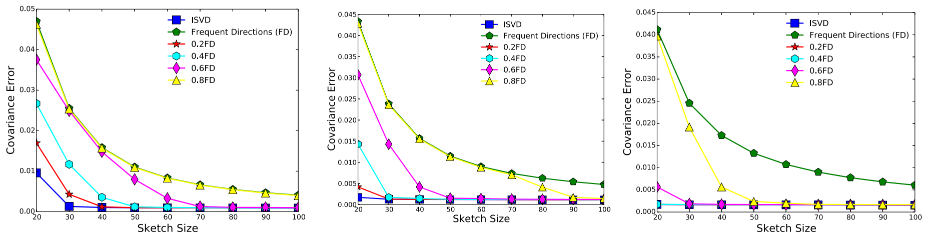


Fig. 4. Parametrized FD on Random Noisy for $m = 30$ (left), 20 (middle), 10 (right)

dramatically less if $\|A - A_k\|_F^2$ is much less than $\|A\|_F^2$ for some not so large k. We also consider proj-err $= \|A - \pi_{B_k}(A)\|_F^2 / \|A - A_k\|_F^2$, always using $k = 10$ (Projection Error); for FD we should have proj-err $\leq \ell/(\ell - 10)$, and ≥ 1 in general.

We see in Figure 1 for Covariance Error and Figure 2 for Projection Error that all baseline algorithms Sampling, Hashing, and Random Projections perform *much* worse than FD and the variants we consider. Each of Sampling, Hashing, and Random Projections perform about the same. Moreover, FD typically is out-performed by iSVD and our best proposed method 0.2-FD. Thus, we now focus only on FD, iSVD, and the new proposed methods which operate in a smaller error regime. For simplicity now we only examine Covariance Error, Projection Error acts similarly.

Next we consider Parametrized FD; we denote each variant as α-FD in Figure 3. We explore the effect of the parameter α, and run variants with $\alpha \in \{0.2, 0.4, 0.6, 0.8\}$, comparing against FD ($\alpha = 1$) and iSVD ($\alpha = 0$). Note that the guaranteed error gets worse for smaller α, so performance being equal, it is preferable to have larger α. Yet, we observe empirically that FD is consistently the worst algorithm, and iSVD consistently the best, and as α decreases, the observed error improves. The difference can be quite dramatic; for instance in the Spam dataset, for $\ell = 20$, FD has err $= 0.032$ while iSVD and 0.2-FD have err $= 0.008$. Yet, as ℓ approaches 100, all algorithms seems to be approaching the same small error. We also explore the effect on α-FD in Figure 4 on Random Noisy data by varying $m \in \{10, 20, 30\}$, and $m = 50$ in Figure 3. We observe that all algorithms get smaller error for smaller m (there are fewer "directions" to approximate), but that each α-FD variant reaches 0.005 err before $\ell = 100$, sooner for smaller α; eventually "snapping" to a smaller 0.002 err level.

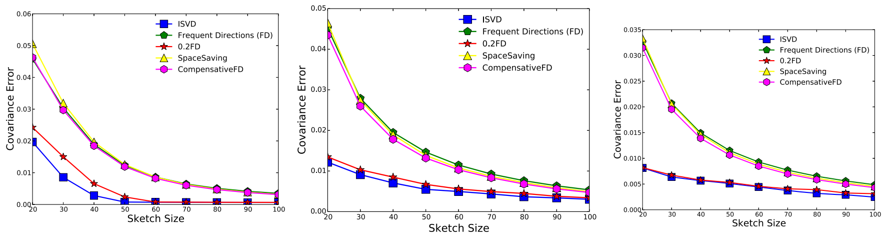


Fig. 5. SpaceSaving algos on Random Noisy(50) (left), Birds (middle), and Spam (right)

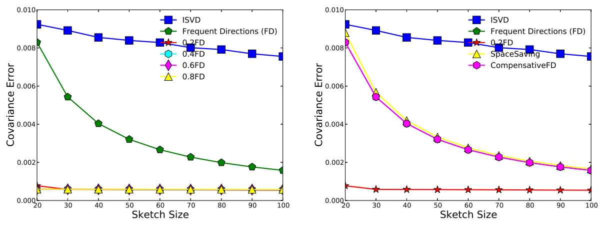


Fig. 6. Demonstrating dangers of iSVD on Adversarial data

In Figure 5, we compare iSVD, FD, and 0.2-FD with the other variants based on the SS streaming algorithm: CFD and SSD. We see that these typically perform slightly better than FD, but not nearly as good as 0.2-FD and iSVD. Perhaps it is surprising that although SpaceSavings variants empirically improve upon MG variants for frequent items, 0.2-FD (based on MG) can largely outperform the all SS variants on matrix sketching.

Finally, we show that iSVD is not always better in practice. Using the Adversarial construction in Figure 6, we see that iSVD can perform much worse than the other techniques. Although at $\ell = 20$, iSVD and FD roughly perform the same (with about $\mathsf{err} = 0.09$), iSVD does not improve much as ℓ increases, obtaining only $\mathsf{err} = 0.08$ for $\ell = 100$. On the other hand, FD (as well as CFD and SSD) decrease markedly and consistently to $\mathsf{err} = 0.02$ for $\ell = 100$. Moreover, all version of α-FD obtain roughly err=0.005 already for $\ell = 20$. The large-norm directions are the first 4 singular vectors (from the second part of the stream) and once these directions are recognized as having the largest singular vectors, they are no longer decremented in any Parametrized FD algorithm.

Reproducibility. All experiments are conducted on a Linux Ubuntu 12.04 machine with 16 cores of Intel(R) Xeon(R) CPU(2.10GHz) and 48GB of RAM. We provide public access to all of our results using a testbed facility *APT* [32]. APT is a platform where researchers can perform experiments and keep them public for verification and validation of the results. We provide our code, datasets, and experimental results in our APT profile with detailed description on how to reproduce, available at: `http://aptlab.net/p/MatrixApx/FrequentDirection`.

References

1. Concept drift in machine learning and knowledge discovery group, `http://mlkd.csd.auth.gr/concept_drift.html`
2. vision.caltech, `http://www.vision.caltech.edu/visipedia/CUB-200-2011.html`
3. Achlioptas, D., McSherry, F.: Fast computation of low rank matrix approximations. In: STOC (2001)
4. Agarwal, P.K., Cormode, G., Huang, Z., Phillips, J.M., Wei, Z., Yi, K.: Mergeable summaries. In: Proceedings of the 31st Symposium on Principles of Database Systems (2012)
5. Arasu, A., Babu, S., Widom, J.: An abstract semantics and concrete language for continuous queries over streams and relations (2002)
6. Bonnet, P., Gehrke, J., Seshadri, P.: Towards sensor database systems. In: Tan, K.-L., Franklin, M.J., Lui, J.C.-S. (eds.) MDM 2001. LNCS, vol. 1987, pp. 3–14. Springer, Heidelberg (2000)
7. Boutsidis, C., Drineas, P., Magdon-Ismail, M.: Near optimal column-based matrix reconstruction. In: FOCS (2011)
8. Brand, M.: Incremental singular value decomposition of uncertain data with missing values. In: Heyden, A., Sparr, G., Nielsen, M., Johansen, P. (eds.) ECCV 2002, Part I. LNCS, vol. 2350, pp. 707–720. Springer, Heidelberg (2002)
9. Chen, J., DeWitt, D.J., Tian, F., Wang, Y.: Niagaracq: A scalable continuous query system for internet databases. ACM SIGMOD Record 29(2), 379–390 (2000)
10. Clarkson, K.L., Woodruff, D.P.: Numerical linear algebra in the streaming model. In: STOC (2009)
11. Cormode, G., Hadjieleftheriou, M.: Finding frequent items in data streams. In: VLDB (2008)
12. Cortes, C., Fisher, K., Pregibon, D., Rogers, A.: Hancock: a language for extracting signatures from data streams. In: KDD (2000)
13. Demaine, E.D., López-Ortiz, A., Munro, J.I.: Frequency Estimation of Internet Packet Streams with Limited Space. In: Möhring, R.H., Raman, R. (eds.) ESA 2002. LNCS, vol. 2461, p. 348. Springer, Heidelberg (2002)
14. Deshpande, A., Vempala, S.S.: Adaptive Sampling and Fast Low-Rank Matrix Approximation. In: Díaz, J., Jansen, K., Rolim, J.D.P., Zwick, U. (eds.) APPROX 2006 and RANDOM 2006. LNCS, vol. 4110, pp. 292–303. Springer, Heidelberg (2006)
15. Drineas, P., Kannan, R.: Pass efficient algorithms for approximating large matrices. In: SODA (2003)
16. Drineas, P., Kannan, R., Mahoney, M.W.: Fast Monte Carlo algorithms for matrices II: Computing a low-rank approximation to a matrix. SIAM Journal on Computing 3636(1), 158–183 (2006)
17. Drineas, P., Mahoney, M.W., Muthukrishnan, S.: Relative-error CUR matrix decompositions. SIAM Journal on Matrix Analysis and Applications 30, 844–881 (2008)
18. Frieze, A., Kannan, R., Vempala, S.: Fast Monte-Carlo algorithms for finding low-rank approximations. In: FOCS (1998)
19. Ghashami, M., Liberty, E., Phillips, J.M.: Frequent directions: Simple and deterministic matrix sketchings. In: Personal Communication (2014)
20. Ghashami, M., Phillips, J.M.: Relative errors for deterministic low-rank matrix approximation. In: SODA (2014)

21. Gilbert, A.C., Kotidis, Y., Muthukrishnan, S., Strauss, M.: Quicksand: Quick summary and analysis of network data. Technical report, DIMACS 2001-43 (2001)
22. Golub, G.H., van Loan, C.F.: Matrix Computations, vol. 3. JHUP (2012)
23. Hall, P., Marshall, D., Martin, R.: Incremental eigenanalysis for classification. In: British Machine Vision Conference (1998)
24. Karp, R.M., Shenker, S., Papadimitriou, C.H.: A simple algorithm for finding frequent elements in streams and bags. ACM ToDS 28, 51–55 (2003)
25. Levey, A., Lindenbaum, M.: Sequential Karhunen-Loeve basis extraction and its application to images. IEEE ToIP 9, 1371–1374 (2000)
26. Liberty, E.: Simple and deterministic matrix sketching. In: KDD (2013)
27. Liberty, E., Woolfe, F., Martinsson, P.-G., Rokhlin, V., Tygert, M.: Randomized algorithms for the low-rank approximation of matrices. PNAS 104(51), 20167–20172 (2007)
28. Mahoney, M.W., Drineas, P.: CUR matrix decompositions for improved data analysis. PNAS 106, 697–702 (2009)
29. Metwally, A., Agrawal, D., Abbadi, A.E.: An integrated efficient solution for computing frequent and top-k elements in data streams. ACM ToDS 31, 1095–1133 (2006)
30. Misra, J., Gries, D.: Finding repeated elements. Sc. Comp. Prog. 2, 143–152 (1982)
31. Papadimitriou, C.H., Tamaki, H., Raghavan, P., Vempala, S.: Latent semantic indexing: A probabilistic analysis. In: PODS (1998)
32. Ricci, R.: Apt (adaptable profile-driven testbed) (2014), `http://www.flux.utah.edu/project/apt`
33. Ross, D.A., Lim, J., Lin, R.-S., Yang, M.-H.: Incremental learning for robust visual tracking. IJCV 77, 125–141 (2008)
34. Rudelson, M., Vershynin, R.: Sampling from large matrices: An approach through geometric functional analysis. Journal of the ACM 54(4), 21 (2007)
35. Sarlos, T.: Improved approximation algorithms for large matrices via random projections. In: FOCS (2006)
36. Sullivan, M., Heybey, A.: A system for managing large databases of network traffic. In: Proceedings of USENIX (1998)
37. Emulab testbed, `http://www.flux.utah.edu/project/emulab`
38. Weinberger, K., Dasgupta, A., Langford, J., Smola, A., Attenberg, J.: Feature hashing for large scale multitask learning. In: ICML (2009)
39. Zhu, Y., Shasha, D.: Statstream: Statistical monitoring of thousands of data streams in real time. In: VLDB (2002)

Computing Regions Decomposable into m Stars[⋆]

Matt Gibson[1], Kasturi Varadarajan[2], and Xiaodong Wu[3]

[1] Dept. of Computer Science
University of Texas at San Antonio
San Antonio, TX, USA
[2] Dept. of Computer Science
[3] Dept. of Electrical and Computer Engineering
University of Iowa
Iowa City, IA, USA

Abstract. Motivated by an approach to image segmentation, we consider an optimization problem on a node-weighted planar grid graph, that of finding a maximum weight subset of nodes that can be partitioned into m subsets, each with a simple geometric structure. The problem has been considered in earlier papers, which give polynomial time algorithms for $m = 2$. We show that the problem admits a polynomial time algorithm for any fixed $m \geq 3$.

1 Introduction

In this article, we consider a planar partitioning problem that is motivated from *image segmentation.* We are given an $(N \times N)$ grid graph $\mathcal{G}$ with node set $\{(i,j) \mid 1 \leq i,j \leq N\}$. There is an arc connecting two nodes (i,j) and (i',j') if and only if $|i-i'| + |j-j'| = 1$. Each grid node c has a weight $w(c) \in \mathbb{R}$ (note weights can be negative). We are interested in computing a maximum-weight subset of nodes in $\mathcal{G}$ that has a particular geometric structure.

One main challenge when dealing with objects in digital geometry is that many standard geometric objects and definitions from Euclidean geometry do not have "trivial" counterparts in the digital setting. For example, it is a non-trivial task to define line segments between grid nodes in the digital setting such that the line segments (1) satisfy some standard axioms of Euclidean line segments and (2) "look similar" to their corresponding Euclidean line segments. The work of [4,6] gives, for each grid node c, a spanning tree $\mathcal{S}_c$ of the grid graph. The *digital line segment* $dig(a,b)$ between two nodes a and b is then simply (the set of nodes on) the unique path between a and b in $\mathcal{S}_a$ (it turns out that we get the same path in $\mathcal{S}_b$). We can now define a subset S of the grid to be *star-shaped* if there exists some "center" grid node $c \in S$ such that for any $s \in S$ we have $dig(c,s) \subseteq S$.

Motivated by this, in the problem we consider we are given a spanning tree $\mathcal{S}_c$ for each node c of $\mathcal{G}$. Finally, we are given an integer parameter m. A feasible

[⋆] This research was partially supported by NSF grants CCF-0844765 and CCF-1318996.

solution to our optimization problem consists of m disjoint subsets of nodes $S'_1, S'_2, \ldots, S'_m$ with the property that each S'_i is star-shaped. The objective is to maximize the sum of the weights of the nodes in $S'_1 \cup S'_2 \cup \cdots \cup S'_m$.

It is not hard to develop a polynomial time algorithm for the problem when $m = 1$: guess node c_1, think of the spanning tree $\mathcal{S}_{c_1}$ as rooted at c_1, and use a recursive algorithm to find the maximum weight subtree of $\mathcal{S}_{c_1}$ that contains c_1. For $m = 2$, a case that is considerably more difficult, Chun et al. [5] gave a polynomial time algorithm based on dynamic programming, and subsequently, Gibson et al. [9] found a polynomial time algorithm based on maximum flow. The latter approach generalizes to the scenario where the underlying graph is an arbitrary graph that is not necessarily a grid. Here we ask if there is a polynomial time algorithm for any fixed $m \geq 3$. Addressing first the general case, we can prove the following theorem (proof omitted due to lack of space).

Theorem 1. *Given an arbitrary node-weighted graph G, a set of spanning trees $\mathcal{S}_c$ for each node c of G, the problem of computing a maximum-weight set of 3 disjoint subsets of nodes S'_1, S'_2, S'_3 with the property that each S'_i is star-shaped is NP-hard.*

This naturally brings up the question of whether the geometry of the grid can somehow be exploited to develop a polynomial time algorithm for fixed $m \geq 3$. This is the question that this article answers.

Motivation from Image Segmentation. Image segmentation, which aims to define accurate boundaries for the objects of interest captured by image data, has recently attracted extensive attention in the pattern recognition and computer vision communities. It is a central problem in medical image analysis. Accurate image segmentation techniques promise to improve medical diagnosis and revolutionize the current medical imaging practice, such as tumor detection, surgery planning, tissue volume measurement, and other health related issues.

In an attempt to find a "good" segmentation, the problem is often cast as an optimization problem. This often involves constructing a weighted grid graph (the graph $\mathcal{G}$ above) where the grid nodes in the graph correspond to the pixels in the input image. The weights are assigned in a way that captures the likelihood of a particular pixel being in the object of interest, and then we attempt to find some subset of the nodes that optimizes an objective function subject to some constraints. In this context, a subset of the node set of the grid is often called a *region*, and the constraints are often used to enforce the output to have a particular geometric shape. There have been several results which show that it is possible to develop efficient algorithms for finding the optimal region with some simple geometric structure, including x-monotone regions, based monotone regions, rectilinear convex regions, and star-shaped regions [3, 6, 7, 11, 12].

Regions that can be Decomposed into Regions with "Simple" Structure. Chun et al. [5] consider the maximum-weight region problem with a twist on the constraints of previous work. The authors are interested in finding a maximum-weight region that may not have simple geometric structure, but can be *decomposed* into objects with simple geometric structure. We say that a subset of the grid nodes can be decomposed into m objects of a particular structure if and only if there exists a coloring of the grid nodes in the subset using m colors such that for each color, the subset consisting of grid nodes of that color has the desired structure. For example, a feasible solution to our problem, $S'_1 \cup S'_2 \cup \cdots \cup S'_m$, can be decomposed into m objects (the S'_i), each of which is star-shaped. Chun et al. give an efficient dynamic programming algorithm for computing the maximum-weight region that is decomposable into two star-shaped regions. Recently, Gibson et al. [9] give an efficient maximum flow based algorithm for the same problem. (Thus these algorithms solve precisely the same problem that we address for $m = 2$.)

Computing these types of objects is an interesting problem both from a practical perspective as well as a theoretical perspective. In practice, we can identify a more complex class of objects while still being able to control the topology of the returned object (see Figure 1). In theory, the decomposablity constraint poses an interesting algorithmic challenge to overcome. If we instead consider computing an object which is the union of m simple objects, the problem becomes much harder, and in fact the problem of computing the maximum-weight object which is the union of two star-shaped objects is NP-hard even when the underlying graph is a grid graph [5]. The decomposability problem may emit polynomial-time algorithms, but the design of such an algorithm is a non-trivial task even for $m = 2$ and often becomes much harder when $m = 3$. The mutual interaction of three disjoint regions forms a known form of so-called "frustrated cycles" [10], which represents a computational barrier in computer vision. Neither the Chun et al. nor the Gibson et al. algorithms are easily extended to handle objects that can be decomposed into more than two stars. Another such example is the work of Delong and Boykov [8]. See Anzai et al. [2] for an example of one case where an algorithm is given that can handle three disjoint regions.

Our Contribution. Let us recall the main question in this article – is there a polynomial time algorithm which computes the maximum weight region ($S'_1 \cup \cdots \cup S'_m$) of the grid that can be decomposed into m objects (the S'_i) such that each object is star-shaped (there is a $c_i \in S'_i$ such that the subgraph of $\mathcal{S}_{c_i}$ induced by S'_i is a spanning tree on S'_i)? Here m is an integer constant. Our main result is an affirmative answer to this question.

Theorem 2. *Given a node-weighted grid graph graph $\mathcal{G}$, a set of spanning trees $\mathcal{S}_c$ for each node c of $\mathcal{G}$, and a fixed integer $m \geq 3$, there is a polynomial-time algorithm which computes m disjoint subsets of nodes $S'_1, S'_2, \ldots, S'_m$ with the properties that each S'_i is star-shaped and the weight of $S'_1 \cup S'_2 \cup \cdots \cup S'_m$ is maximized.*

To explain our approach to obtain the polynomial time algorithm, let us first observe that we can guess the grid node c_i such that S'_i is *star-shaped with respect to c_i*. (That is, S'_i contains c_i and the subgraph of $\mathcal{S}_{c_i}$ induced by S'_i spans S'_i.) We refer to c_i as the *center* of the star-shaped region S'_i. Now the unknown S'_i is potentially (the node set of) any subtree of $\mathcal{S}_{c_i}$ that contains c_i, and is

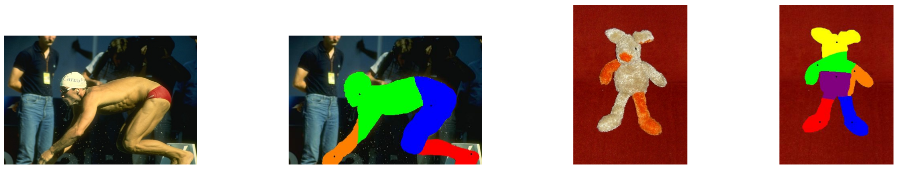


Fig. 1. Two "general" objects which can be decomposed into a small number of star-shaped objects

thus a complex shape. Thus the interaction between two distinct star-shaped regions (or *stars*) S'_i and S'_j in the optimal solution is also a complex one. Yet the algorithms of Chun et al. [5] and Gibson et al. [9] are able to handle this complexity in solving the 2-star problem (the case $m = 2$). So we aim to reduce the problem for $m \geq 3$ to several instances of the 2-star problem.

To be able to do this efficiently, we have to hope that the structure of the optimal solution has some features that we can guess. The optimal stars themselves do not *directly* seem to have any such features in all cases. So we define a Voronoi decomposition of the grid with the stars as the Voronoi centers. Our first contribution is the high level observation that knowing the 'vertices' of this Voronoi diagram, and 'connecting' them to the centers c_i of the stars appropriately, gives us a way of decomposing the problem into several 2-star instances. Our second, more technical, contribution is the careful realization of this high level observation in the context of the grid graph: how to define the Voronoi diagram and its useful vertex features so that there are only a constant number of them? And how to connect the vertex features to the centers of the stars meaningfully, in a way that does not require too much guessing?

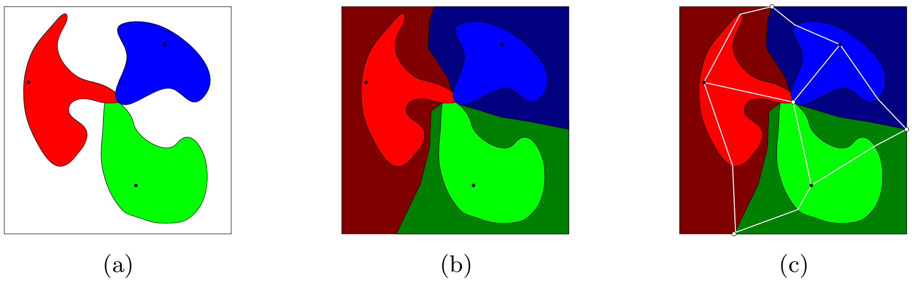


Fig. 2. Algorithm overview: (a) The unknown optimal region, and its decomposition into three stars R, B, and G. (b) The three corresponding Voronoi regions. (c) By guessing the star centers, the Voronoi vertices, and suitable connections, we obtain a decomposition of the grid into 2-star (or 1-star) subproblems.

Our polynomial algorithm has a prohibitively large running time to be directly useful practically. Our result however does point out that the bottleneck to having a useful and general algorithm for $m \geq 3$ is not the NP-hardness of the problem. Also, our Voronoi diagram approach can be an effective technique for other classes of geometric objects, as Ahmed et al. [1] recently demonstrated for objects which can be decomposed into c "based rectilinear convex objects" for any constant c.

To simplify the description of the algorithm, we will describe our algorithm in the setting where we want to find the maximum weight object decomposable into 3 star-shaped regions. It will be apparent that the algorithm can easily be extended to m stars for any constant m.

Organization of the Paper. In Section 2 we introduce some preliminaries and definitions. In Sections 3 and 4 we make some observations about the structure of an optimal solution that will be used in our algorithm. In Section 5 we give our algorithm.

2 Preliminaries

The input to our algorithm consists of the set $\mathcal{G} = \{(i,j) | 1 \leq i, j, \leq N\}$ and a weight $w(c)$ for each node $c \in \mathcal{G}$. For node (i,j), we refer to i as its x-coordinate and j as its y-coordinate. We will also denote by $\mathcal{G}$ the grid graph that was described earlier.

We can think of the grid graph as being embedded in the following natural way: grid node (i,j) embeds to point (i,j), and an adjacency between two grid nodes is represented by the straight line segment connecting the corresponding points. Sometimes, we will view the grid nodes as grid cells forming a cell-complex. For this, we think of grid cell (i,j) as the square $\{(x,y) \;|i - \frac{1}{2} \leq x \leq i + \frac{1}{2}, j - \frac{1}{2} \leq y \leq j + \frac{1}{2}\}$. In this view, each cell is incident to four vertices and four edges. We think of the nodes that are adjacent in the grid graph as also being adjacent as cells/squares. From the context, it will be clear which view of the grid we are referring to – the abstract grid graph, its embedding in the plane, or its view as a cell complex. We will use 'grid nodes' and 'grid cells' interchangeably.

Since we can enumerate the centers of the three stars in the optimal solution, we assume that as part of the input, we are given three designated "center" grid cells in $\mathcal{G}$ which we denote c_r, c_b, and c_g respectively. We want to compute the maximum weight object which can be decomposed into three star-shaped objects: one with respect to c_r, one with respect to c_b, and one with respect to c_g. Let OPT be such a maximum weight region that can be decomposed into three star-shaped regions with respect to these centers. We fix such a decompostion into 3 stars, and we denote $R \subseteq OPT$ the star-shaped object with respect to c_r, $B \subseteq OPT$ the star-shaped object with respect to c_b, and $G \subseteq OPT$ the star-shaped object with respect to c_g. We call these stars the red, blue, and green stars respectively.

Recall that a star shaped object centered at c_r is defined with respect to a spanning tree $\mathcal{S}_r$ of $\mathcal{G}$; this spanning tree is part of the input. Similarly, we have the spanning trees $\mathcal{S}_b$ and $\mathcal{S}_g$ for c_b and c_g. We can view each of these spanning trees as a rooted spanning tree with the corresponding "center" cell as the root of the tree. The path from the root to any grid cell in the tree defines the "star path" for that grid cell. For example, if a grid cell g is in R then every grid cell along the path from c_r to g in $\mathcal{S}_r$ must also be in R, since R is star-shaped with respect to c_r. See Figure 3 for an illustration.

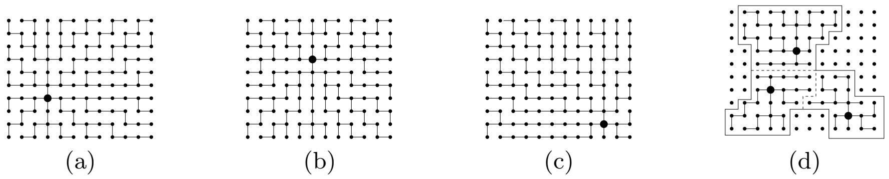


Fig. 3. An illustration of three spanning trees rooted at three different points (parts a, b, and c), and an object decomposable into three star-shaped objects (part d)

Let π be a simple path between x and y, and π' be a simple path between x' and y' in the grid graph. We will now define what it means for the two paths to be *non-crossing.* If there are no end points common to π and π', that is, $\{x, y\} \cap \{x', y'\} = \emptyset$, then the paths are non-crossing if they have no nodes in common. In case there is at least one common endpoint, we can assume w.l.o.g. that this is $y = y'$ and x may or may not be the same as x'. The paths are non-crossing in this scenario if the nodes common to both are contiguous in both paths.

Consider a grid cell g and a set of grid cells Y. We will now define a way of identifying the *unique closest cell (UCC)* in Y to g which we will denote $UCC_Y(g)$. We want that $UCC_Y(g)$ is at least as close to g as all other grid cells in Y under the L_1 metric. In the case where there are multiple such cells, we provide a simple two-step tie-breaking rule to obtain a unique closest cell. First consider the grid cells in Y that are closest to g which have the smallest x-coordinate (there can be at most two such cells). If there is only one such cell then this cell is $UCC_Y(g)$, and if there are two such cells then we choose the cell with the larger y-coordinate.

For a pair of grid cells x and y, we will now define a unique "L-shaped" path from x to y in the grid which we denote $p(x, y)$. We view $p(x, y)$ as directed, from x to y. Note that any L-shaped path between a pair of grid cells is a shortest path between the grid cells. We will describe the path $p(x, y)$ by "walking" along the path from x to y. We start at x and walk horizontally until we have reached a grid cell with the same x-coordinate as y. We then turn and walk towards y vertically until we have reached y. The grid cells that we traversed are the grid cells in our path (note that $p(x, y)$ may not be the same as $p(y, x)$ but for our purposes this is not an issue).

We now can prove the following observation which follows from the fact that a subpath of a shortest path is also a shortest path (proof omitted due to lack of space).

Lemma 1. *For any two grid cells x and x', the paths $p(x, UCC_Y(x))$ and $p(x', UCC_Y(x'))$ are non crossing (as undirected paths).*

3 Voronoi Diagram

In this section, we will define a digital Voronoi Diagram with respect to the three stars that an optimal solution decomposes into. After defining this Voronoi Diagram, we will make several observations about the structure of this Voronoi Diagram which we will crucially use in our algorithm.

For a grid cell g and a star-shaped region S, we define $d(g, S)$ to be $\min_{x \in S} d(g, x)$, where $d(g, x)$ is the number of grid cells in a shortest path between g and x in the grid topology. We will define a digital Voronoi Diagram with respect to R, B, and G. The grid cells in the grid will be partitioned into three subsets which we call *Voronoi regions.* There will be exactly one Voronoi region defined for each of the stars R, B, and G that we denote $V(R), V(B)$, and $V(G)$ respectively, and each grid cell in the grid will be assigned to exactly one Voronoi region. Consider some grid cell g. If $d(g, R) < d(g, B)$ and $d(g, R) < d(g, G)$ (i.e. g is strictly closer to R than to either B or G under the distance measure d), then g will belong to the red Voronoi region. Likewise, if g is strictly closer to B it is in the blue Voronoi region and if it is strictly closer to G then it is in the green Voronoi region. There could be grid cells in which there are two or three stars that are closest to the grid cell. To handle these grid cells, we fix an arbitrary ordering of the stars. Consider the stars a grid cell is closest to. Of these stars, assign the grid cell to the Voronoi region of the star which comes earliest in the ordering. This completes the definition of the digital Voronoi Diagram.

Observation 3. *Let g be a grid cell that is in $V(S)$ for some star S, and let $S' \subseteq S$ be the subset of grid cells in S that are closest to g. Then any shortest path from g to any grid cell in S' consists entirely of grid cells that are in $V(S)$.*

The proof of Observation 3 is omitted due to lack of space. The following corollary immediately follows from the observation.

Corollary 4. *Each Voronoi region in the Voronoi diagram is connected under the grid topology.*

For a grid cell g that belongs to the Voronoi region $V(S)$, let π_g denote the path obtained by concatenating the L-shaped path $p(g, UCC_S(g))$ to the path in the star S from $UCC_S(g)$ to the center of the star S. Note that π_g lies entirely in $V(S)$, by Observation 3. Also, let σ_g denote the piece-wise linear path that connects the centers of the cells in π_g in the order in which they occur in π_g. That is, σ_g is the path corresponding to π_g in the embedding of the grid graph.

Lemma 2. *For any two distinct cells g and g', π_g and $\pi_{g'}$ do not cross.*

4 Marking Cells

In this section, we will introduce a procedure of identifying several key grid cells which will enable us to break the problem down into several instances of the 2-star problem. For the purposes of describing this procedure, it is convenient to add the following set of *dummy cells* to the grid:

$$\{(0,j) \mid 1 \le j \le N\} \cup \{(N+1,j) \mid 1 \le j \le N\}$$
$$\cup\{(i,0) \mid 1 \le i \le N\} \cup \{(i,N+1) \mid 1 \le i \le N\}.$$

These dummy cells are not actually part of the grid and are not eligible to be included in a solution to the problem, but rather serve as a “placeholder” to help describe the procedure in a cleaner way.

Consider the set of points in the plane that lie in the cells belonging to $V(S)$, the Voronoi region for some star S. Abusing terminology slightly, we will refer to this set also as the Voronoi region. Since $V(S)$ is connected under the grid topology, the Voronoi region is a rectilinear polygon that possibly has holes. (Each such hole contains one or more other Voronoi regions.) We will refer to the boundary of the polygon with its holes filled in as the *outer frontier* $OB(S)$ of $V(S)$, and the boundary of each of the holes of the polygon as an *inner frontier* of $V(S)$. Each frontier is a union of edges of grid cells. Each cell edge that belongs to a frontier is incident on two grid cells; one of these is in $V(S)$ and the other is either a dummy cell (one that surrounds the $N \times N$ grid) or belongs to a different Voronoi region. The edges that belong to a frontier (outer or inner) can be cyclically ordered in the way they occur on the boundary.

We now describe a procedure for *marking* certain cells in each of the Voronoi regions. There is a phase of this marking procedure for each of the Voronoi regions; we describe the phase for the red region $V(R)$. The other phases are similar.

Processing the outer frontier of $V(R)$. Each cell edge on the outer frontier $OB(R)$ of R is incident on a cell that can be one of three types: a grid cell in $V(B)$, $V(G)$, or a dummy cell. (Each such edge is of course also incident on a cell in $V(R)$.) Consider any two adjacent cell edges e and e' on $OB(R)$ (adjacent with respect to the cyclic order mentioned above) for which the types of the incident grid cell are different. We mark all the cells in the grid (including the ones in $V(R)$) that are incident on e and e'. We refer to this set of cells as a *vertex feature* on $OB(R)$ induced by e and e'. Intuitively, a vertex feature is a location in the grid in which three Voronoi regions (or two Voronoi regions and the border of the grid) “come together”. We will use these vertex features to help us partition the three-star problem into several instances of a two-star problem. Notice that there may be several vertex features on $OB(R)$, but as we shall see, we can bound this number by a constant. All the corresponding cells are marked while processing the outer frontier of $V(R)$. See part (a) Figure 4 for an illustration.

Processing the boundary of each hole of $V(R)$. Fix a hole h in $V(R)$ and consider its boundary, which is an inner frontier of $V(R)$. We find *one* pair of adjacent

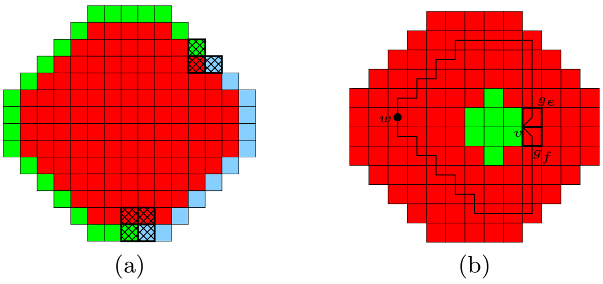


Fig. 4. Marking cells: (a) Marking an outer frontier. The 7 cells with the pattern. (b) Marking an inner frontier. Note that the two marked cells (g_e and g_f) are the two red cells with bold boundaries.

edges e and f on the frontier with the property that (a) the cell g_e in $V(R)$ incident to e and the cell g_f in $V(R)$ incident to f are distinct; and (b) the path σ_{g_e}, the straight-line segment from the center of g_e to the vertex v incident to both e and f, the straight-line segment from v to the center of g_f, and the path σ_{g_f} together enclose the hole. (Notice that σ_{g_e} and σ_{g_f} share a common portion from the center of the cell that is the center of R to some point w, after which they split and don't intersect again. Hence, the plane minus the points in σ_{g_e} and σ_{g_f} is connected. On the other hand, the concatenation of the portion of σ_{g_e} from w to the center of g_e, the segment connecting the center of g_e to v, the segment connecting v to the center of g_f, and the portion of σ_{g_f} from the center of g_f to w forms a Jordan cycle, which we denote by J_h. Condition (b) is asking if the hole h is contained within the Jordan cycle J_h.) The existence of such a pair of adjacent edges e and f follows from a topological argument – if such a pair doesn't exist, we can contract (within $V(R)$) the frontier to a point, a contradiction.

For such e and f, notice that the points contained in the cells belonging to π_{g_e} and π_{g_f} forms a rectilinear polygon, one of whose holes contains h. We will therefore refer to the collection of cells in π_{g_e} and π_{g_f} as the *red enclosure* for h.

Having picked e and f, we mark the cells g_e and g_f. We do the same for each hole of $V(R)$. See part (b) Figure 4 for an illustration.

With this, the marking phase corresponding to $V(R)$ is finished. Notice that this phase can also mark cells not in $V(R)$.

Let M denote the cells that are marked, and let $\pi(M)$ denote all the cells that belong to π_g for $g \in M$. We make the following useful observation:

Lemma 3. *For any star S, the cells $\pi(M) \cap V(S)$ induce a connected subgraph of the grid.*

The main consequence of the marking procedure is that when the cells in $\pi(M)$ are removed, each connected component that remains contains grid cells from at most two different Voronoi regions. As we explain below, this is what enables us to reduce the problem to several instances of the two-star problem.

Lemma 4. *After the removal of $\pi(M)$, any connected component Γ (of the subgraph of the grid induced by the remaining grid cells) contains cells from at most two Voronoi regions. Furthermore, the set of cells in $\pi(M)$ that are adjacent to some cell in Γ come from at most two Voronoi regions.*

We also need a bound on the number of cells marked by the procedure.

Lemma 5. *The set M of marked cells has size at most* 60.

5 The Algorithm

In this section, we give our algorithm for computing the maximum weight object decomposable into three star-shaped objects. Let us call this desired object OPT. We begin with a high level overview of the algorithm. The algorithm iteratively guesses the cells M that are marked by our marking procedure for OPT, and then guesses the unique closest cells in OPT and which star (red, blue, or green) these cells are in for each guess M. This allows us to compute the paths $\pi(M)$, which we remove from the grid, breaking the grid into connected components. For the correct guess of $\pi(M)$, we have that each connected component of the grid contains cells from at most two Voronoi regions and that the cells in $\pi(M)$ that are adjacent to the component contains cells from at most two Voronoi regions by Lemma 4. We have already guessed from which Voronoi regions the cells in $\pi(M)$ that are adjacent to each component come from (from the guess of which stars the unique closest points belong to) and this then gives us from which two Voronoi regions are the grid cells in each component. We then use the algorithm of Gibson et al. [9] to compute the maximum weight region decomposable into two star-shaped regions for each component. We then put the pieces together (combining the two-star solutions in each component with some of the grid cells in $\pi(M)$), and argue that for the correct guess, the solution we obtain is OPT. Following from Lemma 5, we only have to make a polynomial number of guesses to guarantee at some point we made all of the correct guesses.

Algorithm Details. Let M denote the grid cells that are marked by the procedure in Section 4. The algorithm first guesses these cells M. Let us call the guess M'. Then for each grid cell $g \in M'$, we guess it's unique closest cell $UCC_{OPT}(g)$. We call the union of these guessed unique closest cells C. For each cell in C we guess which star this cell is in ($\{r, b, g\}$). Now for this particular guess (M', C, and the colors of the grid cells in C) we will compute 6 sets of grid cells I_r, I_b, I_g, O_r, O_b, and O_g. These sets will contain some of the grid cells that we are guessing belong to a particular Voronoi region but are "outside" of the optimal solution (O_r, O_b, and O_g) or will contain some of the grid cells that we are guessing are "inside" one of the three stars in the optimal solution (I_r, I_b, and I_g). For a grid cell $g \in C$, let color(g) denote the color we guessed for g. For each $m \in M'$, let π_m denote the concatenation of the L-shaped path $p(m, UCC_C(m))$ with the path from $UCC_C(m)$ to $c_{\text{color}(UCC_C(m))}$ in the corresponding star (as defined in Section 3). We place each grid cell in the path $p(m, UCC_C(m))$ up to (but not including) $UCC_C(m)$ into the set $O_{\text{color}(UCC_C(m))}$ (intuitively, we are marking these cells as being in the color$(UCC_C(m))$ Voronoi region but not in the optimal solution). We place all of the grid cells in the path in the star from $UCC_C(m)$ to $c_{\text{color}(UCC_C(m))}$ into the set $I_{\text{color}(UCC_C(m))}$ (marking these cells as being in the optimal solution as part of the color$(UCC_C(m))$ star). After doing this for each $m \in M'$, we check to see that each grid cell in the grid graph was assigned to at most one of these six sets. It can easily be seen that if we have made the correct guess for OPT then it will be the case that each grid cell will be assigned to at most one of the six sets, and therefore if this is not the case then we know that we have made an incorrect guess and we do not continue with this guess and try again. So now we will assume that each grid cell has been assigned to at most one of the six sets.

Let $\pi(M')$ denote the union of π_m for each $m \in M'$, and remove $\pi(M')$ from the grid, leaving the grid broken up into several connected components. Let Γ denote a connected component of the grid after the removal of $\pi(M')$. Let $\pi_\Gamma(M')$ denote the grid cells in $\pi(M')$ that are adjacent to Γ. If the grid cells in $\pi_\Gamma(M')$ have been assigned to more than 2 distinct colors of the six sets (note that I_r and O_r are of the same color), then we know that we guessed incorrectly by Lemma 4 and we guess again. So now assume that we did guess that the grid cells in $\pi_\Gamma(M')$ have been assigned to at most 2 distinct colors of the six sets. Without loss of generality, assume that the grid cells in $\pi_\Gamma(M')$ were assigned the colors red and blue. Then Lemma 4 implies that every grid cell in Γ can only come from the red or blue Voronoi regions.

We are now ready to set up an instance of the 2-star problem for each connected component Γ. To be more precise, we will setup an instance of the more general problem that can be solved with the algorithm of Gibson et al. [9] listed in the Introduction. Without loss of generality assume that two colors of Voronoi regions that the grid cells in Γ come from are red and blue. The vertex set V of this instance is $\Gamma \cup I_r \cup I_b$. To get the two trees S'_r and S'_b, recall the definitions of $\mathcal{S}_r$ and $\mathcal{S}_b$ from Section 2. Consider the subgraph of $\mathcal{S}_r$ induced by the vertex set V. We set S'_r to be the connected component of this subgraph that contains I_r. Likewise, we set S'_b to be connected component of the the subgraph of $\mathcal{S}_b$ induced by V that contains I_b. For each grid cell in Γ the corresponding vertex in V is assigned the same weight it had in the grid graph. For each grid cell in $I_r \cup I_b$, we assign the corresponding vertex in V a weight of α where α is some value larger than the sum of weights of grid cells in Γ that have positive weights. This is because we have already guessed that these grid cells are in OPT, and we want to guarantee that these grid cells will be included in our solution. This completes the setup of the 2-star instance, and we find the optimal 2-star solution O_Γ for this instance using the algorithm of Gibson et al. Note that $I_r \cup I_b$ will always be included in O_Γ.

We denote our final object for this guess $O := \bigcup_\Gamma O_\Gamma$. This object can clearly be decomposed into three star-shaped objects. Lemma 5 implies that we must make at most a polynomial number of guesses before we are guaranteed to make the correct guess for OPT. We iterate through each of these guesses, and we remember the object with the largest weight that we obtained throughout each of the guesses and return this object as our final solution (where the weight of a grid cell is the weight in the original grid graph and not the weight assigned in the 2-star instance). At least one such guess of M', C and the colors assigned to C corresponds with OPT, and therefore the object that we compute for this guess will have weight $w(OPT)$.

References

1. Ahmed, M., Chowdhury, I., Gibson, M., Islam, M.S., Sherrette, J.: On maximum weight objects decomposable into based rectilinear convex objects. In: Dehne, F., Solis-Oba, R., Sack, J.-R. (eds.) WADS 2013. LNCS, vol. 8037, pp. 1–12. Springer, Heidelberg (2013)
2. Anzai, S., Chun, J., Kasai, R., Korman, M., Tokuyama, T.: Effect of corner information in simultaneous placement of k rectangles and tableaux. Discrete Mathematics, Algorithms and Applications 2(4), 527–537 (2010)
3. Chen, D.Z., Chun, J., Katoh, N., Tokuyama, T.: Efficient algorithms for approximating a multi-dimensional voxel terrain by a unimodal terrain. In: Chwa, K.-Y., Munro, J.I. (eds.) COCOON 2004. LNCS, vol. 3106, pp. 238–248. Springer, Heidelberg (2004)
4. Christ, T., Pálvölgyi, D., Stojakovic, M.: Consistent digital line segments. Discrete & Computational Geometry 47(4), 691–710 (2012)
5. Chun, J., Kasai, R., Korman, M., Tokuyama, T.: Algorithms for computing the maximum weight region decomposable into elementary shapes. In: Dong, Y., Du, D.-Z., Ibarra, O. (eds.) ISAAC 2009. LNCS, vol. 5878, pp. 1166–1174. Springer, Heidelberg (2009)
6. Chun, J., Korman, M., Nöllenburg, M., Tokuyama, T.: Consistent digital rays. Discrete & Computational Geometry 42(3), 359–378 (2009)
7. Chun, J., Sadakane, K., Tokuyama, T.: Efficient algorithms for constructing a pyramid from a terrain. IEICE Transactions 89-D(2), 783–788 (2006)
8. Delong, A., Boykov, Y.: Globally optimal segmentation of multi-region objects. In: ICCV, pp. 285–292. IEEE Computer Society Press, Los Alamitos (2009)
9. Gibson, M., Han, D., Sonka, M., Wu, X.: Maximum weight digital regions decomposable into digital star-shaped regions. In: Asano, T., Nakano, S.-I., Okamoto, Y., Watanabe, O. (eds.) ISAAC 2011. LNCS, vol. 7074, pp. 724–733. Springer, Heidelberg (2011)
10. Rother, C., Kolmogorov, V., Lempitsky, V.S., Szummer, M.: Optimizing binary mrfs via extended roof duality. In: CVPR. IEEE Computer Society (2007)
11. Wu, X., Chen, D.Z., Li, K., Sonka, M., Zhang, L.: The layered net surface problems in discrete geometry and medical image segmentation. International Journal of Computational Geometry and Applications 17, 261–296 (2007)
12. Wu, X., Dou, X., Wahle, A., Sonka, M.: Region detection by minimizing intraclass variance with geometric constraints, global optimality, and efficient approximation. IEEE Transactions on Medical Imaging 30, 814–827 (2011)

The Parameterized Complexity of Graph Cyclability*

Petr A. Golovach[1], Marcin Kamiński[2],
Spyridon Maniatis[3], and Dimitrios M. Thilikos[3,4]

[1] Department of Informatics, University of Bergen, Bergen, Norway
[2] Institute of Computer Science, University of Warsaw, Warsaw, Poland
[3] Department of Mathematics, National and Kapodistrian University of Athens, Athens, Greece
[4] AlGCo project-team, CNRS, LIRMM, Montpellier, France

Abstract. The cyclability of a graph is the maximum integer k for which every k vertices lie on a cycle. The algorithmic version of the problem, given a graph G and a non-negative integer k, decide whether the cyclability of G is at least k, is NP-hard. We prove that this problem, parameterized by k, is co-W[1]-hard. We give an FPT algorithm for planar graphs that runs in time $2^{2^{O(k^2 \log k)}} \cdot n^2$. Our algorithm is based on a series of graph theoretical results on cyclic linkages in planar graphs.

1 Introduction

In the opening paragraph of his book *Extremal Graph Theory* Béla Bollobás notes: "*Perhaps the most basic property a graph may posses is that of being connected. At a more refined level, there are various functions that may be said to measure the connectedness of a connected graph.*" Indeed, connectivity is one of the fundamental properties considered in graph theory and studying different variants of connectivity gives a better understanding of this property. Many such alternative connectivity measures have been studied in graph theory but very little is known about their algorithmic properties. The main goal of this paper is to focus on one of such parameters – *cyclability* – from an algorithmic point of view. Cyclability can be thought of as a quantitative measure of Hamiltonicity, or as a natural "tuning" parameter between connectivity and Hamiltonicity.

Cyclability. For a positive integer k, a graph is *k-cyclable* if every k vertices lie on a common cycle; we assume that any graph is 1-cyclable. Respectively, the *cyclability* of a graph G is the maximum integer k for which G is k-cyclable. Cyclability is well studied in the graph theory literature. Dirac proved that

* The first author was supported by the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013)/ERC Grant Agreement n. 267959. The second author was supported by the Foundation for Polish Science (HOMING PLUS/2011-4/8) and National Science Center (SONATA 2012/07/D/ST6/02432). The third and the fourth author were co-financed by the E.U. (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: "Thales. Investing in knowledge society through the European Social Fund".

cyclability of a k-connected graph is at least k, for $k \geq 2$ [6]. Watkins and Mesner [27] characterized the extremal graphs for the theorem of Dirac. There is a variant of cyclability restricted only to a set of vertices of a graph. Generalizing the theorem of Dirac, Flandrin et al. [12] proved that if a set of vertices S in a graph G is k-connected, then there is a cycle in G through any k vertices of S. (A set of vertices S is k-connected in G if a pair of vertices in S cannot be separated by removing at most $k-1$ vertices of G.) Another avenue of research is lower-bounds on cyclability of graphs in restricted families. For example, every 3-connected claw-free graph has cyclability at least 6 [22] and every 3-connected cubic planar graph has cyclability at least 23 [4].

Clearly, a graph G is Hamiltonian if and only if its cyclability equals $|V(G)|$. Therefore, we can think of cyclability as a quantitive measure of Hamiltonicity. A graph G is *hypohamiltonian* if it is not Hamiltonian but all graphs obtained from G by deleting one vertex are. Clearly, a graph G is hypohamiltonian if and only if its cyclability equals $|V(G)|-1$. Hypohamiltonian graphs appear in combinatorial optimization and are used to define facets of the traveling salesman polytope [16]. Curiously, the computational complexity of deciding whether a graph is hypohamiltonian seems to be open.

To our knowledge no algorithmic study of cyclability has been done so far. In this paper we initiate this study. For this, we consider the following problem.

CYCLABILITY
Input: A graph G and a non-negative integer k.
Question: Is every k-vertex set S in G cyclable, i.e., is there a cycle C in G such that $S \subseteq V(C)$?

CYCLABILITY with $k = |V(G)|$ is HAMILTONICITY and HAMILTONICITY is NP-complete for planar cubic graphs [15]. Hence, we have the following.

Proposition 1. CYCLABILITY *is* NP-*hard for cubic planar graphs.*

Parameterized complexity. A parameterized problem has as instances pairs (I, k) where I is the main part and k is the parameterized part. Parameterized Complexity settles the question of whether a parameterized problem is solvable by an algorithm (we call it FPT-*algorithm*) of time complexity $f(k) \cdot |I|^{O(1)}$ where $f(k)$ is a function that does not depend on n. If such an algorithm exists, we say that the parameterized problem belongs in the class FPT. In a series of fundamental papers (see [10,11,8,9]), Downey and Fellows invented a series of complexity classes, namely the classes such as $\mathsf{W}[1] \subseteq \mathsf{W}[2] \subseteq \cdots \subseteq \mathsf{W}[SAT] \subseteq \mathsf{W}[P] \subseteq \mathsf{XP}$ and proposed special types of reductions such that hardness for some of the above classes makes it rather impossible that a problem belongs in FPT (we stress that $\mathsf{FPT} \subseteq \mathsf{W}[1]$). We mention that XP is the class of parameterized problems such that for every k there is an algorithm that solves that problem in time $O(|I|^{f(k)})$, for some function f (that does not depend on $|I|$). For more on parameterized complexity, we refer the reader to [7], [13], or [20].

Our results. In this paper we deal with the parameterized complexity of CYCLABILITY parameterized by k. It is easy to see that CYCLABILITY is in XP. For a graph G, we can check all possible subsets X of $V(G)$ of size k. For each subset

X, we consider $k!$ orderings of its vertices, and for each sequence of k vertices $x_1, \ldots, x_k$ of X, we use the celebrated result of Robertson and Seymour [25] (see also [18]), to check whether there are k disjoint paths that join x_{i-1} and x_i for $i \in \{1, \ldots, k\}$ assuming that $x_0 = x_k$. We return a YES if and only if there is an ordering that has the required disjoint paths for each set X.

Is it possible that CYCLABILITY is FPT when parameterized by k? By showing that CYCLABILITY is co-W[1]-hard (even for split graphs[1]), we show that this is rather unlikely. However, we prove that the problem is FPT on planar graphs.

Theorem 1 **($\star$[2]).** *It is* W[1]*-hard to decide for a split graph G and a positive integer k, whether G has k vertices such that there is no cycle in G that contains these k vertices, when the problem is parameterized by k.*

Theorem 2. *The* CYCLABILITY *problem, when parameterized by k, is in* FPT *when its input graphs are restricted to be planar graphs. Moreover, the corresponding* FPT*-algorithm runs in $2^{2^{O(k^2 \log k)}} \cdot n^2$ steps.*

Our Techniques. Theorem 1 is proved in the appendix and the proof is a reduction from the CLIQUE problem.

The two key ingredients in the proof of Theorem 2 are a new two-step version of the *irrelevant vertex technique*, a new combinatorial concept of *cyclic linkages* and a strong notion of *vitality* on them (vital linkages played an important role in the Graph Minors series, in [26] and [23]). The proof of Theorem 2 is presented in Section 3 (with references to the appendix). Below we give a rough sketch of our method.

We work with a variant of CYCLABILITY in which some vertices (initially all) are colored. We only require that every k colored vertices lie on a common cycle. If the treewidth of the input graph G is "small" (bounded by an appropriate function of k), we employ a dynamic programming routine to solve the problem. Otherwise, there exists a cycle in a plane embedding of G such that the graph H in the interior of that cycle is "bidimensional" (contains a large subdivided wall) but still of bounded treewidth. This structure permits to distinguish in H a sequence $\mathcal{C}$ of sufficiently many concentric cycles that are all traversed by some sufficiently many paths of H. Our first aim is to check whether the distribution of the colored vertices in these cycles yields some "big uncolored area" of H. In this case we declare some "central" vertex of this area *problem-irrelevant* in the sense that its removal creates an equivalent instance of the problem. If such an area does not exists, then R is "uniformly" distributed inside the cycle sequence $\mathcal{C}$. Our next step is to set up a sequence of instances of the problem, each corresponding to the graph "cropped" by the interior of the cycles of $\mathcal{C}$, where all vertices of a sufficiently big "annulus" in it are now uncolored. As the graphs of these instances are subgraphs of H and therefore they have bounded treewidth, we can get an answer for all of them by performing a sequence of dynamic programming calls (each taking a linear number of steps). At this point, we prove that if one of these instances is a NO-instance then we just report that the initial instance is a NO-instance and we stop. Otherwise, we

[1] A graph G is *split* if $V(G)$ can be partitioned into a clique and an independent set.
[2] Proofs of results that are marked with a star "$\star$" have been moved to the Appendix.

pick a colored vertex inside the most "central" cycle of $\mathcal{C}$ and we prove that this vertex is *color-irrelevant*, i.e., an equivalent instance is created when this vertex is not any more colored. In any case, the algorithm produces either a solution or some "simpler" equivalent instance that either contains a vertex less or a colored vertex less. This permits a linear number of recursive calls of the same procedure. To prove that these two last critical steps work correctly, we have to introduce several combinatorial tools. One of them is the notion of strongly vital linkages, a variant of the notion of vital linkages introduced in [26], which we apply to terminals traversed by cycles instead of terminals linked by paths, as it has been done in [26]. This notion of vitality permits a significant restriction of the expansion of the cycles that certify that sets of k vertices are cyclable and is able to justify both critical steps of our algorithm. The proofs of the combinatorial results that support our algorithm are presented in Section 4 and we believe that they have independent combinatorial importance.

Structure of the paper. The paper is organized as follows. In Section 2 we give a minimal set of definitions that are necessary for the presentation of our algorithm. The main steps of the algorithm are presented in Section 3 and the combinatorial results (along with the necessary definitions) are presented in Section 4. We conclude with some discussion and open questions in Section 5. The paper is followed by an Appendix with 4 parts. The first one is the reduction supporting the proof of Theorem 1. The second and the third part contain the proofs of Section 3 and 4 respectively. Finally, the fourth part of the Appendix contains a series of remarks on the dynamic programming procedures that can be used to solve *Cyclability* for graphs of bounded treewidth.

2 Definitions and Preliminary Results

For any graph G, $V(G)$ (respectively $E(G)$) denotes the *set of vertices* (respectively *edges*) of G. A graph G' is a *subgraph* of a graph G if $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$, and we denote this by $G' \subseteq G$. If S is a set of vertices or a set of edges of a graph G, the graph $G \setminus S$ is the graph obtained from G after the removal of the elements of S. Given two graphs G_1 and G_2, we define $G_1 \cup G_2 = (V(G_1) \cup V(G_2), E(G_1) \cup E(G_2))$ and $G_1 \cap G_2 = (V(G_1) \cap V(G_2), E(G_1) \cap E(G_2))$.

For every vertex $v \in V(G)$ *the neighborhood* of v in G, denoted by $N_G(v)$, is the subset of vertices that are adjacent to v, and its size is called the *degree* of v in G, denoted by $\deg_G(v)$. The maximum degree $\Delta(G)$ of a graph G is the maximum value taken by $\deg_G$ over $V(G)$. A *cycle* of G is a subgraph of G that is connected and all its vertices have degree 2. We call a set of vertices $S \subseteq V(G)$ *cyclable* if for some cycle C of G, it holds that $S \subseteq V(C)$.

Treewidth. A *tree decomposition* of a graph G is a pair $\mathcal{D} = (\mathcal{X}, T)$ in which T is a tree and $\mathcal{X} = \{X_i \mid i \in V(T)\}$ is a family of subsets of $V(G)$ such that:
• $\bigcup_{i \in V(T)} X_i = V(G)$

• for each edge $e = \{u, v\} \in E(G)$ there exists an $i \in V(T)$ such that both u and v belong to X_i
• for all $v \in V$, the set of nodes $\{i \in V(T) \mid v \in X_i\}$ forms a connected subtree of T.

The *width* of a tree decomposition is $\max\{|X_i| \mid i \in V(T)\} - 1$. The *treewidth* of a graph G (denoted by $\mathbf{tw}(G)$) is the minimum width over all possible tree decompositions of G.

Concentric cycles. Let G be a graph embedded in the sphere $\mathbb{S}_0$ and let $\mathcal{D} = \{D_1, \dots, D_r\}$, be a sequence of closed disks in $\mathbb{S}_0$. We call $\mathcal{D}$ *concentric* if $D_1 \subseteq D_2 \subseteq \cdots \subseteq D_r$ and no point belongs in the boundary of two disks in $\mathcal{D}$. We call a sequence $\mathcal{C} = \{C_1, \dots, C_r\}$, $r \geq 2$, of cycles of G *concentric* if there exists a concentric sequence of closed disks $\mathcal{D} = \{D_1, \dots, D_r\}$, such that C_i is the boundary of D_i, $i \in \{1, \dots, r\}$. For $i \in \{1, \dots, r\}$, we set $\bar{C}_i = D_i$, $\mathring{C}_i = \bar{C}_i \setminus C_i$, and $\hat{C}_i = G \cap D_i$ (notice that $\bar{C}_i$ and $\mathring{C}_i$ are sets while $\hat{C}_i$ is a subgraph of G). Given $i, j, i \leq j - 1$, we denote by $\hat{A}_{i,j}$ the graph $\hat{C}_j \setminus \mathring{C}_i$. Finally, given a $q \geq 1$, we say that a set $R \subseteq V(G)$ is *q-dense* in $\mathcal{C}$ if, for every $i \in \{1, \dots, r - q + 1\}$, $V(\hat{A}_{i,i+q-1}) \cap R \neq \emptyset$.

Railed annulus. Let r and q be integers such that $r \geq 2$ and $q \geq 1$ and let G be a graph embedded in the $\mathbb{S}_0$. A (r, q)-*railed annulus* in G is a pair $(\mathcal{C}, \mathcal{W})$ such that $\mathcal{C} = \{C_1, C_2, \dots, C_r\}$ is a sequence of r concentric cycles that are all met by a sequence $\mathcal{W}$ of q paths $P_1, P_2, \dots, P_q$ (called *rails*) in such a way that $\bigcup \mathcal{W} \subseteq A_{1,r}$ and the intersection of a cycle and a rail is always connected, that is, it is a (possibly trivial) path.

Walls and subdivided walls. Let $k \geq 1$. A *wall of height* k is the graph obtained from a $((k + 1) \times (2 \cdot k + 2))$-grid with vertices (x, y), $x \in \{1, \dots, 2 \cdot k + 4\}$, $y \in \{1, \dots, k + 1\}$, after the removal of the "vertical" edges $\{(x, y), (x, y + 1)\}$ for odd $x + y$, and then the removal of all vertices of degree 1. We denote such a wall by W_k. A *subdivided wall of height* k is a wall obtained from W_k after replacing some of its edges by paths without common internal vertices. We call such a path an *edge-path* of W. The *perimeter* P_W of a subdivided wall W of height k is the cycle defined by its boundary. Let $C_2 = P_W$ and let C_1 be any cycle of W that has no common vertices with P_W. Notice that $\mathcal{C} = \{C_1, C_2\}$ is a sequence of concentric cycles in G. We define the *compass* K_W *of* W *in* G as the graph $\hat{C}_2$. Given a graph G we denote by $\mathbf{gw}(G)$ the maximum h for which G contains a subdivided wall of height h as a subgraph. The next lemma follows easily combining results in [14], [17], and [24].

Lemma 1. *If G is a planar graph, then* $\mathbf{tw}(G) \leq 9 \cdot \mathbf{gw}(G) + 1$.

3 The Algorithm

This section is devoted to the proof of Theorem 2. We consider the following slightly more general problem.

PLANAR ANNOTATED CYCLABILITY
Input: A plane graph G, a set $R \subseteq V(G)$, and a non-negative integer k.
Question: Does there exist, for every set S of k vertices in R, a cycle C of G such that $S \subseteq V(C)$?

In this section, for simplicity, we denote PLANAR ANNOTATED CYCLABILITY by Π. Theorem 2 follows directly from the following lemma.

Lemma 2. *There is an algorithm that solves* Π *in* $2^{2^{O(k^2 \log k)}} \cdot n^2$ *steps.*

Problem/color-irrelevant vertices. Let (G, k, R) be an instance of Π. We call a vertex $v \in V(G) \setminus R$ *problem-irrelevant* if (G, k, R) is a YES-instance if and only if $(G \setminus v, k, R)$ is a YES-instance. We call a vertex $v \in R$ *color-irrelevant* when (G, k, R) is a YES-instance if and only if $v \in R$ and $(G, k, R \setminus \{v\})$ is a YES-instance.

Before we present the algorithm of Lemma 2, we need to introduce three algorithms that are used in it as subroutines.

Algorithm DP$(G, R, k, q, \mathcal{D})$
Input: A graph G, a vertex set $R \subseteq V(G)$, two non-negative integers k and q, where $k \leq q$, and a tree decomposition $\mathcal{D}$ of G of width q.
Output: An answer whether (G, R, k) is a YES-instance of Π or not.
Running time: $2^{2^{O(q \cdot \log q)}} \cdot n$.

Algorithm **DP** is based on dynamic programming on tree decompositions of graphs. The technical details are omitted in this extended abstract.

Algorithm Compass(G, q)
Input: A planar graph G and a non-negative integer q.
Output: Either a tree decomposition of G of width at most $18q$ or a subdivided wall W of G of height q and a tree decomposition $\mathcal{D}$ of the compass K_W of W of width at most $18q$.
Running time: $2^{q^{O(1)}} \cdot n$.

We describe algorithm **Compass** in Subsection 3.1.

Algorithm concentric_cycles(G, R, k, q, W)
Input: A planar graph G, a set $R \subseteq V(G)$, a non-negative integer k, and a subdivided wall W of G of height at least $392k^2 + 40k$.
Output: Either a problem-irrelevant vertex v or a sequence $\mathcal{C} = \{C_1, C_2, \ldots, C_{98k+2}\}$ of concentric cycles of G, with the following properties:
(1) $C_1 \cap R \neq \emptyset$.
(2) The set R is $32k$-dense in $\mathcal{C}$.
(3) There exists a sequence $\mathcal{W}$ of $2k + 1$ paths in K_W such that $(\mathcal{C}, \mathcal{W})$ is a $(98k + 2, 2k + 1)$-railed annulus.
Running time: $O(n)$.

We describe Algorithm **concentric_cycles** in Subsection 3.2. We now use the above three algorithms to describe the main algorithm of this paper that is the following.

Algorithm Planar_Annotated_Cyclability(G, R, k)
Input: A planar graph G, a set $R \subseteq V(G)$, and a non-negative integer k.
Output: An answer whether (G, R, k) is a YES-instance of Π, or not.
Running time: $2^{2^{O(k^2 \log k)}} \cdot n^2$.

[*Step 1.*] Let $r = 98k^2 + 2k$, $y = 16k$, and $q = 2y + 4r$. If **Compass**(G, q) returns a tree decomposition of G of width $w = 18q$, then return **DP**(G, R, k, w) and stop. Otherwise, the algorithm **Compass**(G, q) returns a subdivided wall W of G of height q and a tree decomposition $\mathcal{D}$ of the compass K_W of W of width at most w.

[*Step 2.*] If the algorithm **concentric_cycles**(G, R, k, q, W) returns a problem-irrelevant vertex v, then return **Planar_Annotated_Cyclability**$(G \setminus v, R \setminus v, k)$ and stop. Otherwise, it returns a sequence $\mathcal{C} = \{C_1, C_2, \ldots, C_r\}$ of concentric cycles of G with the properties (1)–(3).

[*Step 3.*] For every $i \in \{1, \ldots, r - 98k - 2\}$ let w_i be a vertex in $\hat{A}_{i+k, i+33 \cdot k} \cap R$ (this vertex exists as, from property (2), R is $32k$-dense in $\mathcal{C}$), let $R_i = (R \cap V(\hat{C}_i)) \cup \{w_i\}$, and let $\mathcal{D}_i$ be a tree decomposition of $\hat{C}_i$ of width at most w – this tree decomposition can be constructed in linear time from $\mathcal{D}$ as each $\hat{C}_i$ is a subgraph of K_W.

[*Step 4.*] If, for some $i \in \{1, \ldots, r - 98k - 2\}$, the algorithm **DP**$(\hat{C}_i, R_i, k, q, \mathcal{D}_i)$ returns a negative answer, then return a negative answer and stop. Otherwise return **Planar_Annotated_Cyclability**$(G, R \setminus v, k)$ where v is some vertex of $\hat{C}_1$ that belongs in R (the choice of v is possible due to property (1)).

Proof of Lemma 2. The only non-trivial step in the above algorithm is Step 4. Its correctness follows from Lemma 6, presented in Subsection 3.3.

We now proceed with the analysis of the running time of the algorithm. Observe first that the call of **Compass**(G, q) in Step 1 takes $2^{k^{O(1)}} \cdot n$ steps and, in case, a tree decomposition is returned, the **DP** requires $2^{2^{O(k^2 \log k)}} \cdot n$ steps. For Step 2, the algorithm **concentric_cycles** takes $O(n)$ steps and if it returns a problem-irrelevant vertex, then the whole algorithm is applied again for a graph with one vertex less. Suppose now that Step 2 returns a sequence $\mathcal{C}$ of concentric cycles of G with the properties (1)–(3). Then the algorithm **DP** is called $O(k^2)$ times and this takes in total $2^{2^{O(k^2 \log k)}} \cdot n$ steps. After that, the algorithm either concludes to a negative answer or is called again with one vertex less in the set R. In both cases where the algorithm is called again we have that the quantity $|V(G)| + |R|$ is becoming smaller. This means that the recursive calls of the algorithm cannot be more than $2n$. Therefore the total running time is bounded by $2^{2^{O(k^2 \log k)}} \cdot n^2$ as required. □

3.1 The Algorithm Compass

Before we start the description of algorithm **Compass** we present a result that follows by Proposition 1, the algorithms in [21] and [5], and the fact that finding a subdivision of a planar k-vertex graph H that has maximum degree 3 in a graph G can be done, using dynamic programming, in $2^{O(k \cdot \log k)} \cdot n$ steps (see also [1]).

Lemma 3. *There exists an algorithm A_1 that, given a graph G and an integer h, outputs either a tree decomposition of G of width at most $9h$ or a subdivided wall of G of height h. This algorithm runs in $2^{h^{O(1)}} \cdot n$ steps.*

Description of Algorithm **Compass**. Let $q' = 9q$. We use the routine A_2 that receives as input a subdivided wall W of G with height equal to some even number h and outputs a subdivided wall W' of G such that with W' has height $h/2$ and $|V(K_{W'})| \leq |V(G)|/4$. A_2 uses the fact that, in W, there are 4 vertex-disjoint subdivided subwalls of W of height $h/2$. Among them, A_2 outputs the one with the minimum number of vertices and this can be done in $O(n)$ steps. The algorithm **Compass** uses as subroutines the routine A_2 and the algorithm A_1 of Lemma 3.1.

```
Algorithm Compass($G, q$)
[Step 1.] if $A_1(G, 2q)$ outputs a tree decomposition $\mathcal{D}$ of $G$ with
          width at most $2q'$ then return $\mathcal{D}$,
          otherwise it outputs a subdivided wall $W$ of $G$ of height $2q$
[Step 2.] Let $W' = A_2(W)$
          if $A_1(K_{W'}, 2q)$ outputs a tree decomposition $\mathcal{D}$ of
            $K_{W'}$ with width at most $2q'$ then return $W'$ and $\mathcal{D}$,
            otherwise $W \leftarrow W'$ and go to Step 2.
```

Notice that, if A terminates after the first execution of step 1, then it outputs a tree decomposition of G of width at most $2q'$. Otherwise, the output is a subdivided wall W' of height k in G and a tree decomposition of $K_{W'}$ of width at most $2q'$ (notice that as long as this is not the case, the algorithm keeps returning to step 2). The aplication of routine A_2 ensures that the number vertices of every new K_W is at least four times smaller than the one of the previous one. Therefore, the i-th call of the the algorithm A_1 requires $\mathcal{O}(2^{h^{O(1)}} \cdot \frac{n}{2^{2(i-1)}})$ steps. As $\sum_{i=0}^{\infty} \frac{1}{2^{2i}} = O(1)$, algorithm **Compass** has the same running time as algorithm A_1.

3.2 The Algorithm Concentric_Cycles

We require first the following two lemmata, The first one is strongly based on the combinatorial Lemma 9 that is the main result of Section 4.

Lemma 4 **($\star$).** *Let (G, R, k) be an instance of Π and let $\mathcal{C} = \{C_1, \ldots, C_r\}$ be a sequence of concentric cycles in G such that $V(\hat{C}_r) \cap R = \emptyset$. If $r \geq 16 \cdot k$, then all vertices in $\hat{C}_1$ are problem-irrelevant.*

Lemma 5 **($\star$).** *Let y, r, q, z be positive integers such that $y + 1 \leq z \leq r$, G be a graph embedded on $\mathbb{S}_0$ and let $R \subseteq V(G)$ be the set of annotated vertices of G. Given a subdivided wall W in G of height $h = 2 \cdot \max\{y, \lceil \frac{q}{8} \rceil\} + 4r$, then either G contains a sequence $\mathcal{C}' = \{C'_1, C'_2, \ldots, C'_y\}$ of concentric cycles such that $V(\hat{C}'_y) \cap R = \emptyset$ or a sequence $\mathcal{C} = \{C_1, C_2, \ldots, C_r\}$ of concentric cycles such that:*

1. *$\bar{C}_1 \cap R \neq \emptyset$.*

2. R is z-dense in $\mathcal{C}$.
3. There exists a collection $\mathcal{W}$ of q paths in K_W, such that $(\mathcal{C}, \mathcal{W})$ is a (r,q)-railed annulus in G.

Moreover, a sequence $\mathcal{C}'$ or $\mathcal{C}$ of concentric cycles as above can be constructed in $O(n)$ steps.

Description of algorithm **concentric_cycles** This algorithm first applies the algorithm of Lemma 5 for $y = 16k$, $r = 98k^2 + 2k$, $q = 2k+1$, and $z = 32k$. If the output is a sequence $\mathcal{C}' = \{C_1', C_2', \ldots, C_y'\}$ of concentric cycles such that $V(\hat{C}_y') \cap R = \emptyset$ then, the algorithm returns a vertex w of $\hat{C}_1'$. As $V(\hat{C}_r) \cap R = \emptyset$, Lemma 4, implies that w is problem-irrelevant. If the output is a sequence $\mathcal{C}$ the it remains to observe that conditions *1–3* match the specifications of algorithm **concentric_cycles**.

3.3 Correctness of Algorithm Planar_Annotated_Cyclability

As mentioned in the proof of Lemma 2, the main step – [*step 4*] – of algorithm **Planar_Annotated_Cyclability** is based in Lemma 6 bellow.

Lemma 6 ($\star$). *Let (G, R, k) be an instance of problem Π and let $b = 98k + 2$ and $r = 98k^2 + 2k$. Let also $(\mathcal{C}, \mathcal{W})$ be an $(r, 2k+1)$-railed annulus in G, where $\mathcal{C} = \{C_1, \ldots C_r\}$ is a sequence of concentric cycles such that $\hat{C}_1$ contains some vertex $v \in R$ and that, R is $32k$-dense in $\mathcal{C}$. For every $i \in \{1, \ldots, r-b\}$ let $R_i = (R \cap V(\hat{C}_i)) \cup \{w_i\}$, where $w_i \in V(\hat{A}_{i+k+1,33k+i+1}) \cap R$. If $(\hat{C}_{i+b}, R_i, k)$ is a* NO*-instance of* Π, *for some $i \in \{1, \ldots, r-b\}$, then (G, R, k) is a* NO*-instance of* Π. *Otherwise vertex v is* color-irrelevant.

The proof of Lemma 6 is strongly based on Lemma 4.

4 Vital Cyclic Linkages

Tight sequences. A sequence $\mathcal{C} = \{C_1, \ldots, C_r\}$ of concentric cycles of G is *tight* in G, if

- C_1 is *surface minimal*, i.e., there is no closed disk D of $\mathbb{S}$ that is properly contained in $\bar{C}_1$ and whose boundary is a cycle of G;
- for every $i \in \{1, \ldots, r-1\}$, there is no closed disk D such that $\bar{C}_i \subset D \subset \bar{C}_{i+1}$ and such that the boundary of D is a cycle of G.

Graph Linkages. Let G be a graph. A *graph linkage* in G is a pair $\mathcal{L} = (H, T)$ such that H is a subgraph of G without isolated vertices and T is a subset of the vertices of H, called *terminals* of $\mathcal{L}$, such that every vertex of H with degree different than 2 is contained in T. Set $\mathcal{P}(\mathcal{L})$, which we call *path set of* the graph linkage $\mathcal{L}$, contains all paths of H whose endpoints are in T and do not have any other vertex in T. The *pattern* of L is the graph

$$(T, \{\{s,t\} \mid \mathcal{P}(\mathcal{L}) \text{ contains a path from } s \text{ to } t \text{ in } H\}).$$

Two graph linkages of G are *equivalent* if they have the same pattern and are *isomorphic* if their patterns are isomorphic. A graph linkage $\mathcal{L} = (H, T)$ is called *weakly vital* (reps. *strongly vital*) in G if $V(H) = V(G)$ and there is no other equivalent (resp. isomorphic) graph linkage that is different from $\mathcal{L}$. Clearly, if a graph linkage $\mathcal{L}$ is strongly vital then it is also weakly vital. We call a graph linkage $\mathcal{L}$ *linkage* if its pattern has maximum degree 1 (i.e., it consists of a collection of paths). We call a graph linkage $\mathcal{L}$ *cyclic linkage* if its pattern is a cycle.

CGL-configurations. Let G be a graph embedded on the sphere $\mathbb{S}_0$. Then we say that a pair $\mathcal{Q} = (\mathcal{C}, \mathcal{L})$ is a *CGL-configuration* of *depth* r if $\mathcal{C} = \{C_1, \dots, C_r\}$ is a sequence of concentric cycles in G, $\mathcal{L} = (H, T)$ is a graph linkage in G, and $T \cap V(\hat{C}_r) = \emptyset$, i.e., all vertices in the terminals of $\mathcal{L}$ are outside $\hat{C}_r$. The *penetration* of $\mathcal{L}$ in $\mathcal{C}$, $p_{\mathcal{C}}(\mathcal{L})$, is the number of cycles of $\mathcal{C}$ that are intersected by the paths of $\mathcal{L}$ (when $\mathcal{L} = (C, S)$ is cyclic we will sometimes refer to the penetration of $\mathcal{L}$ as the penetration of cycle C). We say that $\mathcal{Q}$ is *touch-free* if for every path $P \in \mathcal{L}$, the number of connected components of $P \cap C_r$ is not 1.

Cheap graph linkages. Let G be a graph embedded on the sphere $\mathbb{S}_0$, let $\mathcal{C} = \{C_1, \dots, C_r\}$ be a sequence of cycles in G, and let $\mathcal{L} = (H, T)$ be a graph linkage where $T \subseteq V(G \setminus \hat{C}_r)$(notice that $(\mathcal{C}, \mathcal{L})$ is a CGL-configuration). We define the function c that matches graph linkages of G to positive integers such that $c(\mathcal{L}) = |E(\mathcal{L}) \setminus \bigcup_{i \in \{1,\dots,r\}} E(C_i)|$.

A graph linkage $\mathcal{L}$ of G is $\mathcal{C}$*-strongly cheap* (resp. $\mathcal{C}$*-weakly cheap*), if $T(\mathcal{L}) \cap \hat{C}_r = \emptyset$ and there is no other isomorphic (resp. equivalent) graph linkage $\mathcal{L}'$ such $c(\mathcal{L}) > c(\mathcal{L}')$. Obviously if $\mathcal{L}$ is $\mathcal{C}$-strongly cheap then it is also $\mathcal{C}$-weakly cheap.

The proof of the next lemma is based on a suitable adaptation of the results in [2] about weakly vital linkages to strongly vital cyclic linkages.

Lemma 7 ($\star$). *Let G be a graph embedded on the sphere $\mathbb{S}_0$ that is the union of $r \geq 2$ concentric cycles $\mathcal{C} = \{C_1, \dots, C_r\}$ and one more cycle C of G. Assume that $\mathcal{C}$ is tight in G, $T \cap V(\hat{C}_r) = \emptyset$ and the cyclic linkage $\mathcal{L} = (C, T)$ is strongly vital in G. Then $r \leq 16 \cdot |T| - 1$.*

A corollary of Lemma 7 with independent combinatorial interest is the following.

Corollary 1. *If a plane graph G contains a strongly vital cyclic linkage $\mathcal{L} = (C, T)$, then $\mathbf{tw}(G) = O(|T|^{3/2})$.*

Notice that, according to what is claimed in [2], we cannot restate the above corollary for weakly vital linkages, unless we change the bound to be an exponential one. That way, the fact that treewidth is (unavoidably, due to [2]) exponential to the number of terminals for (weakly) vital linkages is caused by the fact that the ordering of the terminals is predetermined.

Lemma 8 ($\star$). *Let G be a graph embedded on the sphere $\mathbb{S}_0$ that is the union of r concentric cycles $\mathcal{C} = \{C_1, \dots, C_r\}$ and a hamiltonian cycle C of G. Let also $T \cap V(\hat{C}_r) = \emptyset$. If $\mathcal{L} = (C, T)$ is $\mathcal{C}$-strongly cheap then $\mathcal{L}$ is a strongly vital cyclic linkage in G.*

We are now able to prove the main combinatorial result of this paper.

Lemma 9. *Let G be a plane graph with some sequence of concentric cycles $\mathcal{C} = \{C_1, \ldots, C_r\}$. Let also $\mathcal{L} = (C, T)$ be a cyclic linkage of G where $T \cap V(\hat{C}_r) = \emptyset$. If $\mathcal{L}$ is $\mathcal{C}$- strongly cheap then the penetration of $\mathcal{L}$ in $\mathcal{C}$ is at most $r \leq 16 \cdot |T| - 1$.*

Proof. Suppose that some path in $\mathcal{P}(\mathcal{L})$ intersects $16 \cdot |T|$ cycles in the set $\mathcal{C}^* = \{C_{r-16\cdot|T|+1}, \ldots, C_r\}$. Let G' be the graph obtained by $C \cup \bigcup\mathcal{C}^*$ after dissolving all vertices not in T that have degree 2 and let $\mathcal{L}' = (C', T)$ be the linkage of G' obtained from $\mathcal{L}$ if we dissolve the same vertices in the paths of $\mathcal{L}$. Similarly, by dissolving vertices of degree 2 in the cycles of $\mathcal{C}^*$ we obtain a new sequence of concentric cycles that, for notational convenience, we denote by $\mathcal{C}' = \{C_1, \ldots, C_{r'}\}$, where $r' = 16 \cdot |T|$. $\mathcal{L}'$ is $\mathcal{C}'$-strongly cheap because $\mathcal{L}$ is $\mathcal{C}$-strongy cheap. Notice that C' is a Hamiltonian cycle of G' and, from Lemma 8, $\mathcal{L}'$ is a strongly vital cyclic linkage of G'. We also assume that $\mathcal{C}'$ is tight (otherwise replace it by a tight one and observe that, by its uniqueness, $\mathcal{L}'$ will be cheap to this new one as well). As $\mathcal{L}'$ is $\mathcal{C}'$-strongly cheap and $\mathcal{C}'$ is tight, from Lemma 7, $r' \leq 16 \cdot |T| - 1$, a contradiction. □

5 Discussion

Notice that the bounds in the running time of our algorithm depend heavily on Lemma 4, which makes strong use of the results in [2] (and also [3]) that, in turn, are heavily based on planarity. It is an interesting task to prove the same line of results with the same (optimal) bounds for graphs of bounded genus. In fact, using directly the general results of [26] and [19] we can easily adapt the techniques of this paper to prove that Cyclability, when parameterized by $k + g$ is fixed parameter tractable (here g is the genus of the input graph). We are also convinced that this can be extended to more general minor free graph classes, however, this should be a much more complicated and technical task.

Notice that we have no proof that Cyclability is in NP. The definition of the problem classifies it directly in Π_2^{P}. This prompts us to conjecture the following:

Conjecture 1. Cyclability is Π_2^{P}-complete.

Moreover, while we have proved that Cyclability is co-W[1]-hard, we have no evidence on which level of the parameterized complexity hierarchy it belongs (lower than the XP class). We find it an intriguing question whether there is some $i \geq 1$ for which Cyclability is W[i]-complete (or co-W[i]-complete).

Clearly, a challenging question is whether the, double exponential, parametric dependance of our FPT-algorithm can be improved. We believe that this is not possible and we suspect that this issue might be related with Conjecture 1. Also we find it an interesting question whether the standard parameterization of Cyclability (or its complements) belongs in some class of the W-hierarchy or "higher levers" parameterized complexity classes are required for its classification. A different direction that we find interesting is the approximation complexity of the problem for which no result appear to appear up to this moment.

References

1. Adler, I., Dorn, F., Fomin, F.V., Sau, I., Thilikos, D.M.: Fast minor testing in planar graphs. In: de Berg, M., Meyer, U. (eds.) ESA 2010, Part I. LNCS, vol. 6346, pp. 97–109. Springer, Heidelberg (2010)
2. Adler, I., Kolliopoulos, S.G., Krause, P.K., Lokshtanov, D., Saurabh, S., Thilikos, D.: Tight bounds for linkages in planar graphs. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part I. LNCS, vol. 6755, pp. 110–121. Springer, Heidelberg (2011)
3. Adler, I., Kolliopoulos, S.G., Thilikos, D.M.: Planar disjoint-paths completion. In: Marx, D., Rossmanith, P. (eds.) IPEC 2011. LNCS, vol. 7112, pp. 80–93. Springer, Heidelberg (2012)
4. Aldred, R.E., Bau, S., Holton, D.A., McKay, B.D.: Cycles through 23 vertices in 3-connected cubic planar graphs. Graphs and Combinatorics 15(4), 373–376 (1999)
5. Bodlaender, H.L.: A linear-time algorithm for finding tree-decompositions of small treewidth. SIAM J. Comput. 25(6), 1305–1317 (1996)
6. Dirac, G.A.: In abstrakten Graphen vorhandene vollständige 4-Graphen und ihre Unterteilungen. Math. Nachr. 22, 61–85 (1960)
7. Downey, R.G., Fellows, M.R.: Parameterized Complexity. Springer, New York (1999)
8. Downey, R., Fellows, M.: Fixed-parameter tractability and completeness. III. Some structural aspects of the W hierarchy. In: Complexity Theory, pp. 191–225. Cambridge Univ. Press, Cambridge (1993)
9. Downey, R.G., Fellows, M.R.: Fixed-parameter tractability and completeness. In: 21st Manitoba Conference on Numerical Mathematics and Computing, Winnipeg, MB, vol. 87, pp. 161–178 (1992)
10. Downey, R.G., Fellows, M.R.: Fixed-parameter tractability and completeness. I. Basic results. SIAM J. Comput. 24(4), 873–921 (1995)
11. Downey, R.G., Fellows, M.R.: Fixed-parameter tractability and completeness II: On completeness for $W[1]$. Theoretical Computer Science 141(1-2), 109–131 (1995)
12. Flandrin, E., Li, H., Marczyk, A., Woźniak, M.: A generalization of dirac's theorem on cycles through K vertices in K-connected graphs. Discrete Mathematics 307(7), 878–884 (2007)
13. Flum, J., Grohe, M.: Parameterized Complexity Theory. Springer (2006)
14. Fomin, F.V., Golovach, P.A., Thilikos, D.M.: Contraction obstructions for treewidth. J. Comb. Theory, Ser. B 101(5), 302–314 (2011)
15. Garey, M.R., Johnson, D.S.: Computers and intractability. W. H. Freeman and Co. (1979) a guide to the theory of NP-completeness, A Series of Books in the Mathematical Sciences
16. Grötschel, M.: Hypohamiltonian facets of the symmetric travelling salesman polytope. Zeitschrift für Angewandte Mathematik und Mechanik 58, 469–471 (1977)
17. Gu, Q.-P., Tamaki, H.: Improved bounds on the planar branchwidth with respect to the largest grid minor size. In: Cheong, O., Chwa, K.-Y., Park, K. (eds.) ISAAC 2010, Part II. LNCS, vol. 6507, pp. 85–96. Springer, Heidelberg (2010)
18. Kawarabayashi, K.I.: An improved algorithm for finding cycles through elements. In: Lodi, A., Panconesi, A., Rinaldi, G. (eds.) IPCO 2008. LNCS, vol. 5035, pp. 374–384. Springer, Heidelberg (2008)
19. Kawarabayashi, K., Wollan, P.: A shorter proof of the graph minor algorithm: the unique linkage theorem. In: FOCS 2008 49th Annual IEEE Symposium on Foundations of Computer Science, pp. 771–780 (2008)
20. Niedermeier, R.: Invitation to fixed-parameter algorithms. Habilitation thesis (September 2002)

21. Perkovic, L., Reed, B.A.: An improved algorithm for finding tree decompositions of small width. Int. J. Found. Comput. Sci. 11(3), 365–371 (2000)
22. Plummer, M., Győri, E.: A nine vertex theorem for 3-connected claw-free graphs. Studia Scientiarum Mathematicarum Hungarica 38(1), 233–244 (2001)
23. Robertson, N., Seymour, P.: Graph Minors. XIII. irrelevant vertices in linkage problems. Journal of Combinatorial Theory, Series B 102(2), 530–563 (2012), `http://www.sciencedirect.com/science/article/pii/S0095895611000724`
24. Robertson, N., Seymour, P.D.: Graph Minors. X. Obstructions to Tree-decomposition. J. Combin. Theory Series B 52(2), 153–190 (1991)
25. Robertson, N., Seymour, P.D.: Graph Minors. XIII. The disjoint paths problem. J. Combin. Theory, Ser. B 63(1), 65–110 (1995)
26. Robertson, N., Seymour, P.D.: Graph Minors. XXI. Graphs with unique linkages. J. Combin. Theory Ser. 99(3), 583–616 (2009), `http://dx.doi.org/10.1016/j.jctb.2008.08.003`
27. Watkins, M., Mesner, D.: Cycles and connectivity in graphs. Canad. J. Math. 19, 1319–1328 (1967)

Dimension Reduction via Colour Refinement

Martin Grohe[1,*], Kristian Kersting[2,**], Martin Mladenov[2], and Erkal Selman[1]

[1] RWTH Aachen University
[2] TU Dortmund University

Abstract. Colour refinement is a basic algorithmic routine for graph isomorphism testing, appearing as a subroutine in almost all practical isomorphism solvers. It partitions the vertices of a graph into "colour classes" in such a way that all vertices in the same colour class have the same number of neighbours in every colour class. There is a tight correspondence between colour refinement and fractional isomorphisms of graphs, which are solutions to the LP relaxation of a natural ILP formulation of graph isomorphism.

We introduce a version of colour refinement for matrices and extend existing quasilinear algorithms for computing the colour classes. Then we generalise the correspondence between colour refinement and fractional automorphisms and develop a theory of fractional automorphisms and isomorphisms of matrices.

We apply our results to reduce the dimensions of systems of linear equations and linear programs. Specifically, we show that any given LP L can efficiently be transformed into a (potentially) smaller LP L' whose number of variables and constraints is the number of colour classes of the colour refinement algorithm, applied to a matrix associated with the LP. The transformation is such that we can easily (by a linear mapping) map both feasible and optimal solutions back and forth between the two LPs. We demonstrate empirically that colour refinement can indeed greatly reduce the cost of solving linear programs.

1 Introduction

Colour refinement (a.k.a. "naive vertex classification" or "colour passing") is a basic algorithmic routine for graph isomorphism testing. It iteratively partitions, or colours, the vertices of a graph according to an iterated degree sequence: initially, all vertices get the same colour, and then in each round of the iteration two vertices that so far have the same colour get different colours if for some colour c they have a different number of neighbours of colour c. The iteration

[*] Martin Grohe and Erkal Selman were supported by the German Research Foundation DFG Koselleck grant GR 1492/14-1.

[**] Kristian Kersting was supported by the Fraunhofer ATTRACT fellowship STREAM and by the European Commission under contract number FP7-248258-First-MM. Martin Mladenov and Kristian Kersting were supported by the German Research Foundation DFG, KE 1686/2-1, within the SPP 1527, and the German-Israeli Foundation for Scientific Research and Development, 1180-218.6/2011.

stops if in some step the partition remains unchanged; the resulting partition is known as the *coarsest equitable partition* of the graph. By refining the partition asynchronously using Hopcroft's strategy of "processing the smaller half" (for DFA-minimisation [8]), the coarsest equitable partition of a graph can be computed very efficiently, in time $O((n+m)\log n)$ [5,11] (also see [2] for a matching lower bound). A beautiful result due to Tinhofer [14], Ramana, Scheinerman, and Ullman [12] and Godsil [6] establishes a tight correspondence between equitable partitions of a graph and fractional automorphisms, which are solutions to the LP relaxation of a natural ILP formulation of graph isomorphism.

In this paper, we introduce a version of colour refinement for matrices (to be outlined soon) and develop a theory of equitable partitions and fractional automorphisms and isomorphisms of matrices. A somewhat surprising application of the theory is a method to reduce the dimensions of systems of linear equations and linear programs.

When applied in the context of graph isomorphism testing, the goal of colour refinement is to partition the vertices of a graph as finely as possible; ideally, one would like to compute the partition of the vertices into the orbits of the automorphism group of the graph. In this paper, our goal is to partition the rows and columns of a matrix as coarsely as possible. We show that by "factoring" a matrix associated with a system of linear equations or a linear program through an "equitable partition" of the variables and constraints, we obtain a smaller system or LP equivalent to the original one, in the sense that feasible and optimal solutions can be transferred back and forth between the two via linear mappings that we can compute efficiently. Hence we can use colour refinement as a simple and efficient preprocessing routine for linear programming, transforming a given linear program into an equivalent one in a lower dimensional space and with fewer constraints. We demonstrate the effectiveness of this method experimentally.

Due to the ubiquity of linear programming, our method potentially has a wide range of applications. Of course not all linear programs show the regularities needed by our method to be effective. Yet some do. This work grew out of applications in machine learning, or more specifically, inference problems in graphical models. Actually, many problems arising in a wide variety of other fields such as semantic web, network communication, computer vision, and robotics can also be modelled using graphical models. The models often have inherent regularities, which are not exploited by classical inference approaches such as loopy belief propagation. Symmetry-aware approaches, see e.g. [13,9,1,4], run (a modified) loopy belief propagation on the quotient model of the (fractional) automorphisms of the graphical model and have been proven successful in several applications such as link prediction, social network analysis, satisfiability and boolean model counting problems. Some of these approaches have natural LP formulations, and the method proposed here is a strengthening of the symmetry-aware approaches applied by the second and third author (jointly with Ahmadi) in [10].

Our work is related to other methods of symmetry reduction for linear programs. Most notably, Bödi, Grundhöfer and Herr [3] proposed a method of symmetry reduction for linear programs, which, however, requires to compute the automorphism group of a linear program. In the full version of this paper [7], we show that, in some sense, our method subsumes that of [3].

Colour Refinement on Matrices

Consider a matrix $A \in \mathbb{R}^{V \times W}$.[1] We iteratively compute partitions (or colourings) $\mathcal{P}_i$ and $\mathcal{Q}_i$ of the rows and columns of A, that is, of the sets V and W. We let $\mathcal{P}_0 = \{V\}$ and $\mathcal{Q}_0 = \{W\}$ be the trivial partitions. To define $\mathcal{P}_{i+1}$, we put two rows v, v' in the same class if they are in the same class of $\mathcal{P}_i$ and if for all classes Q of $\mathcal{Q}_i$,

$$\sum_{w \in Q} A_{vw} = \sum_{w \in Q} A_{v'w}. \tag{1.1}$$

Similarly, to define $\mathcal{Q}_{i+1}$, we put two columns w, w' in the same class if they are in the same class of $\mathcal{Q}_i$ and if for all classes P of $\mathcal{P}_i$,

$$\sum_{v \in P} A_{vw} = \sum_{v \in P} A_{vw'}. \tag{1.2}$$

Clearly, for some $i \leq |V| + |W|$ we have $(\mathcal{P}_i, \mathcal{Q}_i) = (\mathcal{P}_{i+1}, \mathcal{Q}_{i+1}) = (\mathcal{P}_j, \mathcal{Q}_j)$ for all $j \geq i$. We let $(\mathcal{P}_\infty, \mathcal{Q}_\infty) := (\mathcal{P}_i, \mathcal{Q}_i)$. To see that this is a direct generalisation of colour refinement on graphs, suppose that A is a 0-1-matrix, and view it as the adjacency matrix of a bipartite graph B_A with vertex set $V \cup W$ and edge set $\{vw \mid A_{vw} \neq 0\}$. Then the coarsest equitable partition of A is equal to the partition obtained by running colour refinement on B_A starting from the partition $\{V, W\}$.

Adopting Paige and Tarjan's [11] algorithm for colour refinement on graphs, we obtain an algorithm that, given a sparse representation of a matrix A, computes $(\mathcal{P}_\infty, \mathcal{Q}_\infty)$ in time $O((n + m) \log n)$, where $n = |V| + |W|$ and m is the total bitlength of all nonzero entries of A. Details on the algorithm can be found in the full version [7].

Slightly abusing terminology, we say that a *partition* of a matrix $A \in \mathbb{R}^{V \times W}$ is a pair $(\mathcal{P}, \mathcal{Q})$ of partitions of V, W, respectively. Such a partition partitions the matrix into "combinatorial rectangles". A partition $(\mathcal{P}, \mathcal{Q})$ of A is *equitable* if for all $P \in \mathcal{P}$, $Q \in \mathcal{Q}$ and all $v, v' \in P$, $w, w' \in Q$ equations (1.1) and (1.2) are satisfied. It is easy to see that the partition $(\mathcal{P}_\infty, \mathcal{Q}_\infty)$ computed by colour refinement is the *coarsest* equitable partition, in the sense that it is equitable and all other equitable partitions refine it.

The key result that enables us to apply colour refinement to reduce the dimensions of linear programs is a correspondence between equitable partitions and *fractional automorphisms* of a matrix. We view an *automorphism* of a matrix

[1] We find it convenient to index the rows and columns of our matrices by elements of finite sets V, W, respectively, which we assume to be disjoint. $\mathbb{R}^{V \times W}$ denotes the set of matrices with real entries and row and column indices from V, W, respectively. The order of the rows and columns of a matrix is irrelevant for us. We denote the entries of a matrix $A \in \mathbb{R}^{V \times W}$ by A_{vw}.

$A \in \mathbb{R}^{V \times W}$ as a pair of permutations of the rows and columns that leaves the matrix invariant, or equivalently, a pair $(X, Y) \in \mathbb{R}^{V \times V} \times \mathbb{R}^{W \times W}$ of permutation matrices such that

$$XA = AY. \tag{1.3}$$

A *fractional automorphism* of A is a pair $(X, Y) \in \mathbb{R}^{V \times V} \times \mathbb{R}^{W \times W}$ of doubly stochastic matrices satisfying (1.3). We shall prove (Theorem 3.1) that every equitable partition of a matrix yields a fractional automorphism and, conversely, every fractional automorphism (X, Y) yields an equitable partition. The classes of this equitable partition are simply the strongly connected components of the directed graphs underlying the square matrices X, Y. This basic result is the foundation for everything else in this paper.

We proceed to studying *fractional isomorphisms* between matrices. However, it turns out that fractional isomorphism is still too fine as an equivalence relation that captures the solvability of linear programs. We introduce a coarser equivalence relation between matrices that we call *partition equivalence.* The idea is that two matrices are equivalent if they have "isomorphic" equitable partitions. We prove that two linear programs with associated matrices that are partition equivalent are equivalent in the sense that there are two linear mappings that map the feasible solutions of one LP to the feasible solutions of the other, and these mappings preserve optimality.

Application to Linear Programming

Every matrix A is partition equivalent to a matrix $[A]$ obtained by "factoring" A through its coarsest equitable partition; we call $[A]$ the *core factor* of A. We can repeat this factoring process and go to matrices $[[A]], [[[A]]]$, et cetera, until we finally arrive at the *iterated core factor* $[\![A]\!]$. Now suppose that A is associated with an LP L, then $[\![A]\!]$ is associated with an LP $[\![L]\!]$. To solve L, we compute $[\![L]\!]$, which we can do efficiently using colour refinement. The colour refinement procedure also yields the matrices that we need to translate between the solution spaces of L and $[\![L]\!]$. Then we solve $[\![L]\!]$ and translate the solution back to a solution of L.

Example 1.1. We consider a linear program in standard form:

$$\begin{array}{rl} \min & c^{\mathrm{t}} x \\ \text{subject to} & Ax = b,\ x \geq 0, \end{array} \tag{L}$$

for the matrix A and the vectors b, c shown in Fig. 1.1. We combine A, b, c in a matrix $\widetilde{A}$ shown in Fig. 1.2. The lines subdividing the matrix indicate the coarsest equitable partition. As the core factor of $\widetilde{A}$ we obtain the matrix

$$[\widetilde{A}] = \left(\begin{array}{cc|cc|c} 3 & 1 & 0 & 2 & 1 \\ 1 & 3 & 2 & 0 & 1 \\ \hline 6 & 6 & 2 & 2 & \infty \end{array}\right).$$

$$A = \begin{pmatrix} 3 & -1 & 1 & 1/4 & 1/4 & 1/4 & 1/4 & 0 & 0 & 3 & -2 & 1/2 & 1/2 \\ -1 & 1 & 3 & 1/4 & 1/4 & 1/4 & 1/4 & 0 & 0 & -2 & 3 & 1/2 & 1/2 \\ 1 & 3 & -1 & 1/4 & 1/4 & 1/4 & 1/4 & 0 & 0 & 1/2 & 1/2 & 1/2 & 1/2 \\ 0 & 1/3 & 2/3 & 0 & 3/2 & 0 & 3/2 & 2 & 0 & 1 & 0 & -1 & 0 \\ 1/3 & 1/3 & 1/3 & 3/2 & 0 & 3/2 & 0 & 2 & 0 & 0 & 1 & 0 & -1 \\ 1/3 & 1/3 & 1/3 & 0 & 3/2 & 0 & 3/2 & 0 & 2 & -1 & 0 & 1 & 0 \\ 2/3 & 1/3 & 0 & 3/2 & 0 & 3/2 & 0 & 0 & 2 & 0 & -1 & 0 & 1 \end{pmatrix}, \quad b = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}, \quad c = \begin{pmatrix} 2 \\ 2 \\ 2 \\ 3/2 \\ 3/2 \\ 3/2 \\ 3/2 \\ 1 \\ 1 \\ 1/2 \\ 1/2 \\ 1/2 \\ 1/2 \end{pmatrix}$$

Fig. 1.1. The LP of Example 1.1

$$\widetilde{A} = \left(\begin{array}{ccc|cccc|cc|cccc|c} 3 & -1 & 1 & 1/4 & 1/4 & 1/4 & 1/4 & 0 & 0 & 3 & -2 & 1/2 & 1/2 & 1 \\ -1 & 1 & 3 & 1/4 & 1/4 & 1/4 & 1/4 & 0 & 0 & -2 & 3 & 1/2 & 1/2 & 1 \\ 1 & 3 & -1 & 1/4 & 1/4 & 1/4 & 1/4 & 0 & 0 & 1/2 & 1/2 & 1/2 & 1/2 & 1 \\ \hline 0 & 1/3 & 2/3 & 0 & 3/2 & 0 & 3/2 & 2 & 0 & 1 & 0 & -1 & 0 & 1 \\ 1/3 & 1/3 & 1/3 & 3/2 & 0 & 3/2 & 0 & 2 & 0 & 0 & 1 & 0 & -1 & 1 \\ 1/3 & 1/3 & 1/3 & 0 & 3/2 & 0 & 3/2 & 0 & 2 & -1 & 0 & 1 & 0 & 1 \\ 2/3 & 1/3 & 0 & 3/2 & 0 & 3/2 & 0 & 0 & 2 & 0 & -1 & 0 & 1 & 1 \\ \hline 2 & 2 & 2 & 3/2 & 3/2 & 3/2 & 3/2 & 1 & 1 & 1/2 & 1/2 & 1/2 & 1/2 & \infty \end{array}\right)$$

Fig. 1.2. The matrix $\widetilde{A}$ of Example 1.1

Again, the lines subdividing the matrix indicate the coarsest equitable partition. The core factor of $[\widetilde{A}]$, which turns out to be the iterated core factor of $\widetilde{A}$, is

$$[\![\widetilde{A}]\!] = [[\widetilde{A}]] = \begin{pmatrix} 4 & 2 & 1 \\ 12 & 4 & \infty \end{pmatrix}$$

This matrix corresponds to the LP

$$\begin{array}{rl} \min & (c')^{\mathsf{t}} x' \\ \text{subject to} & A'x' = b',\ x' \geq 0, \end{array} \tag{L'}$$

where $A' = (4\ \ 2)$, $b' = (1)$, and $(c')^{\mathsf{t}} = (12,\ 4)$. An optimal solution to (L') is $x' = (0, \frac{1}{2})^{\mathsf{t}}$. We can lift x' to a solution of the original LP (L) by multiplying it with a suitable matrix, which yields the (optimal) solution

$$x := (0, 0, 0, 0, 0, 0, 0, 1/2, 1/2, 1/2, 1/2, 1/2, 1/2)^{\mathsf{t}}.$$

2 Preliminaries

We use a standard notation for graphs and digraphs. In graph G, we let $N^G(v)$ denote the set of neighbours of vertex v, and in a digraph D we let $N_+^D(v)$ and $N_-^D(v)$ denote, respectively, the sets of out-neighbours and in-neighbours of v.

We have already introduced some basic matrix notation in the introduction. A *permutation matrix* is a 0-1-matrix that has exactly one 1 in every row and column. We call two matrices $A^1 \in \mathbb{R}^{V^1 \times W^1}$ and $A^2 \in \mathbb{R}^{V^2 \times W^2}$ *isomorphic* (and write $A^1 \cong A^2$) if there are bijective mappings $\pi : V^1 \to V^2$ and $\rho : W^1 \to W^2$ such that $A^1_{vw} = A^2_{\pi(v)\rho(w)}$ for all $v \in V^1, w \in W^1$. Equivalently, A^1 and A^2 are isomorphic if there are permutation matrices $X \subseteq \mathbb{R}^{V^2 \times V^1}$ and $Y \subseteq \mathbb{R}^{W^2 \times W^1}$ such that $XA^1 = A^2Y$.

A matrix $X \in \mathbb{R}^{V \times W}$ is *stochastic* if it is nonnegative and $\sum_{w \in W} X_{vw} = 1$ for all $v \in V$. It is *doubly stochastic* if both X and its transpose X^{t} are stochastic. Observe that a doubly stochastic matrix is always square.

The *direct sum* of two matrices $A^1 \in \mathbb{R}^{V^1 \times W^1}$ and $A^2 \in \mathbb{R}^{V^2 \times W^2}$ is the matrix

$$A^1 \oplus A^2 := \begin{pmatrix} A^1 & 0 \\ 0 & A^2 \end{pmatrix}$$

With every matrix $A \subseteq \mathbb{R}^{V \times W}$ we associate its *bipartite graph* B_A with vertex set $V \cup W$ and edge set $\{vw \mid A_{vw} \neq 0\}$. The matrix A is *connected* if B_A is connected. (Sometimes, this is called *indecomposable.*) Note that A is not connected if and only if it is isomorphic to matrix that can be written as the direct sum of two matrices. A *connected component* of A is a submatrix A' whose rows and columns form the vertex set of a connected component of the bipartite graph B_A. With every square matrix $A \in \mathbb{R}^{V \times V}$ we associate two more graphs: the directed graph D_A has vertex set V and edge set $\{(v, v') \mid A_{vv'} \neq 0\}$. The graph G_A is the underlying undirected graph of D_A. We call A *strongly connected* if the graph D_A is strongly connected. (Sometimes, this is called *irreducible.*) It is not hard to see that a doubly stochastic matrix X is strongly connected if and only if the graph G_A is connected.

Let $A \in \mathbb{R}^{V \times W}$. For all subsets $P \subseteq V, Q \subseteq W$, we let

$$F^A(P, Q) = \sum_{(v,w) \in P \times Q} A_{vw}. \tag{2.1}$$

We write $F^A(v, Q)$, $F^A(P, w)$ instead of $F^A(\{v\}, Q)$, $F^A(P, \{w\})$. We omit the superscript A in F^A if A is clear from the context.

Sometimes, we consider matrices with entries from $\overline{\mathbb{R}} = \mathbb{R} \cup \{\infty\}$. We will only form linear combinations of elements of $\overline{\mathbb{R}}$ with nonnegative real coefficients, using the rules $r + \infty = \infty + r = \infty$ for all $r \in \overline{\mathbb{R}}$ and $0 \cdot \infty = 0$, $r \cdot \infty = \infty$ for $r > 0$.

3 Fractional Automorphisms and Isomorphisms

In this section, we prove the theorem relating fractional automorphisms to equitable partitions. For every pair $(X, Y) \in \mathbb{R}^{V \times V} \times \mathbb{R}^{W \times W}$ of matrices we let

$\mathcal{P}_X$ be the partition of V into the strongly connected components of X, and we let $\mathcal{Q}_Y$ be the partition of W into the strongly connected components of Y. Conversely, for every partition $(\mathcal{P}, \mathcal{Q})$ of A, we let $X_{\mathcal{P}} \in \mathbb{R}^{V \times V}$ be the matrix with entries $X_{vv'} := 1/|P|$ if $v, v' \in P$ for some $P \in \mathcal{P}$ and $X_{vv'} := 0$ otherwise, and we let $Y_{\mathcal{Q}} \in \mathbb{R}^{W \times W}$ be the matrix with entries $Y_{ww'} := 1/|Q|$ if $w, w' \in Q$ for some $Q \in \mathcal{Q}$ and $Y_{vv'} := 0$ otherwise.

Theorem 3.1. *Let $A \in \mathbb{R}^{V \times W}$.*

1. *If $(\mathcal{P}, \mathcal{Q})$ is an equitable partition of A, then $(X_{\mathcal{P}}, Y_{\mathcal{Q}})$ is a fractional automorphism.*
2. *If (X, Y) is a fractional automorphism of A, then $(\mathcal{P}_X, \mathcal{Q}_Y)$ is an equitable partition.*

Proof. To prove (1), let $(\mathcal{P}, \mathcal{Q})$ be an equitable partition of A, and let $X := X_{\mathcal{P}}$ and $Y := Y_{\mathcal{Q}}$. Let $v \in V, w \in W$, and let $P \in \mathcal{P}$ and $Q \in \mathcal{Q}$ be the classes of v and w, respectively. Then

$$(XA)_{vw} = \sum_{v' \in P} \frac{1}{|P|} \cdot A_{v'w} = \frac{1}{|P|} \cdot F(P, w) \overset{(a)}{=} \frac{1}{|Q|} \cdot F(v, Q) = (AY)_{vw}$$

Equality (a) can be established by a double-counting argument: we have $F(P, Q) = \sum_{v' \in P} F(v', Q) = |P| \cdot F(v, Q)$ by (1.1) and $F(P, Q) = \sum_{w' \in Q} F(P, w') = |Q| \cdot F(P, w)$ by (1.2).

To prove (2), let (X, Y) be a fractional automorphism of A. Let $P \in \mathcal{P}_X$ and $Q \in \mathcal{Q}_Y$. We need to prove that P, Q satisfy (1.1) and (1.2).

We first prove (1.1). For every $v \in P$, we have

$$\begin{aligned} F(v, Q) &= \sum_{w' \in Q} A_{vw'} \overset{(b1)}{=} \sum_{w' \in Q} A_{vw'} \sum_{w \in Q} Y_{w'w} = \sum_{w \in Q} \sum_{w' \in Q} A_{vw'} Y_{w'w} \\ &\overset{(b2)}{=} \sum_{w \in Q} \sum_{v' \in P} X_{vv'} A_{v'w} = \sum_{v' \in P} X_{vv'} \underbrace{\sum_{w \in Q} A_{v'w}}_{=F(v', Q)} \overset{(b3)}{=} \sum_{v' \in N_+^{D_X}(v)} X_{vv'} \cdot F(v', Q), \end{aligned}$$

Equation $(b1)$ holds because $\sum_{w \in W} Y_{w'w} = 1$ and $Y_{w'w} = 0$ for $w' \in Q, w \notin Q$. Here we use that Q, which by definition is a strongly connected component of the digraph D_Y, is also a connected component of the undirected graph G_Y. Equation $(b2)$ holds by $XA = AY$ and $Y_{w'w} = 0$ for $w' \notin Q, w \in Q$ and $X_{vv'} = 0$ for $v \in P, v' \notin P$. Equation $(b3)$ holds, because $N_+^{D_X}(v) \subseteq P$ and $X_{vv'} \neq 0 \iff v' \in N_+^{D_X}(v)$. As the matrix X is stochastic, this implies that $F(v, Q)$ is a strictly positive convex combination of the $F(v', Q)$ for $v' \in N_+^{D_X}(v)$. Note that P is the vertex set of a strongly connected component of D_X.

We use the fact that if we have a function f on the vertices of a strongly connected digraph such that for each vertex v the value $f(v)$ is a strictly positive convex combination of the values $f(w)$ for the out-neighbours w of v then this function is constant. Thus $F(v, Q) = F(v', Q)$ for all $v, v' \in P$. This proves (1.1).

(1.2) can be proved similarly. $\square$

In the full version [7], we also introduce a notion of *fractional isomorphism* between two matrices and characterise it in terms of equitable partitions.

4 Factor Matrices and Partition Equivalence

For our applications, fractional isomorphism is an equivalence relation that is still too fine. In this section, we introduce a coarser equivalence relation that we will call *partition equivalence.* For the applications in the next section, it will be helpful to develop partition equivalence for matrices with entries from $\overline{\mathbb{R}} = \mathbb{R} \cup \{\infty\}$.

4.1 Partition Matrices and Factor Matrices

A *partition matrix* is a 0-1 matrix that has exactly one 1-entry in each row and at least one 1-entry in each column. We usually denote partition matrices by $\varPi$ or C, D. With each partition matrix $\varPi \subseteq \{0,1\}^{V \times T}$ we associate a partition $\{P_t \mid t \in T\}$ of V into parts $P_t = \{v \in V \mid \varPi_{vt} = 1\}$. Conversely, with every partition $\mathcal{P}$ of V we associate the partition matrix $\varPi_{\mathcal{P}} \in \{0,1\}^{V \times \mathcal{P}}$ defined by $(\varPi_{\mathcal{P}})_{vP} = 1 \iff v \in P$, for all $v \in V$ and $P \in \mathcal{P}$.

Note that partition matrices are stochastic, but, in general, not doubly stochastic. (The only doubly stochastic partition matrices are the permutation matrices.) For every partition matrix $\varPi \in \mathbb{R}^{V \times T}$, we define its *scaled transpose* to be the matrix $\varPi^{\mathsf{s}} \in \mathbb{R}^{T \times V}$ with entries $\varPi^{\mathsf{s}}_{tv} := \frac{\varPi_{vt}}{\sum_{v' \in V} \varPi_{v't}}$. Then $\varPi^{\mathsf{s}}$ is the transpose of $\varPi$ scaled to a stochastic matrix. Observe that the matrix $\varPi\varPi^{\mathsf{s}} \in \mathbb{R}^{V \times V}$ is symmetric and doubly stochastic. Indeed, if $\varPi = \varPi_{\mathcal{P}}$ for a partition $\mathcal{P}$ of V, then $(\varPi\varPi^{\mathsf{s}})_{vv'} = X_{\mathcal{P}}$ for the matrix $X_{\mathcal{P}}$ defined on page 511. Thus by Theorem 3.1, if $(\mathcal{P}, \mathcal{Q})$ is an equitable partition of a matrix $A \in \mathbb{R}^{V \times W}$, and we let $C := \varPi_{\mathcal{P}}$ and $D := \varPi_{\mathcal{Q}}$, then $(CC^{\mathsf{s}}, DD^{\mathsf{s}})$ is a fractional automorphism of A.

A *factor matrix* of a matrix $A \in \overline{\mathbb{R}}^{V \times W}$ is a matrix $B = \varPi^{\mathsf{s}}_{\mathcal{P}} A \varPi_{\mathcal{Q}} \in \overline{\mathbb{R}}^{\mathcal{P} \times \mathcal{Q}}$, where $(\mathcal{P}, \mathcal{Q})$ is an equitable partition of A. The asymmetry in the definition (multiplying with $\varPi^{\mathsf{s}}_{\mathcal{P}}$ rather than $\varPi^{\mathsf{t}}_{\mathcal{P}}$) may seem strange first, but turns out to be necessary in several places. An immediate advantage of it is that we multiply with stochastic matrices from both sides. Note that for all $P \in \mathcal{P}, Q \in \mathcal{Q}$,

$$B_{PQ} = \frac{1}{|P|} F^A(P, Q) = F^A(v, Q) \tag{4.1}$$

for some (and hence for all) $v \in P$. We will see that factor matrices still carry all information about a matrix necessary to solve systems of linear equations and linear programs.

As the dimensions of B are determined by the number of classes of the partition, there is a unique smallest factor matrix $[A] := \varPi^{\mathsf{s}}_{\mathcal{P}_\infty} A \varPi_{\mathcal{Q}_\infty}$, where, as usual, $(\mathcal{P}_\infty, \mathcal{Q}_\infty)$ denotes the coarsest equitable partition of A. We call $[A]$ the *core factor* of A. Since we can compute the coarsest equitable partition in time $O((n+m)\log n)$, we can also compute the core factor in time $O((n+m)\log n)$.

4.2 Partition Equivalence

We define the relation $\approx$ on the class of all matrices by letting $A^1 \approx A^2$ if there are factor matrices B^1 of A^1 and B^2 of A^2 such that B^1 and B^2 are isomorphic. Observe that for every matrix $A \in \overline{\mathbb{R}}^{V\times W}$ we have $A \approx [A]$. Moreover, fractionally isomorphic matrices are partition equivalent, but in general not vice versa.

Maybe surprisingly, the relation $\approx$ is *not* an equivalence relation. It is obviously reflexive and symmetric, but it is not transitive. In Example 1.1 we have a matrix $\widetilde{A}$ with distinct factor matrices $[\widetilde{A}]$ and $[[\widetilde{A}]]$. Then $\widetilde{A} \approx [\widetilde{A}]$ and $[\widetilde{A}] \approx [[\widetilde{A}]]$, but $A \not\approx [[\widetilde{A}]]$, because there is no factor of $\widetilde{A}$ smaller than the core factor.

We let $\approx^*$ be the transitive closure of $\approx$ and call two matrices A^1, A^2 *partition equivalent* if $A^1 \approx^* A^2$.

Let $A \in \overline{\mathbb{R}}^{V\times W}$. We let $[A]_0 := A$ and $[A]_{i+1} := [[A]_i]$ for every $i \geq 0$. Then there is an $i \leq |V| + |W|$ such that $[A]_i = [A]_{i+1}$. We denote $[A]_i$ by $[\![A]\!]$ and call it the *iterated core factor* of A. Observe that $A \approx^* [\![A]\!]$. In the full version [7], we give an example of partition equivalent matrices with nonisomorphic iterated core factors and also an example of a matrix whose iterated core factor is not the smallest partition equivalent matrix. This is unfortunate, because it leaves us without an efficient way of deciding partition equivalence.

It remains an open question whether partition equivalence is decidable, or even decidable in polynomial time. Note, however, that we can compute $[\![A]\!]$ from A in time $O(n(n+m)\log n)$. It is conceivable that this can be improved to $O((n+m)\log n)$, but this remains open as well.

5 Reducing the Dimension of a Linear Program

In this section, we will apply our theory of fractional automorphisms and partition equivalence to solving systems of linear equations and linear programs. For the ease of presentation, we only consider linear programs in standard form:

$$\begin{aligned} \min\ & c^{\mathrm{t}}x \\ \text{subject to}\ & Ax = b,\ x \geq 0, \end{aligned} \tag{$L_{A,b,c}$}$$

where $A \in \mathbb{R}^{V\times W}$, $b \in \mathbb{R}^V$, $c \in \mathbb{R}^W$, and $x = (x_w \mid w \in W)$. We combine the matrix A and the vectors b, c in one matrix $\widetilde{A} = \widetilde{A}(A, b, c) \in \overline{\mathbb{R}}^{(V\cup\{v_\infty\})\times(W\cup\{w_\infty\})}$, where $v_\infty \neq w_\infty \notin V \cup W$, defined by $\widetilde{A}_{vw} := A_{vw}$ and $A_{vw_\infty} := b_v$ and $\widetilde{A}_{v_\infty w} := c_w$ and $A_{v_\infty w_\infty} := \infty$, for all $v \in V, w \in W$ (see Example 1.1).

As A is a real matrix (that does not contain ∞ as an entry), every equitable partition $(\widetilde{\mathcal{P}}, \widetilde{\mathcal{Q}})$ of $\widetilde{A}$ contains $\{v_\infty\}$ and $\{w_\infty\}$ as separate classes. If we let

$\mathcal{P} := \widetilde{\mathcal{P}} \setminus \{v_\infty\}$ and $\mathcal{Q} := \widetilde{\mathcal{Q}} \setminus \{w_\infty\}$, then $(\mathcal{P}, \mathcal{Q})$ is an equitable partition of A satisfying

$$b_v = b_{v'} \qquad \text{for all } P \in \mathcal{P}, v, v' \in P, \tag{5.1}$$

$$c_w = c_{w'} \qquad \text{for all } Q \in \mathcal{Q}, w, w' \in Q. \tag{5.2}$$

Furthermore, if $(\widetilde{\mathcal{P}}, \widetilde{\mathcal{Q}})$ is the coarsest equitable partition of $\widetilde{A}$ then $(\mathcal{P}, \mathcal{Q})$ is the coarsest equitable partition of A that satisfies (5.1) and (5.2).

Lemma 5.1 (Reduction Lemma). *Let $A, b, c, \widetilde{A}$ as above. Let $(\widetilde{\mathcal{P}}, \widetilde{\mathcal{Q}})$ an equitable partition of $\widetilde{A}$ and $\mathcal{P} := \widetilde{\mathcal{P}} \setminus \{v_\infty\}$, $\mathcal{Q} := \widetilde{\mathcal{Q}} \setminus \{w_\infty\}$. Furthermore, let $C := \Pi_{\mathcal{P}}$ and $D := \Pi_{\mathcal{Q}}$ and $A' := C^{\mathsf{s}}AD$, $b' := C^{\mathsf{s}}b$, and $c' := D^{\mathsf{t}}c$.*

1. *If $x \in \mathbb{R}^W$ is a feasible solution to $(L_{A,b,c})$ then $x' := D^{\mathsf{s}}x$ is a feasible solution to $(L_{A',b',c'})$. Moreover, if x is an optimal solution then x' is an optimal solution as well.*
2. *If $x' \in \mathbb{R}^{\mathcal{Q}}$ is a feasible solution to $(L_{A',b',c'})$, then $x := Dx'$ is a feasible solution to $(L_{A,b,c})$, and if x' is optimal then x is optimal as well.*

Proof. An easy calculation shows that $C^{\mathsf{s}}CC^{\mathsf{s}} = C^{\mathsf{s}}$ and $DD^{\mathsf{s}}D = D$.

To prove (1), let $x \in \mathbb{R}^W$ be a feasible solution to $(L_{A,b,c})$ and $x' := D^{\mathsf{s}}x \in \mathbb{R}^{\mathcal{Q}}$. Then $x' \geq 0$ because $x \geq 0$ and D^{s} is nonnegative. Furthermore,

$$A'x' = C^{\mathsf{s}}ADD^{\mathsf{s}}x \overset{(a)}{=} C^{\mathsf{s}}CC^{\mathsf{s}}Ax \overset{(b)}{=} C^{\mathsf{s}}b = b'.$$

Here (a) holds because $(CC^{\mathsf{s}}, DD^{\mathsf{s}})$ is a fractional automorphism of A and (b) holds because $C^{\mathsf{s}}CC^{\mathsf{s}} = C^{\mathsf{s}}$ and $Ax = b$. Thus x' is a feasible solution to $(L_{A',b',c'})$.

Before we prove the second assertion of (1) regarding optimal solutions, we prove the first assertion of (2). Let $x' \in \mathbb{R}^{\mathcal{Q}}$ be a feasible solution to $(L_{A',b',c'})$ and $x := Dx' \in \mathbb{R}^W$. Then $x \geq 0$ because $x' \geq 0$ and D is nonnegative. Furthermore,

$$Ax = ADx' \overset{(c)}{=} ADD^{\mathsf{s}}Dx' \overset{(d)}{=} CC^{\mathsf{s}}ADx' = CA'x' = Cb' = CC^{\mathsf{s}}b \overset{(e)}{=} b.$$

Here (c) holds, because $DD^{\mathsf{s}}D = D$, and (d) holds, because $(CC^{\mathsf{s}}, DD^{\mathsf{s}})$ is a fractional automorphism of A. (e) follows from (5.1).

It remains to prove the two assertions about optimal solutions. Suppose first that $x \in \mathbb{R}^W$ is an optimal solution to $(L_{A,b,c})$, and let $x' := D^{\mathsf{s}}x$. Then x' is a feasible solution to $(L_{A',b',c'})$. We claim that it is optimal. Let y' be another feasible solution to $(L_{A',b',c'})$. We shall prove that $(c')^{\mathsf{t}}x' \leq (c')^{\mathsf{t}}y'$. Let $y = Dy'$. Then y is a feasible solution to $(L_{A,b,c})$, and thus $c^{\mathsf{t}}x \leq c^{\mathsf{t}}y$ by the optimality of x. Thus

$$(c')^{\mathsf{t}}x' = c^{\mathsf{t}}DD^{\mathsf{s}}x \overset{(f)}{=} c^{\mathsf{t}}x \leq c^{\mathsf{t}}y = c^{\mathsf{t}}Dy' = (c')^{\mathsf{t}}y'.$$

To that (f) holds, observe that $c^{\mathsf{t}}DD^{\mathsf{s}} = c^{\mathsf{t}}$. This follows from (5.2).

Suppose conversely that $x' \in \mathbb{R}^{\mathcal{Q}}$ is an optimal solution to $(L_{A',b',c'})$ and let $x := Dx'$. Then x is a feasible solution to $(L_{A,b,c})$. Let y be another feasible solution. Then $y' := D^{\mathsf{s}}y$ is a feasible solution to $(L_{A',b',c'})$, and by the optimality of x' we have $(c')^{\mathsf{t}}x' \leq (c')^{\mathsf{t}}y'$. Thus

$$c^{\mathsf{t}}x \overset{(f)}{=} c^{\mathsf{t}}DD^{\mathsf{s}}x = (c')^{\mathsf{t}}x' \leq (c')^{\mathsf{t}}y' = c^{\mathsf{t}}DD^{\mathsf{s}}y \overset{(f)}{=} c^{\mathsf{t}}y.$$

The two equations marked (f) hold, because $c^{\mathsf{t}}DD^{\mathsf{s}} = c^{\mathsf{t}}$, as we have seen above. □

Theorem 5.2. *For $j = 1, 2$, let $A^j \in \mathbb{R}^{V^j \times W^j}$ and $b^j \in \mathbb{R}^{V^j}$ and $c^j \in \mathbb{R}^{W^j}$ and $\widetilde{A}^j := \widetilde{A}(A^j, b^j, c^j)$. Suppose that $\widetilde{A}^1 \approx^* \widetilde{A}^2$. Then for $j = 1, 2$ there is a matrix $M^j \in \mathbb{R}^{W^{3-j} \times W^j}$ such that for all $x \in \mathbb{R}^{W^j}$, if x is a feasible solution to (L_{A^j,b^j,c^j}) then $M^j x$ is a feasible solution to $(L_{A^{3-j},b^{3-j},c^{3-j}})$.*

Furthermore, if x is an optimal solution to (L_{A^j,b^j,c^j}) then $M^j x$ is an optimal solution to $(L_{A^{3-j},b^{3-j},c^{3-j}})$.

A proof of the theorem can be found in the full version [7]. Example 1.1 illustrates how the theorem can be applied.

6 Implementation and Computational Evaluation

In our experimental evaluation, we only apply the Reduction Lemma 5.1 once to the coarsest equitable partition. We believe that the gain we may have by searching for a smaller partition equivalent matrix than the core factor, for example the iterated core factor, is almost always outweighed by the additional time spent to find such a matrix. But we have not yet conducted any systematic experiments in this direction yet.

Let us briefly describe our implementation. We are given $A \in \mathbb{R}^{V \times W}$, $b \in \mathbb{R}^V$, $c \in \mathbb{R}^W$ and want to solve the linear program $(L_{A,b,c})$. To apply the Reduction Lemma, instead of computing the coarsest equitable partition of the matrix $\widetilde{A}(A, b, c)$, we directly compute the coarsest equitable partition $(\mathcal{P}, \mathcal{Q})$ of A that refines an initial partition $(\mathcal{P}_0, \mathcal{Q}_0)$ depending on the vectors b and c. ($\mathcal{P}_0$ is the partition of V where v and v' are in the same class if $b_v = b_{v'}$, and $\mathcal{Q}_0$ is defined similarly from c.) We compute $(\mathcal{P}, \mathcal{Q})$ using colour refinement starting from the initial partition $(\mathcal{P}_0, \mathcal{Q}_0)$.

To investigate the computational benefits of our reduction method, we carried out three series of experiments. For the first we used Margot's benchmark of (integer) linear programs with symmetries.[2] They encode various combinatorial optimisation problems. For the second, we considered the computation of the value function of Markov Decision Problems, modelling decision making in situations where outcomes of actions are partly random. This can naturally be modelled as LPs. For the third, we considered MAP inference in Markov logic networks. In all cases, the reduction in the number of variables and constraints of the LPs was significant. Moreover, and more importantly, the time spent in total on solving the LPs — reducing an LP and solving the reduced LP — is

[2] `http://wpweb2.tepper.cmu.edu/fmargot/lpsym.html`

often an order of magnitude smaller than solving the original LP directly. We have compared our method with a method of symmetry reduction for LPs due to Bödi, Grundhöfer and Herr [3]. A more detailed description of our experiments can be found in the full version [7].

References

1. Ahmadi, B., Kersting, K., Mladenov, M., Natarajan, S.: Exploiting symmetries for scaling loopy belief propagation and relational training. Machine Learning Journal 92, 91–132 (2013)
2. Berkholz, C., Bonsma, P., Grohe, M.: Tight lower and upper bounds for the complexity of canonical colour refinement. In: Proceedings of the 21st Annual European Symposium on Algorithms (2013) (to appear)
3. Bödi, R., Grundhöfer, T., Herr, K.: Symmetries of linear programs. Note di Matematica 30(1), 129–132 (2010)
4. Bui, H.H., Huynh, T.N., Riedel, S.: Automorphism groups of graphical models and lifted variational inference. In: Proc. of the 29th Conference on Uncertainty in Artificial Intelligence, UAI-2013 (2013)
5. Cardon, A., Crochemore, M.: Partitioning a graph in $O(|A| \log_2 |V|)$. Theoretical Computer Science 19(1), 85–98 (1982)
6. Godsil, C.D.: Compact graphs and equitable partitions. Linear Algebra and its Applications 255, 259–266 (1997)
7. Grohe, M., Kersting, K., Mladenov, M., Selman, E.: Dimension reduction via colour refinement. ArXiv, 1307.5697 (2014) (full version of this paper)
8. Hopcroft, J.E.: An n log n algorithm for minimizing states in a finite automaton. In: Kohavi, Z., Paz, A. (eds.) Theory of Machines and Computations, pp. 189–196. Academic Press (1971)
9. Kersting, K., Ahmadi, B., Natarajan, S.: Counting Belief Propagation. In: Proc. of the 25th Conf. on Uncertainty in Artificial Intelligence, UAI 2009 (2009)
10. Mladenov, M., Ahmadi, B., Kersting, K.: Lifted linear programming. In: 15th Int. Conf. on Artificial Intelligence and Statistics (AISTATS 2012). JMLR: W&CP 22, vol. 22, pp. 788–797 (2012)
11. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. SIAM Journal on Computing 16(6), 973–989 (1987)
12. Ramana, M.V., Scheinerman, E.R., Ullman, D.: Fractional isomorphism of graphs. Discrete Mathematics 132, 247–265 (1994)
13. Singla, P., Domingos, P.: Lifted First-Order Belief Propagation. In: Proc. of the 23rd AAAI Conf. on Artificial Intelligence (AAAI 2008), Chicago, IL, USA, July 13-17, pp. 1094–1099. AAAI Press, Menlo Park (2008)
14. Tinhofer, G.: A note on compact graphs. Discrete Applied Mathematics 30, 253–264 (1991)

How Experts Can Solve LPs Online[⋆]

Anupam Gupta[1] and Marco Molinaro[2]

[1] Computer Science Department, Carnegie Mellon University, Pittsburgh, PA
[2] ISyE, Georgia Tech, Atlanta, GA

Abstract. We consider the problem of solving packing/covering LPs online, when the columns of the constraint matrix are presented in random order. This problem has received much attention: the main open question is to figure out how large the right-hand sides of the LPs have to be (compared to the entries on the left-hand side of the constraint) to get $(1+\varepsilon)$-approximations online? It is known that the RHS has to be $\Omega(\varepsilon^{-2}\log m)$ times the left-hand sides, where m is the number of constraints.

In this paper we show how to achieve this bound for all packing LPs, and also for a wide class of mixed packing/covering LPs. Our algorithms construct dual solutions using a regret-minimizing online learning algorithm in a black-box fashion, and use them to construct primal solutions. The adversarial guarantee that holds for the constructed duals help us to take care of most of the correlations that arise in the algorithm; the remaining correlations are handled via martingale concentration and maximal inequalities. These ideas lead to conceptually simple and modular algorithms, which we hope will be useful in other contexts.

1 Introduction

In this paper we consider the problem of solving packing-covering LPs online. For concreteness, consider as a simple example the packing LP $\max\{\pi^\intercal x \mid Ax \leq b, x \in [0,1]^n\}$—we will consider more complicated LPs in this paper—where the columns of A are being revealed one by one, and we have to choose values for each x_t and irrevocably before the next columns are revealed.

While such problems have been also studied in the worst-case competitive analysis [5,3], to avoid the pessimistic achievable guarantees a lot of recent research has focused on the *random permutation* model: the matrix A is chosen adversarially but its columns are presented to the algorithm in random order. Several packing problems have been considered in this model, starting from a classic maximization version of the secretary problem [10], to single-knapsack problems [16,4], the AdWords problem [17,13,8] and recently more general packing LPs [11,1,18,15]. These models have a vast range of applications, like online advertisement, online routing, and airline revenue management. In this paper we consider this random permutation model.

⋆ Research partly supported by NSF awards CCF-1016799 and CCF-1319811, and by a grant from the CMU-Microsoft Center for Computational Thinking.

A major question of research has been: *how large must the right-hand sides of the LPs be (compared to the left-hand side coefficients) to get* $(1+\varepsilon)$*-approximations online, say, in expectation?*

Dual methods. Dual-based method are very important in optimization under uncertainty and have been used extensively in the operations management literature [20,22,21]. Most of the works on packing LPs in the random permutation model [8,11,1,18] use the following form of this approach. If the optimal dual solution was known, then one could use it to set the primal variables according to their reduced costs and obtain an (almost) optimal solution. The idea is then to see the first few (say εn) columns of the LP, use this sample to estimate a good dual solution for the LP, and then use this dual solution to decide how to select the primal values for the future columns.

This idea was first analyzed by Devanur and Hayes for the AdWords problem [8], and recently extended to more general packing LPs [11,1]; the latter show that having right-hand sides roughly $\Omega(\varepsilon^{-2} m \log(n/\varepsilon))$ times larger than the left-hand side entries suffices, where m is the number of packing constraints. Recently, Molinaro and Ravi [18] also used a modified version of this idea to show that $\Omega(\varepsilon^{-2} m^2 \log(m/\varepsilon))$ suffices, removing the dependence on n at the expense of extra m factors. Agrawal et al. [1] showed that $\Omega(\varepsilon^{-2} \log m)$ is the best one can do, and this was believed to be the right answer.

The main difficulty in analyzing these algorithms lies in the fact that the primal and dual solutions are, by construction, heavily connected. In order to deal with the correlations that arise, all of these above analyses resort to a massive union bound, which is the root of the extra $\log n$ and m factors.

1.1 Our Techniques

Decoupling primal and dual via regret minimization. One of our contributions in this paper is an alternative algorithmic construction where the primal and the dual are loosely coupled. For that, we construct duals using a *regret-minimizing online learning* algorithm (which offers *adversarial* guarantees); in the primal, we use a greedy strategy which ensures that, with respect to these duals, we are doing at least as well as the optimal offline solution.

The fact that the guarantees for the dual hold in the adversarial setting, together with the greedy primal step, reduces the analysis to that of comparing the dual constructed solution to the *offline optimal* solution (see Section 2.4). Since the latter is a *fixed* solution (modulo the permutation of the columns), it is only loosely correlated with our constructed dual; in fact, correlations only come from the fact we can think of the random permutation model as sampling *without* replacement.

As far as we know, this is the first explicit black-box connection between regret-minimization and optimization in the random permutation model; in the offline packing-covering LP setting such connection was implicit in [19,23,12] and made explicit in [2] (see also [14,6]). In the simpler i.i.d. case, where columns of the LP are sampled *with* replacement, [7] uses multiplicative-weights based dual updates, reminiscent of the multiplicative-weight expert algorithm for regret-minimization. Our analysis then abstracts out the reliance on the details of that technique, giving perhaps a cleaner conceptual explanation of what is happening. Importantly, this isolation between the analysis of the regret-minimization algorithm and the comparison of our dual solution and the offline optimum is instrumental in dealing with the dependencies arising in the random permutation model.

Handling dependencies in random permutations. Handling the dependencies that arise from the fact that we are sampling without replacement is not immediate: the known bounds on the distances between sampling with and without replacement distributions (e.g. [9]) do not seem to be strong enough, and a simple approach requires taking a union bound across all time steps, leading again to extra $\log n$ factors. Instead, we use a maximal inequality for sampling without replacement to compare sampling with/without replacement at *every* time step while avoiding a union bound, together with martingale concentration inequalities. These ideas might find uses in other contexts.

1.2 Our Results

Let us now discuss our results in more detail. For brevity, several proofs are omitted from the paper and are deferred to its full version.

Generalized Load Balancing. First we consider a simple packing-type problem, modeling the following generalized load balancing: We have a set of m machines. For each arriving job t, there are k different options on how to serve it, and the j^{th} choice requires some amount of processing on each of the machines, given by the vector $(a^t_{1j}, a^t_{2j}, \ldots, a^t_{mj})$. When the job arrives, we must (fractionally) choose one of these options. The goal is to minimize the makespan, i.e. the maximum total processing assigned to any machine. Notice that when the matrices A^t's are diagonal $m \times m$ this captures the classic problem of scheduling in unrelated machines.

If the jobs are i.i.d. draws from some distribution, the result of Devanur et al. [7, Section 3] mentioned above achieve with probability $1 - \delta$ a $(1 + \varepsilon)$-approximation to the optimum makespan assuming this optimum is $\Omega(\frac{\log(m/\delta)}{\varepsilon^2})$ times the maximum processing a^t_{ij}. They left as an open question whether the same result holds in the more general random permutation model. Our first result, algorithm **expertLB**, answers this in the affirmative:

Informal Theorem 1 (Load Balancing). *Consider the load-balancing problem where the jobs arrive in random order. Given $\varepsilon > 0$, if the optimal makespan λ^* is at least $\Omega(\frac{\log(m/\delta)}{\varepsilon^2})$ times the maximum amount of processing required by any option for any job, the makespan for the online algorithm **expertLB** is at most $(1 + \varepsilon)\lambda^*$ with probability $1 - \delta$.*

As is common in this context, our algorithm in fact produces *integer* solutions that compare favorably to the optimal solution.

Moreover, we can handle processing requirements that can be both positive or negative, as long as for each machine, the jobs are mostly positive or mostly negative (see Definition 1). This is useful, because we employ this abstraction for the next result, to solve packing-covering linear programs.

Packing-Covering LPs. Again we have n jobs with k processing options. Choice j for a job t has a profit π_j^t. The goal is to make fractional choices $\{x_j^t\}$ (subject to $\sum_j x_j^t \leq 1$) that maximize the profit $\sum_{t,j} \pi_j^t x_j^t$ subject to m constraints, some of them of packing form $\sum_{t,j} a_{ij}^t x_j^t \leq b_i$ some of covering form $\sum_{t,j} c_{ij}^t x_j^t \geq d_i$ (a_{ij}^t and c_{ij}^t non-negative). This is a (multiple-choice) packing-covering LP where the items arrive online, in random order; the *multiple-choice* emphasizes the presence of constraints like $\sum_j x_j^t \leq 1$ which have small RHS. We show how to simply reduce the problem of solving such a packing-covering LP problem to a load balancing problem by viewing the covering constraints and the objective function as yet another packing constraints with (mostly) negative loads. This gives the next result.

Informal Theorem 2 (Packing-Covering LPs). *Consider a feasible multiple-choice packing-covering LP where the items arrive in random order. Given ε, δ, assume that each right hand side in the LP is at least $\Omega(\frac{\log(m/\delta)}{\varepsilon^2})$ times the maximum coefficient of the left-hand side in its row, and that the optimal profit is at least $\Omega(\frac{\log(m/\delta)}{\varepsilon^2})$ times each individual item's profit. Then given an (under)estimate O^* for the optimal profit, the algorithm **LPviaLB** computes an ε-feasible solution online that achieves a profit of $(1-\varepsilon)O^*$ with probability $1-\delta$.*

The astute reader will notice that the above two theorems required us to provide estimates of the optimal profit as input. This is sometimes a reasonable assumption (e.g. when enough historic knowledge of the problem directly provides such estimates). Nonetheless, we show how to construct such estimates as the columns come. However, to obtain good estimates we need to consider *stable* LP's, which informally means allowing solutions to violate the covering constraints by a small factor does not increase the optimal value by much (see Definition 2). We note that a packing-only LP is vacuously stable.

The rough idea to obtain the optimum estimate is to sample half the columns of the LP and solve it (with the capacities reduced by a factor of 2) to obtain a good estimate for the optimal value. This would, however, lose the profit from 50% of the LP right off the bat! We then use a doubling idea of [1] to learn progressively better opt estimates (this was also used in [16,7,18]).

Informal Theorem 3 (Packing-Covering LPs II). *Consider a feasible, stable, multiple-choice packing-covering LP in the random order setting. Given ε, δ, assume that each right hand side in the LP is at least $\Omega(\frac{\log(m/\delta)}{\varepsilon^2})$ times the maximum coefficient of the left-hand side in its row, and that the optimal profit is at least $\Omega(\frac{\log(m/\delta)}{\varepsilon^2})$ times each individual item's profit. Then the **MultiphaseLP** algorithm computes a solution that is ε-feasible with probability $1-\delta$ and has expected value at least $(1-\varepsilon)$ times the optimum.*

As far as we know this is the first guarantee for packing-covering LPs in the random permutation (or i.i.d.) model.

Packing LPs. This entire section appears only in the full version of the paper. The assumption of individual item profits being small compared to the optimal net profit is often a reasonable one. However, there are reasonable situations where we want a stronger result. Our final result removes all assumptions about the magnitude of the profit whenever we are dealing with packing-only LPs (without multiple-choice constraints of the form $\sum_j x_j^t \leq 1$). Obtaining good estimates of opt without any assumptions on the item values in our setting requires significantly new ideas.

Informal Theorem 4 (Ultimate Packing LP). *Consider a packing LP in the random order setting. Given $\varepsilon > 0$, suppose the capacities are $\Omega(\frac{\log(m/\varepsilon)}{\varepsilon^2})$ times any left hand side entry. Then the* ***MultiSkimLP*** *algorithm computes a solution online with expected value at least $(1-\varepsilon)$ times the optimum.*

As mentioned earlier, this problem has been approached using dual-based methods by several authors [8,1,18], showing that bounds $\Omega(\varepsilon^{-2} m \log n)$ and $\Omega(\varepsilon^{-2} m^2 \log m)$ are sufficient and $\Omega(\varepsilon^{-2} \log m)$ is necessary. Hence our result is asymptotically optimal.

Very recently, and independently of our work, Kesselheim et al. [15] give a primal-only algorithm for the more general multiple-choice packing LPs that achieves a $(1+\varepsilon)$-approximation when the bound on the capacities is $\Omega(\varepsilon^{-2} \log d)$, where d is the *column-sparsity* of the constraint matrix (i.e. maximum number of non-zero entries in each column). Since $d \leq m$, this bound is optimal. Their algorithm is very elegant, where at time t the optimal solution of a primal LP comprising the columns seen up to time t is directly used to set the variable x_t; somewhat surprisingly, this strategy, which does not directly impose consistency of the x_t's set in different time steps, still works.

While the guarantee for packing LPs we obtain does not fully recover the result of Kesselheim et al., we believe that it is still worth presenting, specially given the potential of this method to extend to more general cases, e.g. to packing-covering LPs, and also as a different primal-dual approach to contrast with their primal-only ideas.

2 Load-Balancing Using Experts

In this section, we formally define the generalized load-balancing problem and present our online algorithm for it.

2.1 Definitions: Offline and Online Instances

An instance of the *offline* version of the Load-balance problem is a set of matrices $\{A^t\}$, each in $\mathbb{R}^{m \times k}$. The goal is to find vectors $p^1, p^2, \ldots, p^n \in \Delta^k$ to minimize $\|\sum_{t=1}^n A^t p^t\|_{\max}$, the load of the most-loaded machine, where $\|v\|_{\max} = \max_j v_j$

and Δ^k is the simplex $\{p \in [0,1]^k : \sum_i p_i = 1\}$. We use throughout λ^* to denote the offline optimum value.

In the *online* version of this problem, we consider the random permutation model. Now the number of time steps n is the only information known upfront. Let $\mathbf{A}^1, \mathbf{A}^2, \ldots, \mathbf{A}^n$ be matrices sampled from the set $\{A^1, \ldots, A^n\}$ uniformly *without replacement*. At time step t, the algorithm has seen matrices $\mathbf{A}^1, \ldots, \mathbf{A}^t$ and must decide on a (random) vector $\mathbf{p}^t \in \Delta$; the randomness is both from the random order and the internal coin flips of the algorithm (if any). The goal is to minimize $\|\sum_{t=1}^n \mathbf{A}^t \mathbf{p}^t\|_{\max}$. The vectors $\{\mathbf{p}^t\}_t$ output by the algorithm are called the *online solution* for the online instance $\{\mathbf{A}^t\}_t$ corresponding to the offline instance $\mathcal{I}$.

2.2 Well-Bounded Instances

While instances arising from scheduling-type applications usually consist of positive loads, for future sections it is useful to handle instances where some entries A^t_{ij} are also negative. To get good guarantees we need to control the class of permissible instances: loosely speaking, while we allow the entries of the load matrices A^t to be in the symmetric interval $[-M, M]$, on each machine the loads are either mostly positive on all steps (with any negative loads being very small), or mostly a negative load on all steps.

Definition 1. *For $M, \gamma \geq 0$, an instance $A^1, \ldots, A^n$ of the load-balance problem is* (M,γ)-well-bounded *if $A^t_{ij} \in [-M, M]$ for all i, j, t, and moreover for each $i \in [m]$ we have: either $A^t_{ij} \in [-\min\{\frac{\gamma\lambda^*}{n}, M\}, M]$ for all $t \in [n]$, or $A^t_{ij} \in [-M, \min\{\frac{\gamma\lambda^*}{n}, M\}]$ for all $t \in [n]$.*

In particular, this is satisfied with $\gamma = 0$ if the A^t's are non-negative. The reader can think of γ as a small constant, say one. The main motivation behind this definition is that it allows us to control the variation of random processes defined over $\{A^t\}_t$, as the next lemma shows.

Lemma 1. *Suppose $\{A^t\}_{t=1}^n$ is an (M,γ)-well-bounded instance for some $M, \gamma \geq 0$, and consider $\widehat{p}^1, \ldots, \widehat{p}^n \in \Delta^k$. Let the sequence $\mathbf{A}^1\widehat{\mathbf{p}}^1, \ldots \mathbf{A}^n\widehat{\mathbf{p}}^n$ be sampled without replacement from the set $\{A^t\widehat{p}^t\}_t$. Then for every event $\mathcal{E}$ and for every $i \in [m]$ and $t \in [n]$,*

$$\mathbb{E}\left[|\mathbf{A}^t_i\widehat{\mathbf{p}}^t - \mathbb{E}[\mathbf{A}^t_i\widehat{\mathbf{p}}^t \mid \mathcal{E}]| \mid \mathcal{E}\right] \leq \frac{2\gamma\lambda^*}{n} + 2|\mathbb{E}[\mathbf{A}^t_i\widehat{\mathbf{p}}^t \mid \mathcal{E}]|.$$

2.3 The ExpertLB Algorithm and Its Guarantee

To define the algorithm **expertLB**, we need to recall the *online experts problem* [2]: We are given an adversarial sequence of vectors $o^1, o^2, \ldots, o^n \in [-1,1]^m$. At time t, using only information up to time $t-1$ (i.e. $o^1, \ldots, o^{t-1}$), we need to compute a vector $w^t \in \Delta^m$; then we incur a reward of $\langle w^t, o^t \rangle$. The goal is to maximize the total reward $\sum_t \langle w^t, o^t \rangle$.

Given an online instance $\{\mathbf{A}^t\}$ to the generalized load-balancing problem and values n, M, and ε, the following algorithm **expertLB** (for "expert load-balancing") runs a primal greedy strategy and a dual online experts algorithm, restarting at timestep $n/2$. The algorithm maintains primal vectors $\mathbf{p}^1, \ldots, \mathbf{p}^{n/2} \in \Delta^k$ and "dual" vectors $\mathbf{w}^1, \ldots, \mathbf{w}^{n/2} \in \Delta^m$ as follows:

(P) primal step: in step t, the algorithm sees the random item $\mathbf{A}^t$, computes $\mathbf{w}^t\mathbf{A}^t$ and chooses $\mathbf{p}^t \in \Delta^k$ so as to minimize $\langle \mathbf{w}^t\mathbf{A}^t, \mathbf{p}^t \rangle$;

(D) dual step: run an online experts algorithm (with $\mathbf{A}^1\mathbf{p}^1, \ldots, \mathbf{A}^t\mathbf{p}^t$ as the adversarial vectors) to compute $\mathbf{w}^{t+1}$ (so the "reward" accrued by this experts algorithm at time $t+1$ is $\langle \mathbf{w}^{t+1}, \mathbf{A}^{t+1}\mathbf{p}^{t+1} \rangle$.

At time $n/2$, the algorithm restarts the online experts subroutine afresh, and computes the vectors $\mathbf{p}^{n/2+1}, \ldots, \mathbf{p}^n$ and $\mathbf{w}^{n/2+1}, \ldots, \mathbf{w}^n$ in the same way as before. For concreteness, the online experts algorithm we use is a multiplicative-weights update algorithm from [2, Section 2.3], scaling down by M to ensure gains are in $[-1, 1]$, and using ε for the learning rate η.

Note the simplicity of the algorithm: it is perhaps the "natural" algorithm, once we decide to reduce load-balancing to the experts algorithm. The main theorem of this section states the guarantee of this algorithm; given a vector $v \in \mathbb{R}^m$, define $|v| := (|v_1|, \ldots, |v_m|)$.

Theorem 5 (Load Balancing Guarantee). *Suppose $\{A^t\}$ is (M, γ)-well-bounded load-balancing instance for $\gamma \geq 1$. Let λ^* be its optimal load and suppose $\varepsilon \leq \varepsilon_1 := \frac{1}{80}$ and $\delta \in (0, \varepsilon]$ are such that $\lambda^* \geq \frac{3M \log(m/\delta)}{\varepsilon^2}$. Given values of n, M and ε, the algorithm **expertLB** finds an online solution $\{\mathbf{p}^t\}_t$ such that with probability at least $1-\delta$, $\|\sum_t \mathbf{A}^t\mathbf{p}^t - \varepsilon \sum_t |\mathbf{A}^t\mathbf{p}^t|\|_{\max} \leq \lambda^*(1 + c_1(1+\gamma)\varepsilon)$ for a universal constant c_1.*

Moreover, if all A^t's are non-negative, we can take $\gamma = 1$ and then have $\|\sum_t \mathbf{A}^t\mathbf{p}^t\|_\infty \leq (1 + O(\varepsilon))\lambda^*$ with probability at least $1-\delta$.

2.4 Analysis of ExpertLB

In this section we outline the analysis of algorithm **expertLB**. Let $\widehat{p}^1, \ldots \widehat{p}^n$ be the optimal solution for the offline instance. We see this as a mapping from matrix A^t to solution $\widehat{p}^t$ (say $\phi(A^t) = \widehat{p}^t$); this way, define $\widehat{\mathbf{p}}^t$ as the random solution with respect to the random matrix $\mathbf{A}^t$ (i.e. $\widehat{\mathbf{p}}^t = \phi(\mathbf{A}^t)$). (Formally, if there are repeated matrices then we can assume that the optimal solution does the same thing for all of them.)

To simplify the notation, let $\mathbf{o}^t := \mathbf{A}^t\mathbf{p}^t$ denote the load vector incurred at step t by our algorithm; for an integer ℓ, we use $\mathbf{A}^{\leq \ell}$ to denote the sequence $\mathbf{A}^1, \ldots, \mathbf{A}^\ell$, and similarly for other objects.

Theorem 5 seeks to bound $\|\sum_t \mathbf{o}^t - \varepsilon \sum_t |\mathbf{o}^t|\|_{\max}$. By exchangeability of our distribution, $(\mathbf{A}^1, \ldots, \mathbf{A}^{n/2})$ has the same distribution as $(\mathbf{A}^{n/2+1}, \ldots, \mathbf{A}^n)$. This implies that our algorithm behaves in the same in the two halves of the process, namely $(\mathbf{A}^{\leq n/2}, \mathbf{p}^{\leq n/2})$ has the same distribution as $(\mathbf{A}^{> n/2}, \mathbf{p}^{> n/2})$.

Fact 6 (Suffices to Analyze First Half). *The random variables* $\|\sum_{t\leq n/2}\mathbf{o}^t - \varepsilon\sum_{t\leq n/2}|\mathbf{o}^t|\ \|_{\max}$ *and* $\|\sum_{t>n/2}\mathbf{o}^t - \varepsilon\sum_{t>n/2}|\mathbf{o}^t|\ \|_{\max}$ *have the same distribution.*

We want to show that the computed dual solutions $\mathbf{w}^t$ capture our (non-linear) total load, i.e., that $\sum_t \mathbf{w}^t \cdot \mathbf{o}^t \approx \|\sum_t \mathbf{o}^t\|_{\max}$. To this end we formally show:

Fact 7 (Dual Captures Load). *For every scenario,*

$$\sum_{t=1}^{n/2}\left\langle \mathbf{w}^t, \mathbf{o}^t\right\rangle \geq \left\| \sum_{t\leq n/2}\mathbf{o}^t - \varepsilon\sum_{t\leq n/2}|\mathbf{o}^t|\right\|_{\max} - \frac{M\log m}{\varepsilon} \;\;\geq\;\; \left\| \sum_{t\leq n/2}\mathbf{o}^t - \varepsilon\sum_{t\leq n/2}|\mathbf{o}^t|\right\|_{\max} - \varepsilon\lambda^*.$$

Let $\widehat{\mathbf{o}}^t := \mathbf{A}^t\widehat{\mathbf{p}}^t$ denote the load incurred by the optimal solution $\widehat{\mathbf{p}}$ at step t. By our primal greedy choice of the $\mathbf{p}^t$'s, we directly have the following.

Fact 8 (Optimality of Algorithm wrt Duals).

$$\textstyle\sum_{t=1}^{n/2}\langle \mathbf{w}^t, \mathbf{o}^t\rangle = \sum_{t=1}^{n/2}\mathbf{w}^t\mathbf{A}^t\mathbf{p}^t \leq \sum_{t=1}^{n/2}\mathbf{w}^t\mathbf{A}^t\widehat{\mathbf{p}}^t = \sum_{t=1}^{n/2}\langle \mathbf{w}^t, \widehat{\mathbf{o}}^t\rangle.$$

From these facts, in order to give a guarantee in expectation, it suffices to show that

$$\textstyle\mathbb{E}[\sum_{t\leq n/2}\langle \mathbf{w}^t, \widehat{\mathbf{o}}^t\rangle] \lesssim \sum_{t\leq n/2}\langle \mathbb{E}[\mathbf{w}^t], \mathbb{E}[\widehat{\mathbf{o}}^t]\rangle \leq \|\mathbb{E}[\sum_{t\leq n/2}\widehat{\mathbf{o}}^t]\|_{\max} = \frac{\lambda^*}{2}. \quad (1)$$

Notice that these are easy implications if we were sampling *with replacement*, since in that case $\mathbf{w}^t$ and $\widehat{\mathbf{o}}^t$ are independent. For the random permutation model we need to work harder.

Let $\mathcal{G}_{i,t}$ be the good event that $\mathbb{E}[\widehat{\mathbf{o}}^t_i \mid \mathbf{A}^{<t}] \leq (1+80(1+\gamma)\varepsilon)\frac{\lambda^*}{n}$, i.e., that the expected occupation at timestep t is at most $\approx \frac{\lambda^*}{n}$ even after conditioning on the history up to time $t-1$. What we are able to show is that with high probability, the good events $\mathcal{G}_{i,t}$ hold *for all* $t \leq n/2$ simultaneously; effectively, this says that for our purposes, sampling with and without replacement has essentially the same effect. Note that applying Bernstein's inequality to each $\widehat{\mathbf{o}}^t_i$ and taking a union bound over the t's would only give $\mathbb{E}[\widehat{\mathbf{o}}^t_i \mid \mathbf{A}^{<t}] \lesssim (1+\gamma\log n\varepsilon)\frac{\lambda^*}{n}$, with an extra $\log n$ factor. To avoid this we employ a *maximal* version of Bernstein's inequality.

Lemma 2. *With probability* $1-\delta/m$, $\mathcal{G}_{i,t}$ *holds for all* i *and all* $t \leq n/2$*; i.e.,* $\Pr(\bigvee_i \bigvee_{t\leq n/2}\overline{\mathcal{G}}_{i,t}) \leq \delta/m$.

Using this lemma, it is an easy exercise to show that (1) indeed holds (within a factor $(1\pm\gamma\varepsilon)$); this then gives that the bound in Theorem 5 holds in expectation.

Moreover, we can show that *with probability at least* $1-\delta$, $\sum_{t\leq n/2}\langle \mathbf{w}^t, \widehat{\mathbf{o}}^t\rangle \leq \frac{\lambda^*}{2} + O(\varepsilon\gamma)\lambda^*$. For that we need to employ martingale concentration (Freedman's inequality), and use Lemma 2 to control the predictable variation of our martingale. This then gives the full statement of Theorem 5.

3 Solving Packing-Covering LPs via Load-Balancing

We now show how to solve packing-covering LPs using the load-balancing algorithm we developed in Theorem 5, provided an estimate of the optimum is available. The packing-covering LPs (PCLPs) we will solve will be of the following form:

$$\begin{aligned} \max \ & \textstyle\sum_{t=1}^{n} \pi^t x^t && \text{(PCLP)} \\ st \ & \textstyle\sum_{t=1}^{n} A^t x^t \leq b && (2) \\ & \textstyle\sum_{t=1}^{n} C^t x^t \geq d && (3) \\ & x^t \in \blacktriangle^k \quad \forall t \in [n] && (4) \end{aligned}$$

where all the data π^t, A^t, C^t, b, d is **non-negative** and $\blacktriangle^k$ denotes the "full simplex" $\{x \in [0,1]^k : \sum_j x_j \leq 1\}$. We use m_p to denote the number packing constraints (2) and m_c to denote the number of covering constraints (3), so our matrices have dimensions $A^t \in \mathbb{R}_+^{m_p \times k}$, and $C^t \in \mathbb{R}_+^{m_c \times k}$. We allow for either m_c or m_p to be zero—i.e., it could be a pure packing or covering LP. Note that the variables $\{x_1^t, x_2^t, \dots, x_k^t\}$ in each block t are bound together by $\sum_j x_j^t \leq 1$. Given a packing-covering LP $\mathcal{L}$, we use $\mathsf{opt}(\mathcal{L})$ to denote its optimal value; we omit $\mathcal{L}$ when it is clear from the context.

In the *online* version of PCLP, we again present a fixed PCLP $\mathcal{L}$ in random order. We know upfront the number of time steps n and the right-hand sides b, d. At each time $t = 1, \dots, n$, a "block" is sampled from $\mathcal{L}$ without replacement and revealed to the algorithm, which then outputs a vector $\mathbf{x}^t \in \blacktriangle^k$. Formally, define the random variables $\boldsymbol{\pi}^t, \mathbf{A}^t, \mathbf{C}^t$ so that triples $(\boldsymbol{\pi}^1, \mathbf{A}^1, \mathbf{C}^1), \dots, (\boldsymbol{\pi}^n, \mathbf{A}^n, \mathbf{C}^n)$ are sampled from $\{(\pi^t, A^t, C^t)\}_t$ uniformly *without replacement*. Define the randomly permuted LP $\boldsymbol{\mathcal{L}}$

$$\boldsymbol{\mathcal{L}} = \max \left\{ \sum_t \boldsymbol{\pi}^t x^t : \sum_t \mathbf{A}^t x^t \leq b, \ \sum_t \mathbf{C}^t x^t \geq d, \ x^t \in \blacktriangle^k \ \ \forall t \right\} \tag{5}$$

At time step t, the algorithm computes a (random) vector $\mathbf{x}^t \in \blacktriangle^k$ based on the information seen up to time t, i.e., $(\boldsymbol{\pi}^1, \mathbf{A}^1, \mathbf{C}^1), \dots, (\boldsymbol{\pi}^t, \mathbf{A}^t, \mathbf{C}^t)$, plus n and b, d. We call such $\{\mathbf{x}^t\}_t$ an *online solution*. The online solution $\{\mathbf{x}^t\}$ is ϵ*-feasible* for $\boldsymbol{\mathcal{L}}$ if it satisfies the packing constraints (i.e. $\sum_{t \leq n} \mathbf{A}^t \mathbf{x}^t \leq b$) and *almost* satisfies the covering constraints (i.e. $\sum_{t \leq n} \mathbf{C}^t \mathbf{x}^t \geq (1 - \epsilon) d$). The goal in the online PCLP is to obtain an online solution $\{\mathbf{x}^t\}$ which (with high probability) is ϵ-feasible, and gets a reward $\sum_{t \leq n} \boldsymbol{\pi}^t \mathbf{x}^t \geq (1 - O(\varepsilon))\mathsf{opt}(\boldsymbol{\mathcal{L}})$ (notice that $\mathsf{opt}(\boldsymbol{\mathcal{L}}) = \mathsf{opt}(\mathcal{L})$, so again we are comparing against the optimal offline solution).

Before we state our main result for this section, we need one more concept. For a packing-covering LP $\mathcal{L}$ of the form (PCLP), define its *generalized width* to be

$$\max \left\{ \max_{i,j,t} \frac{a_{i,j}^t}{b_i}, \quad \max_{i,j,t} \frac{c_{i,j}^t}{d_i}, \quad \max_{t,j} \frac{\pi_j^t}{\mathsf{opt}} \right\}. \tag{6}$$

Theorem 9. *Consider a feasible (multiple-choice) packing-covering LP $\mathcal{L}$, and let $\varepsilon \leq \frac{\varepsilon_1}{4}$ and $\delta \in (0, \varepsilon]$ be such that its generalized width is at most $\frac{\varepsilon^2}{\log(m/\delta)}$. Suppose the algorithm is given an approximation $\widehat{\mathsf{opt}}$ to the optimal value, as well as the right-hand sides b, d and values ε, δ, n. If the estimate $\widehat{\mathsf{opt}} \in [\frac{\mathsf{opt}}{2}, \mathsf{opt}]$, then the algorithm **LPviaLB** below finds an online solution $\{\mathbf{x}^t\}$ that with probability at least $1 - \delta$ is $(c_2\varepsilon)$-feasible for $\boldsymbol{\mathcal{L}}$ and has value $\sum_t \boldsymbol{\pi}^t \mathbf{x}^t \geq (1 - c_3\varepsilon)\widehat{\mathsf{opt}}$, for constants $c_2, c_3 \geq 1$.*

3.1 The Algorithm LPviaLB

The idea of the reduction is very simple: take the covering constraints (and the objective function) and flip their signs to make them packing constraints; also shift these negated values by adding a constant to both sides and then rescale the constraints so that all of them have same right-hand side. Notice that this creates negative entries in the left-hand side (negative loads); this is precisely why we had considered the load-balancing problem in this generality in the previous section.

First, a minor detail: we can assume that $\varepsilon^2 \geq \frac{\log(m/\delta)}{n}$. Indeed, if $\varepsilon^2 < \frac{\log(m/\delta)}{n}$ then the assumption that the generalized width is at most $\frac{\varepsilon^2}{\log(m/\delta)}$ implies $c^t_{i,j} < \frac{d_i}{n}$ for all t, i; this means the covering constraints, if any, cannot be satisfied and the LP is infeasible. On the other hand, if $m_c = 0$ and there are no covering constraints, then another implication of the low value of ε is that $a^t_{i,j} < \frac{b_i}{n}$; in this case the packing constraints are so loose that the optimal solution is to choose the most profitable choice for each time t independently.

Given a packing-covering LP $\mathcal{L}$, define the matrices $H^1, \ldots, H^n$ with $k+1$ columns and $m_p + m_c + 1$ rows (indexed from 0 to $m := m_p + m_c$) as follows: the zeroth row of H^t equals the vector

$$H^t_{0,\star} := \left(\tfrac{2}{n} - \tfrac{\pi^t_1}{\widehat{\mathsf{opt}}}, \ldots, \tfrac{2}{n} - \tfrac{\pi^t_k}{\widehat{\mathsf{opt}}}, \tfrac{2}{n}\right) ;$$

for $i \in \{1, \ldots, m_p\}$, the i^{th} row of H^t is

$$H^t_{i,\star} := \left(\tfrac{a^t_{i1}}{b_i}, \ldots, \tfrac{a^t_{ik}}{b_i}, 0\right),$$

and for $i \in \{m_p + 1, \ldots, m_p + m_c\}$, the $(m_p + i)^{th}$ row of H^t is

$$H^t_{m_p+i,\star} := \left(\tfrac{2}{n} - \tfrac{c^t_{i1}}{d_i}, \ldots, \tfrac{2}{n} - \tfrac{c^t_{ik}}{d_i}, \tfrac{2}{n}\right).$$

(For simplicity, we assume that $\widehat{\mathsf{opt}}$ and all b_i, d_is are strictly positive.)

The algorithm **LPviaLB** can be thought of as having three phases. In the first phase, it computes the matrices $H^1, \ldots, H^n$. In the second, it runs load-balancing algorithm **expertLB** over the instance $\{H^t\}_t$ with parameters $M = \frac{2\varepsilon^2}{\log(m/\delta)}$, $\varepsilon' = 4\varepsilon$ and δ (given), obtaining a solution $\{\tilde{\mathbf{x}}^t\}_t$. In the last phase, the algorithm simply outputs the scaled and truncated solution $\{\widehat{\mathbf{x}}^t\}_t$, for $\widehat{\mathbf{x}}^t_j :=$

$\frac{(1-2\varepsilon')}{(1+c_1\varepsilon')}\tilde{\mathbf{x}}_j^t$ with $j = 1, \ldots, k$, as an (approximate) solution to $\boldsymbol{\mathcal{L}}$, where c_1 is the constant from Theorem 5. Notice that all the steps in this algorithm can be implemented to run in an online manner.

The analysis of **LPviaLB** relies on: 1) making sure the assumptions of Theorem 5 are satisfied by the load-balancing instance $\{H^t\}_t$, and 2) connecting feasible solutions for the instance $\{H^t\}_t$ to solutions of the LP $\mathcal{L}$. These give Theorem 9.

4 Solving Packing/Covering LPs with Unknown OPT

We now turn to the problem of estimating the value of opt for a packing-covering problem, so that we can use it in Theorem 9. For that, we will require a kind of *stability property*, loosely asking that the covering constraints are not "very tight".

Definition 2 (Stability). *A packing-covering LP $\mathcal{L}$ is called* (ε, σ)-stable *if any optimal ε-feasible solution for it has value at most $(1 + \sigma\varepsilon)\mathsf{opt}(\mathcal{L})$.*

For a randomly permuted LP $\boldsymbol{\mathcal{L}}$, the basic idea to obtain an estimate of $\mathsf{opt}(\boldsymbol{\mathcal{L}})$ is to see the first $n/2$ random blocks and form a *sampled LP* with these blocks and right-hand side scaled by a factor of $1/2$; computing an optimal $O(\varepsilon)$-solution for this sampled LP should give a good estimate of $\mathsf{opt}(\boldsymbol{\mathcal{L}})/2$.

Definition 3 (Restricted LP). *Given a packing-covering LP $\mathcal{L}$, and a subset I of $[n]$, the restricted LP $\mathcal{L}^I$ is obtained by retaining only the columns of $\mathcal{L}$ that belong to I, and setting the right hand side $\mathsf{rhs}(\mathcal{L}^I)$ to be $\frac{|I|}{n}\mathsf{rhs}(\mathcal{L})$.*

For a (ε, σ)-stable $\mathcal{L}$, we show that an $O(\varepsilon)$-optimal solution for the sampled LP $\boldsymbol{\mathcal{L}}^{\{1,\ldots,\frac{n}{2}\}}$ is, with probability at least $1 - \delta$, is about $\frac{1}{2}\mathsf{opt}(\mathcal{L}) \pm \sigma\varepsilon\mathsf{opt}(\mathcal{L})$; while the lower bound is a straightforward application of Bernstein's inequality, for the upper bound we uses (ε, σ)-stability to control the variance of the (dual of the) sampled LP.

Given this bound, we can estimate the optimum of the remaining LP $\boldsymbol{\mathcal{L}}^{>\frac{n}{2}}$ using the sampled LP $\mathsf{opt}(\boldsymbol{\mathcal{L}}^{\leq\frac{n}{2}})$ and employ Theorem 9 to the former to obtain the following.

Theorem 10. *Let $\varepsilon \leq \frac{\varepsilon_1}{4}$, and $\delta \in (0, \varepsilon]$. Consider a packing-covering LP $\mathcal{L}$ with generalized width at most $\frac{\varepsilon^2}{32\log(m/\delta)}$ and that is (ε, σ)-stable for $\sigma \in [1, \frac{1}{12\sqrt{2}\varepsilon}]$. Suppose that the number n of columns of $\mathcal{L}$, the right-hand sides of $\mathcal{L}$, ε, δ and σ are known a priori. Then with probability at least $1 - 5\delta$ we can find an online $\varepsilon(\sqrt{2} + c_2)$-feasible solution to the random LP $\boldsymbol{\mathcal{L}}^{>\frac{n}{2}}$ with value at least*

$$(1 - c_3\varepsilon)\,\mathsf{opt}(\boldsymbol{\mathcal{L}}^{\leq\frac{n}{2}}) - \varepsilon\,(5\sqrt{2}\sigma)\,\mathsf{opt}(\boldsymbol{\mathcal{L}}).$$

To avoid the total loss of value from the first $n/2$ columns of the LP, instead of applying the above theorem directly to input LP, we apply it to the sub-LPs of doubling sizes $\boldsymbol{\mathcal{L}}^{\leq 2\varepsilon n}$, $\boldsymbol{\mathcal{L}}^{\leq 4\varepsilon n}$, $\boldsymbol{\mathcal{L}}^{\leq 8\varepsilon n}$, etc. to obtain a sequence of "disjoint" solutions which, when put together, give an $O(\varepsilon)$-feasible solution for the whole LP. The total loss in value is about $\mathsf{opt}(\boldsymbol{\mathcal{L}}^{\leq \varepsilon n}) + \sigma \log \varepsilon^{-1} \mathsf{opt}(\mathcal{L})$. This leads to the following result.

Theorem 11. *Let* $\varepsilon \leq (\frac{\varepsilon_1}{4})^2$*, and* $\delta \in (0, \varepsilon]$*. Consider a packing-covering LP* $\mathcal{L}$ *with generalized width at most* $\frac{\varepsilon^2}{32 \log(m/\delta)}$ *and that is* (ε, σ)*-stable for* $\sigma \in [1, \frac{1}{12\sqrt{2}\sqrt{\varepsilon}}]$*. Suppose that the number* n *of columns of* $\mathcal{L}$*, the right-hand sides of* $\mathcal{L}$*,* ε*,* δ *and* σ *are known a priori. Then we can find an online solution to the random LP* $\boldsymbol{\mathcal{L}}$ *that is* $\varepsilon(\sqrt{2} + c_2)$*-feasible with probability at least* $1 - 5\delta \log \varepsilon^{-1}$*, and has expected value at least* $(1 - O(\sigma\varepsilon + \delta \log \varepsilon^{-1}))\mathsf{opt}(\boldsymbol{\mathcal{L}})$*.*

Note: *Most proofs, as well as a section proving the Informal Theorem 4, are presented in the final version of the paper.*

References

1. Agrawal, S., Wang, Z., Ye, Y.: A dynamic near-optimal algorithm for online linear programming. CoRR abs/0911.2974 (2009), to appear in Opertaions Research
2. Arora, S., Hazan, E., Kale, S.: The multiplicative weights update method: a meta-algorithm and applications. Theory of Computing 8(6), 121–164 (2012)
3. Azar, Y., Bhaskar, U., Fleischer, L., Panigrahi, D.: Online mixed packing and covering. In: SODA. pp. 85–100 (2013)
4. Babaioff, M., Immorlica, N., Kempe, D., Kleinberg, R.: A knapsack secretary problem with applications. In: APPROX-RANDOM (2007)
5. Buchbinder, N., Naor, J.: The design of competitive online algorithms via a primal-dual approach. Foundations and Trends in Theoretical Computer Science 3(2-3), 93–263 (2009)
6. Clarkson, K.L., Hazan, E., Woodruff, D.P.: Sublinear optimization for machine learning. J. ACM 59(5), 23:1–23:49 (2012)
7. Devanur, N.R., Jain, K., Sivan, B., Wilkens, C.A.: Near optimal online algorithms and fast approximation algorithms for resource allocation problems. In: EC (2011)
8. Devenur, N.R., Hayes, T.P.: The adwords problem: online keyword matching with budgeted bidders under random permutations. In: EC (2009)
9. Diaconis, P., Freedman, D.: Finite exchangeable sequences. The Annals of Probability 8(4), 745–764 (1980)
10. Dynkin, E.B.: The optimum choice of the instant for stopping a Markov process. Soviet Math. Dokl 4 (1963)
11. Feldman, J., Henzinger, M., Korula, N., Mirrokni, V.S., Stein, C.: Online stochastic packing applied to display ad allocation. In: ESA (2010)
12. Garg, N., Könemann, J.: Faster and simpler algorithms for multicommodity flow and other fractional packing problems. SIAM J. Comput. 37(2) (2007)
13. Goel, G., Mehta, A.: Online budgeted matching in random input models with applications to adWords. In: SODA (2008)
14. Hazan, E.: Approximate convex optimization by online game playing. Tech. rep.
15. Kesselheim, T., Radke, K., Tönnis, A., Vöcking, B.: Primal beats dual on online packing lps in the random-order model. CoRR abs/1311.2578 (2013), to appear in STOC 2014

16. Kleinberg, R.: A multiple-choice secretary algorithm with applications to online auctions. In: SODA (2005)
17. Mehta, A., Saberi, A., Vazirani, U., Vazirani, V.: AdWords and generalized online matching. J. ACM 54(5) (2007)
18. Molinaro, M., Ravi, R.: Geometry of online packing linear programs. In: ICALP (2012)
19. Plotkin, S.A., Shmoys, D.B., Tardos, É.: Fast approximation algorithms for fractional packing and covering problems. Math. Oper. Res. 20(2), 257–301 (1995)
20. Simpson, R.W.: Using network flow techniques to find shadow prices for market and seat inventory control (1989), MIT Memorandum M89-1
21. Telluri, K., van Ryzin, G.: An Analysis of Bid-Price Controls for Network Revenue Management. Management Science 44, 1577–1593 (1998)
22. Williamson, E.L.: Airline network seat inventory control : methodologies and revenue impacts (1992), Ph. D. Thesis, MIT
23. Young, N.E.: Sequential and parallel algorithms for mixed packing and covering. In: FOCS (2001)

Parameterized Complexity of the k-Arc Chinese Postman Problem

Gregory Gutin, Mark Jones, and Bin Sheng

Royal Holloway, University of London
Egham, Surrey TW20 0EX, UK

Abstract. In the Mixed Chinese Postman Problem (MCPP), given an edge-weighted mixed graph G (G may have both edges and arcs), our aim is to find a minimum weight closed walk traversing each edge and arc at least once. The MCPP parameterized by the number of edges was known to be fixed-parameter tractable using a simple argument. Solving an open question of van Bevern et al., we prove that the MCPP parameterized by the number of arcs is also fixed-parameter tractable. Our proof is more involved and, in particular, uses a very useful result of Marx, O'Sullivan and Razgon (2013) on the treewidth of torso graphs with respect to small separators.

1 Introduction

A *mixed graph* is a graph that may contain both edges and arcs (i.e., directed edges). A mixed graph G is *strongly connected* if for each ordered pair x, y of vertices in G there is a path from x to y that traverses each arc in its direction. We provide further definitions and notation on (mainly) directed graphs in the next section.

In this paper, we will study the following problem.

Mixed Chinese Postman Problem (MCPP)
Instance: A strongly connected mixed graph $G = (V, E \cup A)$, with vertex set V, set E of edges and set A of arcs; a weight function $w : E \cup A \to \mathbb{N}_0$.
Output: A closed walk of G that traverses each edge and arc at least once, of minimum weight.

There is numerous literature on various algorithms and heuristics for MCPP; for informative surveys, see [2,4,8,12,14]. When $A = \emptyset$, we call the problem the Undirected Chinese Postman Problem (UCPP), and when $E = \emptyset$, we call the problem the Directed Chinese Postman Problem (DCPP). It is well-known that UCPP is polynomial-time solvable [7] and so is DCPP [3,5,7], but MCPP is NP-complete, even when G is planar with each vertex having total degree 3 and all edges and arcs having weight 1 [13]. It is therefore reasonable to believe that MCPP may become easier the closer it gets to UCPP or DCPP.

Van Bevern *et al.* [2] considered two natural parameters for MCPP: the number of edges and the number of arcs. They showed that MCPP is fixed-parameter tractable[1] (FPT) when parameterized by the number k of edges. Their algorithm is as follows. Replace every undirected edge uv by either the arc $\overrightarrow{uv}$ or arc $\overrightarrow{vu}$ or the pair $\overrightarrow{uv}$ and $\overrightarrow{vu}$ (all arc have the same weight as uv) and solve the resulting DCPP. Thus, the MCPP can be solved in time $O(3^k n^3)$, where n is the number of vertices in G. We describe a faster algorithm in the full version of this paper [10].

Van Bevern *et al.* [2] and Sorge [15] left it as an open question whether the MCPP is fixed-parameter tractable when parameterized by the number of arcs.

k-arc Chinese Postman Problem (k-arc CPP)
Instance: A strongly connected weighted mixed graph $G = (V, E \cup A)$, with vertex set V, set E of edges and set A of arcs; a weight function $w : E \cup A \to \mathbb{N}_0$.
Parameter: $k = |A|$.
Output: A closed walk of G that traverses each edge and arc at least once, of minimum weight.

This parameterized problem is of interest, for example, if we view the mixed graph as a network of streets in a city: while edges represent two-way streets, arcs are for one-way streets. Many cities have a relatively small number of one-way streets and so the number of arcs appears to be a good parameter for optimizing, say, police patrol in such cities [2].

We will assume for convenience that the input G of k-arc CPP is a *simple* graph, i.e. there is at most one edge or one arc (but not both) between any pair of vertices. The multigraph version of the problem may be reduced to the simple graph version by subdividing arcs and edges. As the number of arcs and edges is at most doubled by this reduction, this does not affect the parameterized complexity of the problem.

We will show that k-arc CPP is fixed-parameter tractable. Our proof is significantly more complicated than the one for the MCPP parameterized by the number of edges as a similar approach will not work. Instead, in FPT time, we reduce the problem to the Balanced Chinese Postman Problem (BCPP), in which there are no arcs but instead a demand function on the imbalance of the vertices. Unfortunately this problem is still NP-hard, and so we must use further techniques to solve the problem.

Marx, O'Sullivan and Razgon [11] use the following notion of a graph torso. Let $G = (V, E)$ be a graph and $S \subseteq V$. The graph torso(G, S) has vertex set S and vertices $a, b \in S$ are connected by an edge ab if $ab \in E$ or there is a path in G connecting a and b whose internal vertices are not in S.

Marx *et al.* [11] show that for a number of graph separation problems, it is possible to derive a graph closely related to a torso graph, which has the same separators as the original input graph. The separation problem can then be

[1] That is, MCPP can be solved in time $f(k)n^{O(1)}$, where f is a function only depending on k, and n is the number of vertices in G. For background and terminology on parameterized complexity we refer the reader to [6].

solved on this new graph, which has bounded treewidth. By contrast, we use the torso graph as a tool to construct a tree decomposition of the original graph, which does not have bounded width, but has enough other structural restrictions to make a dynamic programming algorithm possible. So, our application of Marx *et al.*'s result is quite different from its use in [11], and we believe it may be used for designing fixed-parameter algorithms for other problems on graphs. Note that Marx *et al.* are interested in small separators (i.e. sets of vertices whose removal disconnects a graph), whereas we are interested in small cuts (sets of edges whose removal disconnects a graph). This necessitates an extra step in the construction of our tree decomposition, to ensure that all minimal cuts are covered by minimal separators.

The rest of the paper is organized as follows. The next section contains further terminology and notation. In Section 3, we reduce k-ARC CPP to BALANCED CHINESE POSTMAN PROBLEM (BCPP). In Section 4, we introduce and study two key notions that we use to solve BCPP: t-roads and small t-cuts. In Section 5, we investigate a special tree decomposition of the input graph of BCPP. This decomposition is used in a dynamic programming algorithm given in Section 6. The last section contains some conclusions and open problems.

2 Further Terminology and Notation

For a positive integer p and an integer q, $q < p$, $[q, p]$ will denote the set $\{q, q+1, \dots, p\}$ and $[p]$ the set $[1, p]$. To avoid confusion, we denote an edge between two vertices u, v as uv, and an arc from u to v as $\overrightarrow{uv}$.

For a mixed multigraph G, let D be a directed multigraph derived from G by replacing each arc $\overrightarrow{uv}$ of G with multiple copies of $\overrightarrow{uv}$ (at least one), and replacing each edge in uv in G with multiple copies of the arcs $\overrightarrow{uv}$ and $\overrightarrow{vu}$ (such that there is at least one copy of $\overrightarrow{uv}$ or at least one copy of $\overrightarrow{vu}$). Then we say D is a *multi-orientation* of G. If D has exactly one copy $\overrightarrow{uv}$ for each arc $\overrightarrow{uv}$ in G, and either exactly one copy of $\overrightarrow{uv}$ or exactly one copy of $\overrightarrow{vu}$ for each edge uv in G, we say D is an *orientation* of G. If D is an orientation of G, we say that G is the *undirected version* of D (if D has parallel arcs then G has parallel edges).

For a mixed multigraph G, $\mu_G(\overrightarrow{uv})$ denotes the number of arcs of the form $\overrightarrow{uv}$ in G, and $\mu_G(uv)$ denotes the number of edges of the form uv. For a weighted graph G and a multi-orientation D of G, the *weight* of D is the sum of the weights of all its arcs, where the weight of an arc in D is the weight of the corresponding edge or arc in G.

For a directed multigraph $D = (V, A)$ and $v \in V$, $d^+_D(v)$ and $d^-_D(v)$ denote the out-degree and in-degree of v in D, respectively. Let $t : V \rightarrow \mathbb{Z}$ be a function. We say that a vertex u in D is *t-balanced* if $d^+_D(u) - d^-_D(u) = t(u)$. We say that D is *t-balanced* if every vertex is t-balanced. Note that if D is t-balanced then $\sum_{v \in V} t(v) = 0$. When $t(v) = 0$ for all $v \in V$, we omit t and speak of *balanced* vertices and *balanced* directed multigraphs. Let $V^+_t = \{v \in V : \ t(v) > 0\}$ and $V^-_t = \{v \in V : \ t(v) < 0\}$.

In directed multigraphs, all walks (in particular, paths and cycles) that we consider are directed. A directed multigraph D is *Eulerian* if there is a closed walk of D traversing every arc exactly once. It is well-known that a directed multigraph D is Eulerian if and only if D is balanced and the undirected version of D is connected [1].

For an undirected graph $G = (V, E)$ and vertices a, b of G, a set S of edges (vertices, respectively) is called an (a, b)-*cut* ((a, b)-*separator*, respectively) if a and b are in different components of $G - S$.

Observe that the following is an equivalent formulation of the k-ARC CPP.

k-ARC CHINESE POSTMAN PROBLEM (k-ARC CPP)
Instance: A strongly connected mixed graph $G = (V, E \cup A)$, with vertex set V, set E of edges and set A of arcs; weight function $w : E \cup A \to \mathbb{N}_0$.
Parameter: $k = |A|$.
Output: A directed multigraph D of minimum weight such that D is a multi-orientation of G and D is Eulerian.

3 Reduction to Balanced CPP

Our first step is to reduce k-ARC CPP to a problem on a graph without arcs. Essentially, given a graph $G = (V, E \cup A)$ we will "guess" the number of times each arc in A is traversed in an optimal solution. This then leaves us with a problem on $G' = (V, E)$. Rather than trying to find an Eulerian multi-orientation of G, we now try to find a multi-orientation of G' in which the imbalance between the in- and out-degrees of each vertex depends on the guesses for the arcs in A incident with that vertex.

More formally, we will provide a Turing reduction to the following problem:

BALANCED CHINESE POSTMAN PROBLEM (BCPP)
Instance: An undirected graph $G = (V, E)$; a weight function $w : E \to \mathbb{N}_0$; a demand function $t : V \to \mathbb{Z}$ such that $\sum_{v \in V} t(v) = 0$.
Parameter: $p = \sum_{v \in V_t^+} t(v)$.
Output: A minimum weight t-balanced multi-orientation D of G.

Observe that when $t(v) = 0$ for all $v \in V$, BCPP is equivalent to UCPP. BCPP was studied by Zaragoza Martínez [16] who proved that the problem is NP-hard. We will reduce k-ARC CPP to BCPP by guessing the number of times each arc is traversed. In order to ensure a fixed-parameter aglorithm, we need a bound (in terms of $|A|$) on the number guesses. We will do this by bounding the total number of times any arc can be traversed in an optimal solution.

Lemma 1. *Let $G = (V, A \cup E)$ be a mixed graph, and let $k = |A|$. Then for any optimal solution D to k-*ARC *CPP on G with minimal number of arcs, we have that $\sum_{\overrightarrow{uv} \in A} \mu_D(\overrightarrow{uv}) \leq k^2/2 + 2k$.*

Proof. Let $A = A_1 \cup A_2$ where $A_1 = \{\overrightarrow{uv} : \overrightarrow{uv} \in A \text{ and } \mu_D(\overrightarrow{uv}) \geq 3\}$ and $A_2 = A \setminus A_1$. Let $|A_1| = p$ and $|A_2| = k - p = q$.

Consider an arc $\overrightarrow{uv} \in A$. Since D is balanced, we have that D has $\mu_D(\overrightarrow{uv})$ arc-disjoint directed cycles, each containing exactly one copy of $\overrightarrow{uv}$. We claim that each such cycle must contain at least one copy of an arc in A_2. Indeed, otherwise, there is a cycle C containing $\overrightarrow{uv}$ that does not contain any arc in A_2, which means that C consists of arcs in A_1 and arcs corresponding to (undirected) edges in G. We may construct a directed multigraph D' as follows: Remove from D two copies of each arc in A_1 that appears in C, and reverse the arcs in C that correspond to undirected edges in G. Observe that D' is Eulerian and is also a multi-orientation of G, and so D' is a solution with smaller weight than D or an optimal solution with fewer arcs than D, contradicting the minimality of D.

So each of the $\mu_D(\overrightarrow{uv})$ cycles contains at least one copy of an arc in A_2. Observe that D has at most $2q$ copies of arcs in A_2, and so $\mu_D(\overrightarrow{uv}) \leq 2q$. Thus, we have $\sum_{\overrightarrow{uv}\in A} \mu_D(\overrightarrow{uv}) = \sum_{\overrightarrow{uv}\in A_1} \mu_D(\overrightarrow{uv}) + \sum_{\overrightarrow{uv}\in A_2} \mu_D(\overrightarrow{uv}) \leq p \cdot 2q + 2q \leq 2 \cdot (\frac{p+q}{2})^2 + 2k = k^2/2 + 2k$. □

Now we may prove the following:

Lemma 2. *If* BCPP *is FPT then so is* k-ARC CPP.

Proof. Let $(G = (V, A \cup E), w)$ be an instance of k-ARC CPP, and let $k = |A|$. Let $\kappa = \lfloor k^2/2 + 2k \rfloor$. By Lemma 1, $\sum_{\overrightarrow{uv}\in A} \mu_D(\overrightarrow{uv}) \leq \kappa$ for any optimal solution D to k-ARC CPP on (G, w) with minimal number of arcs.

Let $G' = (V, E)$ and let w' be w restricted to E. Given a function $\phi : A \to [\kappa]$ such that $\sum_{\overrightarrow{uv}\in A} \phi(\overrightarrow{uv}) \leq \kappa$, let $t_\phi : V \to [-\kappa, \kappa]$ be the function such that $t_\phi(v) = \sum_{\overrightarrow{uv}\in A} \phi(\overrightarrow{uv}) - \sum_{\overrightarrow{vu}\in A} \phi(\overrightarrow{vu})$ for all $v \in V$. Observe that $\sum_{v\in V^+_{t_\phi}} t_\phi(v) \leq \sum_{\overrightarrow{uv}\in A} \phi(\overrightarrow{uv}) \leq \kappa$, and thus BCPP on (G', w', t_ϕ) has parameter $p_\phi \leq \kappa$.

Observe that given a solution D_ϕ to BCPP on (G', w', t_ϕ), if we add $\phi(\overrightarrow{uv})$ copies of each arc $\overrightarrow{uv} \in A$ to D_ϕ, then the resulting graph D is a solution to k-ARC CPP on (G, w) with weight $w'(D_\phi) + \sum_{\overrightarrow{uv}\in A} \phi(\overrightarrow{uv})w(\overrightarrow{uv})$. Furthermore for any solution D to k-ARC CPP on (G, w), let $\phi(\overrightarrow{uv}) = \mu_D(\overrightarrow{uv})$ for each $\overrightarrow{uv} \in A$ and let D_ϕ be D restricted to E. Then D_ϕ is a solution to BCPP on (G', w', t_ϕ) and D has weight $w'(D_\phi) + \sum_{\overrightarrow{uv}\in A} \phi(\overrightarrow{uv})w(\overrightarrow{uv})$.

Suppose that there exists an algorithm which finds the optimal solution to an instance of BCPP on (G', w', t') with parameter p in time $f(p)|V|^{O(1)}$. There are at most $\binom{q}{k}$ ways of choosing positive integers $x_1, \ldots, x_k$ such that $\sum_{i\in[k]} x_i \leq q$. Indeed, for each $i \in [k]$ let $y_i = \sum_{j=1}^{i} x_j$. Then $y_i < y_j$ for $i < j$ and $y_i \in [q]$ for all i, and for any such choice of $y_1, \ldots, y_k$ there is corresponding choice of $x_1, \ldots, x_k$ satisfying $\sum_{i=1}^{k} x_i \leq q$. Therefore the number of valid choices for $x_1, \ldots, x_k$ is the number of ways of choosing $y_1, \ldots, y_k$, which is the number of ways of choosing k elements from a set of q elements.

Therefore there are at most $\binom{\kappa}{k}$ choices for a function $\phi : A \to [\kappa]$ such that $\sum_{\overrightarrow{uv}\in A} \phi(\overrightarrow{uv}) \leq \kappa$. Each choice leads to an instance of BCPP with parameter at most κ. Therefore in time $\binom{\kappa}{k} f(\kappa)|V|^{O(1)}$ we can find the optimal solution D_ϕ to BCPP on (G', w', t_ϕ) for every valid choice of ϕ.

It then remains to choose the function ϕ that minimizes $w'(D_\phi) + \sum_{\overrightarrow{uv}\in A} \phi(\overrightarrow{uv})w(\overrightarrow{uv})$, and return the graph D_ϕ together with $\phi(\overrightarrow{uv})$ copies of each arc $\overrightarrow{uv} \in A$. $\square$

Due to Lemma 2, we may now focus on BCPP.

4 *t*-Roads and *t*-Cuts

Lemma 3. *Let (G, w, t) be an instance of* BCPP, *with $p = \sum_{v\in V_t^+} t(v)$. Then for any optimal solution D to* BCPP *on (G, w, t) with minimal number of arcs, we have that $\mu_D(\overrightarrow{uv}) + \mu_D(\overrightarrow{vu}) \leq \max\{p, 2\}$ for each edge uv in G.*

Proof. Suppose that $\mu_D(\overrightarrow{uv}) + \mu_D(\overrightarrow{vu}) > \max\{p, 2\}$ for some edge uv in G. Observe that if $\mu_D(\overrightarrow{uv}) \geq 1$ and $\mu_D(\overrightarrow{vu}) \geq 1$, then by removing one copy of $\overrightarrow{uv}$ and one copy of $\overrightarrow{vu}$, we obtain a solution to BCPP on (G, w, t) with weight at most that of D but with fewer arcs. Therefore, we may assume that $\mu_D(\overrightarrow{uv}) > \max\{p, 2\}$ and $\mu_D(\overrightarrow{vu}) = 0$.

We now show that there must exist a cycle in D containing a copy of $\overrightarrow{uv}$.

Modify D by adding a new vertex x, with $t(v)$ arcs from x to v for each $v \in V_t^+$, and $-t(v)$ arcs from v to x for each $v \in V_t^-$. Let D^* be the resulting directed graph. Then observe that D^* is balanced, and therefore D^* has $\mu_D(\overrightarrow{uv})$ arc-disjoint cycles, each containing exactly one copy of $\overrightarrow{uv}$. At most p of these cycles can pass through x. Therefore there is at least one cycle containing $\overrightarrow{uv}$ which is a cycle in D.

So now let $v = v_1, v_2, \ldots, v_l = u$ be a sequence of vertices such that $\mu_D(\overrightarrow{v_iv_{i+1}}) \geq 1$ for each $i \in [l-1]$. Replace one copy of each arc $\overrightarrow{v_iv_{i+1}}$ with a copy of $\overrightarrow{v_{i+1}v_i}$ and remove 2 copies of $\overrightarrow{uv}$. Observe that the resulting graph covers every edge of G, and the imbalance of each vertex is the same as in D. Therefore, we have a solution to BCPP on (G, w, t) with weight at most that of D but with fewer arcs. This contradiction proves the lemma. $\square$

Definition 1. *Let $H = (V, E)$ be an undirected multigraph and t a function $V \to \mathbb{Z}$. A* t-road *is a directed multigraph T such that for each vertex v, $d_T^+(v) - d_T^-(v) = t(v)$. We say H* has a t-road T *if there is a subgraph H' of H such that T is an orientation of H'.*

Observe that given a solution D to the BCPP on (G, w, t), the undirected version of D has a t-road.

Definition 2. *Let $H = (V, E(H))$ be an undirected multigraph and $t : V \to \mathbb{Z}$ a function such that $\sum_{v\in V} t(v) = 0$. Let H^* be the multigraph derived from H by creating two new vertices a, b, with $t(v)$ edges between a and v for each $v \in V_t^+$, and $-t(v)$ edges between b and v for each $v \in V_t^-$. Let $p = \sum_{v\in V_t^+} t(v)$. Then a* small t-cut *is a set of edges $F \subseteq E(H)$ such that $F = E(H) \cap F'$ for some minimal (a, b)-cut F' of H^* and $|F'| < p$.*

Note that a small t-cut can be the empty set. A t-road, if one exists, can be found in polynomial time by computing a flow of value p from a to b in the unit capacity network N with underlying multigraph H^*. The next lemma follows from the well-known max-flow-min-cut theorem for N.

Lemma 4. *An undirected multigraph H has a t-road if and only if H does not have a small t-cut.*

Let $(G = (V, E), w, t)$ be an instance of BCPP, and let F be the union of all small t-cuts in G. We say a t-road T is *well-behaved* if $\mu_T(\overrightarrow{uv}) + \mu_T(\overrightarrow{vu}) \leq 1$ for all $uv \in E \setminus F$.

Lemma 5. *Let D be an optimal solution to* BCPP *on $(G = (V, E), w, t)$, and let H be the undirected version of D. Then H has a well-behaved t-road.*

Proof. Let $F \subseteq E$ be the union of all small t-cuts in G. Let J be the undirected multigraph derived from H by removing all but one copy of every edge in $E \setminus F$. Observe that every t-road in J is also a t-road in H and every t-road in J is well-behaved. So, it is sufficient to show that J has a t-road.

Note that if J does not have a t-road, then by Lemma 4 J has a small t-cut. Note also that by construction H has a t-road and therefore does not have a small t-cut. Consider a small t-cut S in J and suppose that every edge in S is a copy of an edge in F. As S is not a small t-cut in H, there are vertices $u \in V_t^+$ and $v \in V_t^-$ such that $H \setminus S$ contains a path $v_1 v_2 \dots v_l$, where $v_1 = u$ and $v_l = v$. Note that $v_1 \dots v_l$ is also a path in $J \setminus S$, unless all copies of the edge $v_i v_{i+1}$ are in S for some $i \in [l-1]$. However, as $S \subseteq F$, if all copies of $v_i v_{i+1}$ in J are in S, then all copies of $v_i v_{i+1}$ in H are in S (as $\mu_H(v_i v_{i+1}) = \mu_J(v_i v_{i+1})$), and $v_1 \dots v_l$ is not a path in $H \setminus S$, a contradiction. Therefore $v_1 \dots v_l$ is a path in $J \setminus S$, and so S is not a small t-cut in J, a contradiction. Therefore every small t-cut in J contains a copy of an edge not in F. If J has a small t-cut, then as every small t-cut in J is also a small t-cut in G, it follows that there is a small t-cut in G containing edges not in F. This is a contradiction by definition of F. Therefore we may conclude that J does not have a small t-cut, and so J has a t-road, as required. □

If $|F|$ is bounded by a function on p then, using Lemma 5 we can solve BCPP in FPT time by guessing the multiplicities of each edge in F for an optimal solution D. Unfortunately, $|F|$ may be larger than any function of p in general. It is also possible to solve the problem on graphs of bounded treewidth using dynamic programming techniques, but in general the treewidth may be unbounded. In Section 5 we give a tree decomposition of G in which the number of edges from F in each bag is bounded by a function of p. This allows us to combine both techniques. In Section 6 we give a dynamic programming algorithm utilizing Lemma 5 that runs in FPT time.

5 Tree Decomposition

In this section, we provide a tree decomposition of G which we will use for our dynamic programming algorithm. The tree decomposition does not have bounded treewidth (i.e. the bags do not have bounded size), but the intersection between bags is small, and each bag has a bounded number of vertices from small t-cuts. This will turn out to be enough to develop a fixed-parameter algorithm, as in some sense the hardness of BCPP comes from the small t-cuts.

Our tree decomposition is based on a result by Marx, O'Sullivan and Razgon [11], in which they show that the minimal small separators of a graph "live in a part of the graph that has bounded treewidth" [11].

Definition 3. *Given an undirected graph $G = (V, E)$, a* tree decomposition *of G is a pair $(\mathcal{T}, \beta)$, where $\mathcal{T}$ is a tree and $\beta : V(\mathcal{T}) \to 2^V$ such that $\bigcup_{x \in V(\mathcal{T})} \beta(x) = V$, for each edge $uv \in E$, there exists a node $x \in V(\mathcal{T})$ such that $u, v \in \beta(x)$, and for each $v \in V$, the set $\beta^{-1}(v)$ of nodes form a connected subgraph in $\mathcal{T}$.*

The width *of $(\mathcal{T}, \beta)$ is $\max_{x \in V(\mathcal{T})}(|\beta(x)| - 1)$. The* treewidth *of G (denoted $tw(G)$) is the minimum width of all tree decompositions of G.*

Lemma 6. *[11, Lemma 2.11] Let a, b be vertices of a graph $G = (V, E)$ and let l be the minimum size of an (a, b)-separator. For some $e \geq 0$, let S be the union of all minimal (a, b)-separators of size at most $l + e$. Then there is an $f(l, e) \cdot (|E| + |V|)$ time algorithm that returns a set $S' \supseteq S$ disjoint from $\{a, b\}$ such that $tw(torso(G, S')) \leq g(l, e)$, for some functions f and g depending only on l and e.*

Definition 4. *Given an undirected graph $G = (V, E)$, a* nice tree decomposition *$(\mathcal{T}, \beta)$ is a tree decomposition such that $\mathcal{T}$ is a rooted tree, and each of the nodes $x \in V(\mathcal{T})$ falls under one of the following classes:*

- x **is a Leaf node:** *Then x has no children in $\mathcal{T}$;*
- x **is an Introduce node:** *Then x has a single child y in $\mathcal{T}$, and there exists a vertex $v \notin \beta(y)$ such that $\beta(x) = \beta(y) \cup \{v\}$;*
- x **is a Forget node:** *Then x has a single child y in $\mathcal{T}$, and there exists a vertex $v \in \beta(y)$ such that $\beta(x) = \beta(y) \setminus \{v\}$;*
- x **is a Join node:** *Then x has two children y and z, and $\beta(x) = \beta(y) = \beta(z)$.*

It is well-known that given a tree decomposition of a graph, it can be transformed into a nice tree decomposition of the same width in polynomial time.

Our tree decomposition will be similar but not identical to a nice tree decomposition. We are now ready to give our tree decomposition, which is the main result of this section. Lemma 7 is proved in the Appendix.

Lemma 7. *Let $(G = (V, E), w, t)$ be an instance of* BCPP*, let C be the non-empty set of vertices appearing in edges in small t-cuts. Then there is an $f(p) \cdot (n+m)^{O(1)}$ time algorithm that returns a set S' and a (binary) tree decomposition $(\mathcal{T}, \beta)$ of G such that:*

1. *$C \subseteq S'$;*
2. *For any nodes $x \neq y$ in $\mathcal{T}$, $\beta(x) \cap \beta(y) \subseteq S'$;*
3. *For any node x in $\mathcal{T}$, $|\beta(x) \cap S'| \leq g(p)$*

for some functions f and g depending only on p. In addition, $\mathcal{T}$ is a rooted tree, and each of the nodes $\mathcal{T}$ falls under one of the following classes:

1. x **is a Leaf node:** *Then x has no children;*
2. x **is an Introduce node:** *Then x has a single child y in $\mathcal{T}$, $\beta(x) \subseteq S'$, and there exists a vertex $v \notin \beta(y)$ such that $\beta(x) = (\beta(y) \cap S') \cup \{v\}$;*
3. x **is a Forget node:** *Then x has a single child y in $\mathcal{T}$, $\beta(x) \subseteq S'$, and there exists a vertex $v \in \beta(y) \cap S'$ such that $\beta(x) = (\beta(y) \cap S') \setminus \{v\}$;*
4. x **is a Join node:** *Then x has two children y and z, $\beta(x) \subseteq S'$, and $\beta(x) = \beta(y) \cap S' = \beta(z) \cap S'$.*

Note that in our tree decomposition, the only nodes of $\mathcal{T}$ with bags not contained in S' are the Leaf nodes.

6 Dynamic Programming

Let (G, w, t) be an instance of BCPP. Let $(\mathcal{T}, \beta)$ be the tree decomposition of G and S' the set of vertices containing all vertices of every small t-cut given by Lemma 7. In this section we give a dynamic programming algorithm based on this decomposition.

Unlike the usual use of dynamic programming tree decompositions, we will not construct the restrictions of potential solutions to each bag when that bag is processed. Instead, we will produce undirected multigraphs corresponding to the undirected versions of potential solutions. When we have the optimal undirected multigraph for the whole problem, we then find its orientation as a last step. We do this because trying to find an optimal orientation at each bag $\beta(x)$ would involve making the bag t'-balanced for an arbitrary function $t' : \beta(x) \to [-p, p]$. On the other hand, to find the undirected version of a solution it is enough to guess the parity of the degree of each vertex within each bag.

To this end we introduce a function $h : V(G) \to \{\text{ODD}, \text{EVEN}\}$, where $h(v) = \text{ODD}$ if $t(v)$ is odd and $h(v) = \text{EVEN}$ if $t(v)$ is even. Observe that in the undirected version of any solution to BCPP on (G, w, t), each vertex v will have odd degree if and only if $t(v)$ is odd. Thus, h and similar functions will be used to tell us whether a vertex should have odd or even degree.

To simplify some expressions, we adopt the convention that $\text{ODD} + \text{ODD} = \text{EVEN}, \text{EVEN} + \text{EVEN} = \text{EVEN}$, and $\text{ODD} + \text{EVEN} = \text{ODD}$. We say a vertex v is *h-balanced* if it has odd degree if and only if $h(v) = \text{ODD}$. An undirected multigraph H is *h-balanced* if every vertex is h-balanced.

Let $\alpha(x) = \beta(x) \cap S'$. Thus $\beta(x) \cap \beta(y) \subseteq \alpha(x)$ for all nodes $x \neq y$, and $\alpha(x) = \beta(x)$ for every non-leaf x. Furthermore, for any Join node x with two children y and z, we have that $\alpha(x) = \alpha(y) = \alpha(z)$, even if one or both of the children of x is a Leaf node whose bag contains vertices not in S'.

Let $\gamma(x)$ be the union of the bags of all predecessors of x including x itself. Thus, if r is the root node of $\mathcal{T}$, then $\gamma(r) = V(G)$.

We now define the set of graphs constructed in our dynamic programming algorithm. Let x be a node of $\mathcal{T}$, let H' be an undirected multigraph with underlying graph $G[\alpha(x)]$, such that $\mu_{H'}(uv) \leq \max\{p, 2\}$ for all edges uv. Let T' be a directed graph with vertex set $\alpha(x)$, such that $\mu_{T'}(\overrightarrow{uv}) + \mu_{T'}(\overrightarrow{vu}) \leq \mu_{H'}(uv)$ for all edges uv. Let t' be a function $\alpha(x) \to [-p, p]$ and let h' be a function $\alpha(x) \to \{\text{ODD}, \text{EVEN}\}$. Then let $\psi(x, H', T', t', h')$ be an undirected multigraph H with underlying graph $G[\gamma(x)]$, of minimum weight such that

1. $H[\alpha(x)] = H'$.
2. H has a well-behaved t^*-road T such that T restricted to $\alpha(x)$ is T', where $t^* : \gamma(x) \to [-p, p]$ is the function such that $t^*(v) = t'(v)$ for $v \in \alpha(x)$ and $t^*(v) = t(v)$, otherwise.
3. H is h^*-balanced, where $h^* : \gamma(x) \to \{\text{ODD}, \text{EVEN}\}$ is the function such that $h^*(v) = h'(v)$ if $v \in \alpha(x)$ and $h^*(v) = h(v)$, otherwise.

Lemma 8. *Let r be the root node of $\mathcal{T}$. Let t' be t restricted to $\alpha(r)$, and let h' be h restricted to $\alpha(r)$. Let H' and T' be chosen such that the weight of $H = \psi(r, H', T', t', h')$ is minimized. Then the weight of H is the weight of an optimal solution to* BCPP *on (G, w, t), and given H we may construct an optimal solution to* BCPP *on (G, w, t) in polynomial time.*

Proof. Observe that by construction of t' and h', t^* and h^* in the definition of $\psi(r, H', T', t', h')$ are t and h, respectively. Let $H = \psi(r, H', T', t', h')$ for some choice of H' and T'. Then by definition H has a well-behaved t-road T and H is h-balanced. For each arc $\overrightarrow{uv}$ in T, orient a copy of the edge uv in H from u to v. Let D' be the resulting mixed multigraph. Then for every vertex v in D', we have $d^+_{D'}(v) - d^-_{D'}(v) = t(v)$. By definition of h and the fact that H was h-balanced, every v has an even number of edges incident to it.

Thus, the undirected edges can be partitioned into a set of cycles. By orienting each of these cycles to make a directed cycle, we get a directed multigraph D which is a solution to BCPP on (G, w, t). This shows that for every choice of H' and T', the graph $\psi(r, H', T', t', h')$ can be oriented to produce a solution to BCPP on (G, w, t). We will now show that an optimal solution D to BCPP on (G, w, t) is an orientation of $H = \psi(r, H', T', t', h')$ for some choice of H', T'.

Let H' be the undirected version of D restricted to $\alpha(r)$. Given a t-road T in D, let T' be T restricted to $\alpha(r)$. By Lemma 5, we may assume that T is well-behaved. Observe that H satisfies the conditions of $\psi(r, H', T', t', h')$. $\square$

Given an undirected graph F and a set X of vertices of F of even size, a set J of edges of F is an *X-Join* if $d_{F[J]}(v)$ is odd if and only if $v \in X$. When F has weights on its edges, we can speak of the Minimum Weight X-Join Problem; this problem can be solved in time $O(|V(F)|^3)$ [7]. (Traditionally, the Minimum Weight X-Join Problem is called the Minimum Weight T-Join Problem, but we use T for t-roads.)

Lemma 9. *$\psi(x, H', T', t', h')$ can be calculated in FPT time, for all choices of x, H', T', t', h'.*

Proof. Consider some node x, and assume that we have already calculated $\psi(y, H'', T'', t'', h'')$, for all descendants y of x and all choices of H'', T'', t'', h''. We consider the possible types of nodes separately.

x **is a Leaf node:** If $\beta(x) \subseteq S'$, then the only possible graph is H'. So return H' if H' is a solution, and return NULL, otherwise.

If $\beta(x) \setminus S' \neq \emptyset$, proceed as follows. Let $G_x = (\beta(x), E(G[\beta(x)]) \setminus E(G[\alpha(x)]))$. For each $v \in \beta(x)$, let $t''(v) = t'(v) - \sum_u \mu_{T'}(\overrightarrow{vu}) + \sum_u \mu_{T'}(\overrightarrow{uv})$. Then for any t'-road T^* that agrees with T' on $\alpha(x)$, T^* is the union of T' and a t''-road T'' on G_x. Furthermore, if T^* is well-behaved then $\mu_{T''}(\overrightarrow{uv}) + \mu_{T''}(\overrightarrow{vu}) \leq 1$ for any u, v. Thus, if $\psi(x, H', T', t', h') \neq$ NULL, then G_x has a t''-road. So we may proceed as follows. Check if G_x has a t''-road. If it does not, then return NULL. Otherwise, let $h^{**} : \beta(x) \to \{\text{ODD}, \text{EVEN}\}$ be such that if $v \in \alpha(x)$ has odd degree in H', then $h^{**}(v) = h^*(v) + \text{ODD}$, and otherwise $h^{**}(v) = h^*(v)$. Observe that the restriction of $\psi(x, H', T', t', h')$ to G_x will be h^{**}-balanced. Then to find $\psi(x, H', T', t', h')$, it suffices to find a minimum weight (multi)set of edges to add to G_x to make it h^{**}-balanced. This can be done by solving the Minimum Weight X-Join Problem, where X is the set of all vertices in $\beta(x)$ that are not h^{**}-balanced in G_x.

x **is an Introduce node:** Let y be the child node of x, and let v be the single vertex in $\beta(x) \setminus \alpha(y)$. Then no vertices in $\gamma(x)$ are adjacent with v, except for those in $\alpha(x)$. Therefore if v is not h'-balanced in H' or is not t'-balanced in T', we may return NULL. Otherwise, let H'' be H' restricted to $\alpha(y)$. Let T'' be T' restricted to $\alpha(y)$. Let $t'' : \alpha(y) \to [-p, p]$ be such that $t''(u) = t'(u) - \mu_{T'}(\overrightarrow{uv}) + \mu_{T'}(\overrightarrow{vu})$. Let $h'' : \alpha(y) \to \{\text{ODD}, \text{EVEN}\}$ be such that if $\mu_{H'}(uv)$ is odd then $h''(u) = h'(u) + \text{ODD}$, and otherwise $h''(u) = h'(u)$. Then $\psi(x, H', T', t', h')$ is $\psi(y, H'', T'', t'', h'')$ together with the edges of H' incident with v.

x **is a Forget node:** Let y be the child node of x, and let v be the single vertex in $\alpha(y) \setminus \beta(x)$. Let $t'' : \alpha(y) \to [-p, p]$ be the function that extends t' and assigns v to $t(v)$. Let $h'' : \alpha(y) \to \{\text{ODD}, \text{EVEN}\}$ be the function that extends h' and assings v to $h(v)$. Then $\psi(x, H', T', t', h')$ is $\psi(y, H'', T'', t'', h'')$, for some choice of H'' and T'' minimizing the weight of $\psi(y, H'', T'', t'', h'')$ such that H'' restricted to $\alpha(x)$ is H', and T'' restricted to $\alpha(x)$ is T'.

The proof of the case when x **is a Join node** and the analysis of algorithm's running time are in the Appendix. □

Lemmas 8 and 9 imply the following:

Theorem 1. BCPP *is fixed-parameter tractable.*

Theorem 1 and Lemma 2 imply the following:

Theorem 2. k-ARC CPP *is fixed-parameter tractable.*

7 Conclusions and Open Problems

We have solved an open problem in [2] by showing that MCPP parameterized by the number of arcs is fixed-parameter tractable. To prove this result we reduced MCPP to a generalization of UCPP and applied a very useful lemma of Marx *et al.* [11] on treewidth of the torso graph with respect to small separators. Note that our application of the lemma is significantly different from those in [11] and we believe that our approach will be of interest in designing fixed-parameter algorithms for other problems.

Van Bevern *et al.* [2] mention two other parameterizations of MCPP. One of them is by $\mathrm{tw}(G)$. It was proved by Fernandes *et al.* [9] that this parameterisation of MCPP is in XP, but it is unknown whether it is FPT [2]. A vertex v of G is called even if the number of arcs and edges incident to v is even. Edmonds and Johnson [7] proved that if all vertices of G are even then MCPP is polynomial time solvable. So, the number of odd (not even) vertices is a natural parameter. It is unknown whether the corresponding parameterization of MCPP is FPT [2].

Acknowledgement. Research of GG was supported by Royal Society Wolfson Research Merit Award.

References

1. Bang-Jensen, J., Gutin, G.: Digraphs: Theory, Algorithms and Applications, 2nd edn. Springer (2009)
2. van Bevern, R., Niedermeier, R., Sorge, M., Weller, M.: Complexity of Arc Rooting Problems. In: Corberán, A., Laporte, G. (eds.) Arc Routing: Problems, Methods and Applications, ch. 2. SIAM, Phil
3. Beltrami, E.J., Bodin, L.D.: Networks and vehicle routing for municipal waste collection. Networks 4(1), 65–94 (1974)
4. Brucker, P.: The Chinese postman problem for mixed graphs. In: Noltemeier, H. (ed.) Graphtheoretic Concepts in Computer Science. LNCS, vol. 100, pp. 354–366. Springer, Heidelberg (1981)
5. Christofides, N.: The optimum traversal of a graph. Omega 1, 719–732 (1973)
6. Downey, R.G., Fellows, M.R.: Fundamentals of Parameterized Complexity. Springer (2013)
7. Edmonds, J., Johnson, E.L.: Matching, Euler tours and the Chinese postman. Mathematical Programming 5, 88–124 (1973)
8. Eiselt, H.A., Gendreau, M., Laporte, G.: Arc routing problems. I. The Chinese postman problem. Oper. Res. 43, 231–242 (1995)
9. Fernandes, C.G., Lee, O., Wakabayashi, Y.: Minimum cycle cover and Chinese postman problems on mixed graphs with bounded tree-width. Discrete Applied Mathematics 157(2), 272–279 (2009)
10. Gutin, G., Jones, M., Sheng, B.: Parameterized Complexity of the k-Arc Chinese Postman Problem, `http://arxiv.org/abs/1403.1512`
11. Marx, D., O'Sullivan, B., Razgon, I.: Finding small separators in linear time via treewidth reduction. ACM Trans. Algorithms 9, article 30 (2013)
12. Minieka, E.: The Chinese postman problem for mixed networks. Management Sci. 25, 643–648 (1979/80)
13. Papadimitriou, C.H.: On the complexity of edge traversing. J. ACM 23, 544–554 (1976)
14. Peng, Y.: Approximation algorithms for some postman problems over mixed graphs. Chinese J. Oper. Res. 8, 76–80 (1989)
15. Sorge, M.: Some Algorithmic Challenges in Arc Routing. In: Talk at NII Shonan Seminar, no. 18 (May 2013)
16. Zaragoza Martínez, F.J.: Postman Problems on Mixed Graphs. PhD thesis, University of Waterloo (2003)

Approximating the Maximum Overlap of Polygons under Translation

Sariel Har-Peled and Subhro Roy

University of Illinois, Urbana-Champaign

Abstract. Let P and Q be two simple polygons in the plane of total complexity n, each of which can be decomposed into at most k convex parts. We present an $(1-\varepsilon)$-approximation algorithm, for finding the translation of Q, which maximizes its area of overlap with P. Our algorithm runs in $O(cn)$ time, where c is a constant that depends only on k and ε.

This suggest that for polygons that are "close" to being convex, the problem can be solved (approximately), in near linear time.

1 Introduction

Shape matching is an important problem in databases, robotics, visualization and many other fields. Given two shapes, we want to find how similar (or dissimilar) they are. Typical problems include matching point sets by the Hausdorff distance metric, or matching polygons by the Hausdorff or Fréchet distance between their boundaries. See the survey by Alt and Guibas [5].

The maximum area of overlap is one possible measure for shape matching that is not significantly effected by noise. Mount *et al.* [18] studied the behavior of the area of overlap function, when one simple polygon is translated over another simple polygon. They showed that the function is continuous and piece-wise polynomial of degree at most two. If the polygons P and Q have complexity m and n, respectively, the area of overlap function can have complexity of $\Theta(m^2n^2)$. Known algorithms to find the maximum of the function work by constructing the entire overlap function. It is also known that the problem is 3SUM-Hard [8], that is, it is believed no subquadratic time algorithm is possible for the problem.

Approximating maximum overlap of general polygons. Cheong *et al.* [13] gave a $(1-\varepsilon)$-approximation algorithm for maximizing the area of overlap under translation of one simple polygon over the other using random sampling techniques. However, the error associated with the algorithm is additive, and the algorithm runs in near quadratic time. Specifically, the error is an ε fraction of the area of the smaller of the two polygons. Under rigid motions, the running time deteriorates to being near cubic. More recently, Cheng and Lam [12] improved the running times, and can also handle rigid motions, and present a near linear time approximation algorithm if one of the polygons is convex.

Maximum overlap in the convex case under translations. de Berg *et al.* [10] showed that finding maximum overlap translation is relatively easier in case of convex polygons. Specifically, the overlap function in this case is unimodal (as a consequence of the Brunn-Minkowski Theorem). Using this property, they gave a near linear time exact algorithm for computing the translation that maximizes the area of overlap of two convex polygons. The complexity of the graph of the overlap function is only $O\big(m^2 + n^2 + \min(m^2 n, mn^2)\big)$ in this case. Alt *et al.* [4] gave a constant-factor approximation for the minimum area of the symmetric difference of two convex polygons.

Approximating maximum overlap in the convex case. As for $(1-\varepsilon)$-approximation, assuming that the two polygons are provided in an appropriate form (i.e., the vertices are in an array in their order along the boundary of the polygon), then one can get a sub-linear time approximation algorithm. Specifically, Ahn *et al.* [3] show an $(1-\varepsilon)$-approximation algorithm, with running time $O((1/\varepsilon)\log(n/\varepsilon))$ for the case of translation, and $O((1/\varepsilon)\log n + (1/\varepsilon)^2 \log 1/\varepsilon))$ for the case of rigid motions. (For a result using similar ideas in higher dimensions see the work by Chazelle *et al.* [11].)

Overlap of union of balls. de Berg *et al.* [9] considered the case where X and Y are disjoint unions of m and n unit disks, with $m \leq n$. They computed a $(1-\varepsilon)$ approximation for the maximal area of overlap of X and Y under translations in time $O((nm/\varepsilon^2)\log(n/\varepsilon))$. Cheong *et al.* [13] gave an additive error ε-approximation algorithm for this case, with near linear running time.

Other relevant results. Avis *et al.* [6] computes the overlap of a polytope and a translated hyperplane in linear time, if the polytope is represented by a lattice of its faces. Vigneron [21] presented $(1-\varepsilon)$-approximation algorithms for maximum overlap of polyhedra (in constant dimension) that runs in polynomial time. Ahn *et al.* [1] approximates the maximum overlap of two convex polytopes in three dimensions under rigid motions. Ahn *et al.* [2] approximates the maximum overlap of two polytopes in $\mathbb{R}^d$ under translation in $O\Big(n^{\lfloor d/2 \rfloor + 1} \log^d n\Big)$ time.

Our Results

As the above indicates, there is a big gap between the algorithms known for the convex and non-convex case. Our work aims to bridge this gap, showing that for "close" to convex polygons, under translation, the problem can be solved approximately in near linear time.

Specifically, assume we are given two polygons P and Q of total complexity n, such that they can be decomposed into k convex parts, we show that one can $(1-\varepsilon)$-approximate the translation of Q, which maximizes its area of overlap with P, in linear time (for k and ε constants). The translation returned has overlap area which is at least $(1-\varepsilon)\mu_{\max}(\mathsf{P}, \mathsf{Q})$, where $\mu_{\max}(\mathsf{P}, \mathsf{Q})$ is the maximum area of overlap of the given polygons.

Approach. We break the two polygons into a minimum number of convex parts. We then approximate the overlap function for each pair of pieces (everywhere). This is required as one cannot just approximate the two polygons (as done by Ahn *et al.* [3]) since the optimal solution does not realize the maximum overlap of each pair of parts separately, and the alignment of each pair of parts might be arbitrary.

To this end, if the two convex parts are of completely different sizes, we approximate the smaller part, and approximate the overlap function by taking slices (i.e., level sets) of the overlap function. In the other case, where the two parts are "large", which is intuitively easier, we can approximate both convex parts, and then the overlap function has constant complexity. Finally, we overlap all these functions together, argue that the overlap has low complexity, and find the maximum area of overlap.

Our approach has some overlap in ideas with the work of Ahn *et al.* [3]. In particular, a similar distinction between large and small overlap, as done in Section 4.1 and Section 4.2 was already done in [3, Theorem 17].

Why the "naive" solution fails? The naive solution to our problem is to break the two polygons into k convex polygons, and then apply to each pair of them the approximation of Ahn *et al.* [3]. Now, just treat the input polygon as the union of their respective approximations, and solve problem using brute force approach. This fails miserably as the approximation of Ahn *et al.* [3] captures only the maximum overlap of the two polygons. It does not, and can not, approximates the overlap if two convex polygons are translated such that their overlap is "far" from the maximum configuration, especially if the two polygons are of different sizes. This issue is demonstrated in more detail in the beginning of Section 4.1. A more detailed counterexample is presented in the full version of the paper [16].

Paper organization. We start in Section 2 by defining formally the problem, and review some needed results. In Section 3, we build some necessary tools. Specifically, we start in Section 3.1 by observing that one can get $O(1/\varepsilon)$ approximation of a convex polygon, where the error is an ε-fraction of the width of the polygon. In Section 3.2, we show how to compute a level set of the overlap function of two convex polygons efficiently. In Section 3.3, we show that, surprisingly, the polygon formed by the maximum overlap of two convex polygons, contains (up to scaling by a small constant and translation) the intersection of any translation of these two convex polygons. Among other things this implies an easy linear time constant factor approximation for the maximum overlap (which also follows, of course, by the result of Ahn *et al.* [3]). In Section 4, we present the technical main contribution of this paper, showing how to approximate, by a compact representation that has roughly linear complexity, the area overlap function of two convex polygons. In Section 5 we put everything together and present our approximation algorithm for the non-convex case.

2 Preliminaries

For any vector $\mathsf{t} \in \mathbb{R}^2$ and a set Q, let $\mathsf{t} + \mathsf{Q}$ denote the translation of Q by t; formally, $\mathsf{t} + \mathsf{Q} = \left\{ \mathsf{t} + \mathsf{q} \;\middle|\; \mathsf{q} \in \mathsf{Q} \right\}$. Also let $\mu(\mathsf{P}, \mathsf{Q}) = \mathrm{area}(\mathsf{P} \cap \mathsf{Q})$, which is the ***area of overlap*** of sets P and Q. We are interested in the following problem.

Problem 1. We are given two polygons X and Y in the plane, such that each can be decomposed into at most k convex polygons. The task is to compute the translation t of Y, which maximizes the area of overlap between X and $\mathsf{t} + \mathsf{Y}$. Specifically our purpose is to approximate the quantity

$$\mu_{\max}(\mathsf{X}, \mathsf{Y}) = \max_{\mathsf{t} \in \mathbb{R}^2} \mu(\mathsf{X}, \mathsf{t} + \mathsf{Y}) \,.$$

For a polygon P, let $|\mathsf{P}|$ denote the number of vertices of P. For $\mathsf{X}, \mathsf{Y} \subseteq \mathbb{R}^d$, the set X is ***contained under translation*** in Y, denoted by $\mathsf{X} \sqsubseteq \mathsf{Y}$, if there exists $\boldsymbol{x}$ such that $\boldsymbol{x} + \mathsf{X} \subseteq \mathsf{Y}$.

Unimodal. A function $f : \mathbb{R} \to \mathbb{R}$ is ***unimodal***, if there is a value α, such that f is monotonically increasing (formally, non-decreasing) in the range $[-\infty, \alpha]$, and f is monotonically decreasing (formally, non-increasing) in the interval $[\alpha, +\infty]$.

From width to inner radius. For a convex polygon P, the ***width*** of P, denoted by $\omega(\mathsf{P})$, is the minimum distance between two parallel lines that enclose P.

Lemma 1 ([14]). *For a convex shape* X *in the plane, we have that the largest disk enclosed inside* X*, has radius at least* $\mathrm{width}(\mathsf{X}) / 2\sqrt{3}$.

Convex Decomposition of Simple Polygons. A vertex of a polygon is a ***notch*** if the internal angle at this vertex is reflex (i.e. $> 180°$). For a non-convex polygon P with n vertices and r notches, Keil and Snoeyink [17] solves the minimal convex decomposition problem in $O\big(n + r^2 \min(r^2, n)\big)$ time, that is, they compute a decomposition of P into minimum number of convex polygons. Observe, that if the number of components in the minimum convex decomposition is k, the number of notches r is upper bounded by $2k$.

Scaling similarity between polygons. For two convex polygons X and Y, let us define their ***scaling similarity***, denoted by $\mathrm{ssim}(\mathsf{X}, \mathsf{Y})$, as the minimum number $\alpha \geq 0$, such that $\mathsf{X} \sqsubseteq \alpha\mathsf{Y}$. Using low-dimensional linear programming, one can compute $\mathrm{ssim}(\mathsf{X}, \mathsf{Y})$ in linear time. In particular, the work by Sharir and Toledo [19] implies the following.

Lemma 2 (ssim). *Given two convex polygons* X *and* Y *of total complexity* n*, one can compute, in linear time,* $\mathrm{ssim}(\mathsf{X}, \mathsf{Y})$*, and the translation that realizes it.*

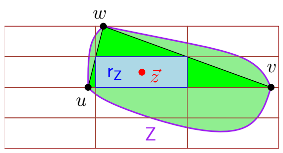
3 Building Blocks

3.1 A Better Convex Approximation in the Plane

Let B be the minimum volume bounding box of some bounded convex set $K \subseteq \mathbb{R}^d$. We have that $v + c_d B \subseteq K \subseteq B$ [15], for some vector v and a constant c_d which depends only on the dimension d. This approximation can be computed in $O(n)$ time [7], where n is the number of vertices of the convex-hull of K. The more powerful result showing that a convex body can be approximated by an ellipsoid (up to a scaling factor of d), is known as John's Theorem [15].

We need the following variant of the algorithm of Barequet and Har-Peled [7].

Lemma 3. *Given a convex polygon Z in the plane, with n vertices, one can compute, in linear time, a rectangle $\mathsf{r_Z}$ and a point $\boldsymbol{z}$, such that $\boldsymbol{z} + \mathsf{r_Z} \subseteq \mathsf{Z} \subseteq \boldsymbol{z} + 5\mathsf{r_Z}$.*



Proof. This is all well known, and we include the details for the sake of completeness. Using rotating caliper [20] compute the two vertices u and v of Z realizing its diameter. Let w be the vertex of Z furthest away from uv, Consider the rectangle r'_Z having its base on uv, having half the height of $\triangle uvw$, and contained inside this triangle. Now, let $\boldsymbol{z}$ be the center of r'_Z, and set $\mathsf{r_Z} = \mathsf{r}'_\mathsf{Z} - \boldsymbol{z}$, see figure on the right. It is now easy to verify that the claim holds with $\mathsf{r_Z}$ and $\boldsymbol{z}$. □

Observation 1. *Given two bodies $\mathsf{X}, \mathsf{Y} \subseteq \mathbb{R}^2$ and a non-singular affine transformation M, we have* $\dfrac{\mathrm{area}(\mathsf{X})}{\mathrm{area}(\mathsf{Y})} = \dfrac{\mathrm{area}(\mathsf{M}(\mathsf{X}))}{\mathrm{area}(\mathsf{M}(\mathsf{Y}))}$.

Since a similar construction is described by Ahn *et al.* [3], we delegate the proof of this lemma to the full version of this paper [16].

Lemma 4 (approxPolygon). *Given a convex polygon P, and a parameter $m > 0$, we can compute, in $O(|\mathsf{P}|)$ time, a convex polygon P' with $O(m)$ vertices, such that (i) $\mathsf{P}' \subseteq \mathsf{P}$, and (ii) for any point $\mathsf{p} \in \mathsf{P}$, its distance from P' is at most $\omega(\mathsf{P})/m$, where $\omega(\mathsf{P})$ is the width of P.*

3.2 The Level Set of the Area of Overlap Function

Definition 1. *The superlevel set of a function $f : \mathbb{R}^d \to \mathbb{R}$, for a value α is the set $L_\alpha(f) = \left\{ \mathsf{p} \in \mathbb{R}^d \;\middle|\; f(\mathsf{p}) \geq \alpha \right\}$. We will refer to it as the **α-slice** of f.*

Lemma 5. *Given two convex polygons X and Y, the slice $\mathsf{Z} = L_\alpha(\mu(\mathsf{X}, \mathsf{t} + \mathsf{Y}))$ is convex, and has complexity $O(m)$, where $m = |\mathsf{X}|\,|\mathsf{Y}|$. Furthermore, given a point $\mathsf{p} \in \mathsf{Z}$, the convex body Z can be computed in $O(m \log m)$ time.*

The proof is in the full version of the paper [16].

3.3 The Shape of the Polygon Realizing the Maximum Area Overlap

In the following, all the ellipses being considered are centered in the origin.

Lemma 6. *Given two ellipses $\mathcal{E}_1$ and $\mathcal{E}_2$, the translation which maximizes their area of overlap is the one in which their centers are the same points.*

Proof. Translate $\mathcal{E}_1$ and $\mathcal{E}_2$ such that their centers are at the origin. Consider any unit vector $\boldsymbol{u}$, translate $\mathcal{E}_2$ along the direction of $\boldsymbol{u}$, and consider the behavior of the overlap function $f(x) = \mu\Big(\mathcal{E}_1, \mathcal{E}_2 + x\boldsymbol{u}\Big)$, where x varies from $-\infty$ to $+\infty$. The function f is unimodal [10]. By symmetry, we have

$$f(x) = \mu\Big(\mathcal{E}_1, \mathcal{E}_2 + x\boldsymbol{u}\Big) = \mu\Big(-\mathcal{E}_1, -(\mathcal{E}_2 + x\boldsymbol{u})\Big) = \mu\Big(\mathcal{E}_1, \mathcal{E}_2 - x\boldsymbol{u}\Big) = f(-x),$$

as $\mathcal{E}_i = -\mathcal{E}_i$. If the maximum is attained at $x \neq 0$, we will get another maximum at $-x$, which implies, as f unimodal, that $f(0) = f(x) = f(-x)$, as desired. □

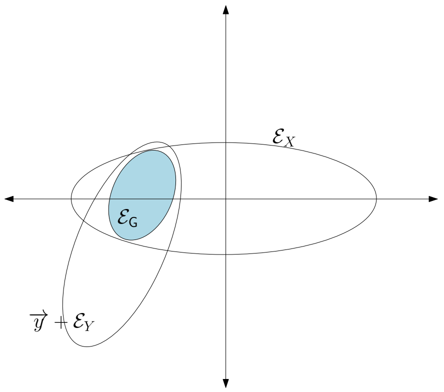


Fig. 3.1.

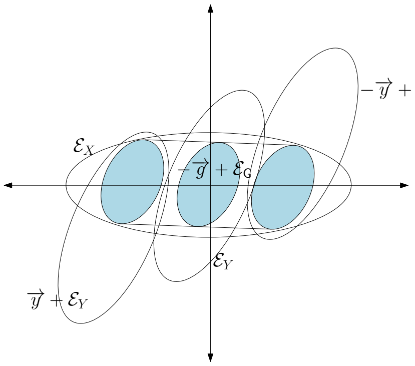

$-\overrightarrow{y} + \mathcal{E}_Y$
$\mathcal{E}_X$
$-\overrightarrow{g} + \mathcal{E}_\mathsf{G}$
$\mathcal{E}_Y$
$\overrightarrow{y} + \mathcal{E}_Y$

Fig. 3.2.

Lemma 7. *Consider two ellipses $\mathcal{E}_\mathsf{X}$ and $\mathcal{E}_\mathsf{Y}$ in the plane, and consider any two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$, then there is a vector $\boldsymbol{u}$ such that $\boldsymbol{u} + (\boldsymbol{x} + \mathcal{E}_\mathsf{X}) \cap (\boldsymbol{y} + \mathcal{E}_\mathsf{Y}) \subseteq 2\mathcal{E}_\mathsf{X} \cap 2\mathcal{E}_\mathsf{Y}$.*

Proof. For the sake of simplicity of exposition, assume that $\boldsymbol{x} = 0$. Now, consider the intersection $\mathsf{G} = \mathcal{E}_\mathsf{X} \cap (\boldsymbol{y} + \mathcal{E}_\mathsf{Y})$, and let $\mathcal{E}_\mathsf{G}$ be the largest area ellipse contained inside G. John's theorem implies that there is a translation vector $\boldsymbol{g}$, such that $\boldsymbol{g} + \mathcal{E}_\mathsf{G} \subseteq \mathsf{G} \subseteq \boldsymbol{g} + 2\mathcal{E}_\mathsf{G}$, see Figure 3.1.

Observe that $\boldsymbol{g} + \mathcal{E}_\mathsf{G} \subseteq \mathcal{E}_\mathsf{X}$, and by the symmetry of $\mathcal{E}_\mathsf{G}$ and $\mathcal{E}_\mathsf{X}$, we have that $-\boldsymbol{g} + \mathcal{E}_\mathsf{G} = -\boldsymbol{g} - \mathcal{E}_\mathsf{G} \subseteq -\mathcal{E}_\mathsf{X} = \mathcal{E}_\mathsf{X}$. This by convexity implies that $\mathcal{E}_\mathsf{G} \subseteq \mathcal{E}_\mathsf{X}$. A similar argument implies that $\mathcal{E}_\mathsf{G} \subseteq \mathcal{E}_\mathsf{Y}$. As such, $\mathcal{E}_\mathsf{G} \subseteq \mathcal{E}_\mathsf{X} \cap \mathcal{E}_\mathsf{Y}$.

Thus, we have that $\mathsf{G} \subseteq \boldsymbol{g} + 2\mathcal{E}_\mathsf{G} \subseteq \boldsymbol{g} + 2\mathcal{E}_\mathsf{X} \cap 2\mathcal{E}_\mathsf{Y}$, as desired. □

Lemma 8 ([15, Lemma 22.5]). *Any convex set $K \subseteq \mathbb{R}^d$ contained in a unit square, contains a ball of radius* $\text{area}(K)/8$

The following lemma is one of our key insights – the maximum area of intersection of two polygons contains any intersection of translated copies of these polygons up to translation and a constant factor scaling.

Lemma 9. *Let X and Y be two convex polygons, and let M be the polygon realizing their maximum area of intersection under translation. Let $\boldsymbol{u}$ be any vector in the plane, and consider the polygon $\mathsf{D} = \mathsf{X} \cap (\boldsymbol{u} + \mathsf{Y})$, then there exists a vector $\boldsymbol{v}$ such that, $\boldsymbol{v} + \mathsf{D} \subseteq c_0\mathsf{M}$, for some fixed constant c_0.*

Proof. Let $\mathcal{E}_\mathsf{X}$ (resp., $\mathcal{E}_\mathsf{Y}$) denote the maximum area ellipse (centered at the origin) contained inside X (resp. Y). By John's Theorem, we have $\boldsymbol{x} + \mathcal{E}_\mathsf{X} \subseteq \mathsf{X} \subseteq \boldsymbol{x} + 2\mathcal{E}_\mathsf{X}$ and $\boldsymbol{y} + \mathcal{E}_\mathsf{Y} \subseteq \mathsf{Y} \subseteq \boldsymbol{y} + 2\mathcal{E}_\mathsf{Y}$, where $\boldsymbol{x}, \boldsymbol{y}$ are some vector. Let $\mathsf{B} = \mathcal{E}_\mathsf{X} \cap \mathcal{E}_\mathsf{Y}$, and let $\mathcal{E}_\mathsf{B}$ be the maximum area ellipse contained inside B. Observe that B is symmetric and centered at the origin, and by John's theorem $\mathcal{E}_\mathsf{B} \subseteq \mathsf{B} \subseteq 2\mathcal{E}_\mathsf{B}$.

By Lemma 7, there are vectors $\boldsymbol{z}$ and $\overrightarrow{w}$, such that

$$\begin{aligned}\mathsf{D} = \mathsf{X} \cap (\boldsymbol{u} + \mathsf{Y}) &\subseteq (\boldsymbol{x} + 2\mathcal{E}_\mathsf{X}) \cap (\boldsymbol{z} + \boldsymbol{y} + 2\mathcal{E}_\mathsf{Y}) \subseteq \overrightarrow{w} + 4\mathcal{E}_\mathsf{X} \cap 4\mathcal{E}_\mathsf{Y} = \overrightarrow{w} + 4\mathsf{B} \\ &\subseteq \overrightarrow{w} + 8\mathcal{E}_\mathsf{B}.\end{aligned}$$

Applying a similar argument, we have that $\mathsf{M} \subseteq \overrightarrow{m} + 8\mathcal{E}_\mathsf{B}$, for some vector $\overrightarrow{m}$.

Apply the linear transformation that maps $\mathcal{E}_\mathsf{B}$ to disk(1/16), where disk(r) denotes the disk of radius r centered at the origin. By Observation 1, we can continue our discussion in the transformed coordinates. This implies that $\mathsf{M} - \overrightarrow{m} \subseteq \text{disk}(1/2)$ (which is contained inside a unit square). By Lemma 8, there is a vector $\boldsymbol{x_1}$, such that $\boldsymbol{x_1} + \text{disk}(\text{area}(\mathsf{M})/8) \subseteq \mathsf{M}$.

Observe that $\mathsf{B} = \mathcal{E}_\mathsf{X} \cap \mathcal{E}_\mathsf{Y} \subseteq (-\boldsymbol{x} + \mathsf{X}) \cap (-\boldsymbol{y} + \mathsf{Y})$. As such, the area of B must be smaller than the area of M (by the definition of M). We thus have $\text{area}(\mathsf{M}) \geq \text{area}(\mathsf{B}) \geq \text{area}(\mathcal{E}_\mathsf{B}) = \text{area}(\text{disk}(1/16))$ which is a constant bounded away from zero. Therefore,

$$\begin{aligned}\mathsf{D} &\subseteq \overrightarrow{w} + 8\mathcal{E}_\mathsf{B} = \overrightarrow{w} + \text{disk}\left(\frac{1}{2}\right) = \overrightarrow{w} + \frac{4}{\text{area}(\mathsf{M})} \cdot \text{disk}\left(\frac{\text{area}(\mathsf{M})}{8}\right) \\ &\subseteq \overrightarrow{w} + \frac{4}{\text{area}(\mathsf{M})}(\mathsf{M} - \boldsymbol{x_1}),\end{aligned}$$

which implies the claim. ☐

Constant Approximation to the Maximum Overlap

Lemma 10 (constApproxByRect). *Let X and Y be two convex polygons, and let M be the polygon realizing their maximum area intersection under translation. Then, one can compute, in $O(|\mathsf{X}| + |\mathsf{Y}|)$ time, a rectangle r, such that $\mathsf{r} \subseteq \boldsymbol{u} + \mathsf{M} \subseteq c_r\mathsf{r}$, where c_r is a constant. That is, one can compute a constant factor approximation to the maximum area overlap in linear time.*

Furthermore, for any translation $\mathsf{t_Y}$, we have that $\mathsf{X} \cap (\mathsf{Y} + \mathsf{t_Y}) \sqsubseteq c_r\mathsf{r}$.

Proof. We are going to implement the algorithmic proof of Lemma 9. Instead of John's ellipsoid we use the rectangle of Lemma 3. Clearly, the proof of Lemma 9 goes through with the constants being somewhat worse. Specifically, we compute, in linear time, vectors $\boldsymbol{x}, \boldsymbol{y}$, and rectangles $\mathsf{r}_\mathsf{X}, \mathsf{r}_\mathsf{Y}$, such that $\boldsymbol{x} + \mathsf{r}_\mathsf{X} \subseteq \mathsf{X} \subseteq \boldsymbol{x} + 5\mathsf{r}_\mathsf{X}$ and $\boldsymbol{y} + \mathsf{r}_\mathsf{Y} \subseteq \mathsf{Y} \subseteq \boldsymbol{y} + 5\mathsf{r}_\mathsf{Y}$. Again, compute a rectangle r_M, such that $\mathsf{r}_\mathsf{M}/5 \subseteq \mathsf{r}_\mathsf{X} \cap \mathsf{r}_\mathsf{Y} \subseteq \mathsf{r}_\mathsf{M}$. Arguing as in Lemma 9, and setting $\mathsf{r} = \mathsf{r}_\mathsf{M}/c_3$, for some constant c_3, is the desired rectangle. □

4 Approximating the Overlap Function of Convex Polygons

Definition 2 *Given two convex polygons* X *and* Y *in the plane, of total complexity* n*, and parameters* $\varepsilon \in (0,1)$, ν, ρ, *a function* $\psi(\mathsf{t})$ *is* **(ε,ν,ρ)-approximation** *of* $\mu(\mathsf{X}, \mathsf{t}+\mathsf{Y})$, *if the following conditions hold:*

(A) $\forall \mathsf{t} \in \mathbb{R}^2$, *we have* $|\mu(\mathsf{X}, \mathsf{t}+\mathsf{Y}) - \psi(\mathsf{t})| \leq \varepsilon \mu_{\max}(\mathsf{X},\mathsf{Y})$.

(B) There are convex polygons $\mathsf{P}_1, \ldots, \mathsf{P}_\nu$, *each of maximum complexity* ρ, *such that inside every face of the arrangement* $\mathcal{A} = \mathcal{A}(\mathsf{P}_1, \ldots, \mathsf{P}_\nu)$, *the approximation function* $\psi(\mathsf{t})$ *is the same quadratic function.*

That is, the total descriptive complexity of $\psi(\cdot)$ *is the complexity of the arrangement* $\mathcal{A}$.

Algorithm 3 *The input is two convex polygons* X *and* Y *in the plane, of total complexity* n*, and a parameter* $\varepsilon \in (0,1)$*. As a first step, the algorithm is going to approximate* X *and* Y *as follows:*

(A) $\mathsf{r}_\mathsf{M} \leftarrow$ **constApproxByRect**(X,Y), *see Lemma 10.*

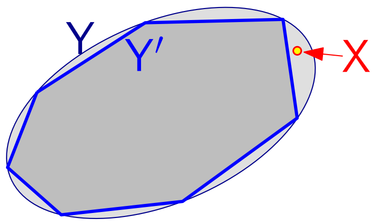
(B) $\mathcal{T} \leftarrow$ *affine transformation that maps* $2c_r\mathsf{r}_\mathsf{M}$ *to* $[0,1]^2$.

(C) $\mathsf{X}'_\mathcal{T} \leftarrow$ **approxPolygon**$(\mathcal{T}(\mathsf{X})\,,\, N)$ *and* $\mathsf{Y}'_\mathcal{T} \leftarrow$ **approxPolygon**$(\mathcal{T}(\mathsf{Y})\,,\, N)$.
See Lemma 4, here $N = \lceil c_4/\varepsilon \rceil$, *and* c_4 *is a sufficiently large constant.*

(D) $\mathsf{X}' \leftarrow \mathcal{T}^{-1}(\mathsf{X}'_\mathcal{T})$ *and* $\mathsf{Y}' \leftarrow \mathcal{T}^{-1}(\mathsf{Y}'_\mathcal{T})$.

4.1 If One Polygon is Smaller than the Other

Assume, without loss of generality, that X is smaller than Y, that is, X can be translated so that it is entirely contained inside Y (i.e., ssim$(\mathsf{X},\mathsf{Y}) \leq 1$, see Lemma 2). The maximum area of overlap is now equal to area(X). The challenge is, that for any approximation of Y, we can always have a sufficiently small X which can be placed in $\mathsf{Y} \setminus \mathsf{Y}'$, as shown in the figure on the right. Therefore for all those translations for which X is placed inside $\mathsf{Y} \setminus \mathsf{Y}'$, our approximation will show zero overlap, even though the actual overlap is area(X).



To get around this problem, we will first approximate the smaller polygon X, using our approximation scheme, to get polygon X', then we will compute level sets of the overlap function and use them to approximate it.

Lemma 11. *Given convex polygons* X *and* Y*, such that* $\mathrm{ssim}(\mathsf{X}, \mathsf{Y}) < 1$*, and parameter* $\varepsilon > 0$*, and let* X' *be the approximation to* X*, as computed by Algorithm 3. Then, we have, for all translations* $\mathsf{t} \in \mathbb{R}^2$*, that* $\left|\mu(\mathsf{X}', \mathsf{t}+\mathsf{Y}) - \mu(\mathsf{X}, \mathsf{t}+\mathsf{Y})\right| \leq \varepsilon\mu_{\max}(\mathsf{X}, \mathsf{Y})$ *.*

Proof. Consider the overlap of $\mathsf{X}_{\mathcal{T}} = \mathcal{T}(\mathsf{X})$ and $\mathsf{Y}_{\mathcal{T}} = \mathcal{T}(\mathsf{Y})$. Lemma 10 implies that any intersection polygon of $\mathsf{X}_{\mathcal{T}}$ and $\mathsf{Y}_{\mathcal{T}}$ can be contained (via translation) in $\mathcal{T}(c_r \mathsf{r}_{\mathsf{M}})$ (which is a translation of the square $[0, 1/2]^2$). Clearly, in this case, $\mathsf{X}_{\mathcal{T}}$ and $\mathsf{X}'_{\mathcal{T}}$ can both be translated to be contained in this square, both contain a disk of constant radius, the maximum distance between $\mathsf{X}_{\mathcal{T}}$ and $\mathsf{X}'_{\mathcal{T}}$ is $O(\varepsilon)$, and the total area of $\mathsf{X}_{\mathcal{T}} \setminus \mathsf{X}'_{\mathcal{T}}$ is $O(\varepsilon)$, as the perimeter of $\mathsf{X}_{\mathcal{T}} \leq 4$. Thus, setting c_4 to be sufficiently large, implies that $\mathrm{area}(\mathsf{X}_{\mathcal{T}} \setminus \mathsf{X}'_{\mathcal{T}}) \leq \varepsilon\mu_{\max}(\mathsf{X}_{\mathcal{T}}, \mathsf{Y}_{\mathcal{T}})$, as $\mu_{\max}(\mathsf{X}_{\mathcal{T}}, \mathsf{Y}_{\mathcal{T}}) = \Omega(1)$. This implies that $\left|\mu(\mathsf{X}'_{\mathcal{T}}, \mathsf{t}+\mathsf{Y}_{\mathcal{T}}) - \mu(\mathsf{X}_{\mathcal{T}}, \mathsf{t}+\mathsf{Y}_{\mathcal{T}})\right| \leq \varepsilon\mu_{\max}(\mathsf{X}_{\mathcal{T}}, \mathsf{Y}_{\mathcal{T}})$, which implies the claims by applying $\mathcal{T}^{-1}$ to both sides. □

Therefore, $\mu(\mathsf{X}', \mathsf{t}+\mathsf{Y})$ is a good approximation for $\mu(\mathsf{X}, \mathsf{t}+\mathsf{Y})$. However, $\mu(\mathsf{X}', \mathsf{t}+\mathsf{Y})$ has complexity $O\left(|\mathsf{X}'|^2 |\mathsf{Y}|^2\right)$ [10], in the worst case, which is still too high.

Lemma 12 (approxLevelSet). *Given two convex polygons* X *and* Y*, of total complexity* n*, and a parameter* ε*, such that* $\mathrm{ssim}(\mathsf{X}, \mathsf{Y}) < 1$*, then one can construct in* $O(n/\varepsilon^2)$ *time, a* $\left(\varepsilon, O(1/\varepsilon^2), O(n/\varepsilon^2)\right)$*-approximation* $\psi(\cdot)$ *to* $\mu(\mathsf{X}, \mathsf{t}+\mathsf{Y})$*.*

Proof. There is a translation of X such that it is contained completely in Y. Approximate X from the outside by a rectangle r, using Lemma 3. Next, spread a grid in r by partitioning each of its edges into $O(1/\varepsilon)$ equal length intervals. Let S be the set of points of the grid that are in X. It is easy to verify, that for any convex body Z and a translation t, we have

$$\left|\mu(\mathsf{X}, \mathsf{t}+\mathsf{Z}) - \frac{|(\mathsf{t}+\mathsf{Z}) \cap \mathsf{S}|}{|\mathsf{S}|}\right| \leq \varepsilon\,\mathrm{area}(\mathsf{X})$$

Namely, to approximate the overlap area for $\mathsf{t}+\mathsf{Y}$, we need to count the number of points of S that it covers. To this end, for each point $\mathsf{p} \in \mathsf{S}$, we generate a 180° rotated and translated copy of Y, denoted by Y'_{p}, such that $\mathsf{p} \in \mathsf{t}+\mathsf{Y}$ if and only if $\mathsf{t} \in \mathsf{Y}'_{\mathsf{p}}$.

Clearly, the generated set of polygons is the desired $\left(\varepsilon, O(1/\varepsilon^2), O(n/\varepsilon^2)\right)$-approximation $\psi(\cdot)$ to $\mu(\mathsf{X}, \mathsf{t}+\mathsf{Y})$.

The time to build this approximation is $O(n/\varepsilon^2)$.

We next describe a slightly slower algorithm that generates a slightly better approximation.

Lemma 13 (approxLevelSet). *Given two convex polygons* X *and* Y*, of total complexity* n*, and a parameter* ε*, such that* $\mathrm{ssim}(\mathsf{X}, \mathsf{Y}) < 1$*, then one can construct in* $O(\varepsilon^{-2} n \log n)$ *time, a* $(\varepsilon, O(1/\varepsilon), O(n/\varepsilon))$*-approximation* $\psi(\cdot)$ *to* $\mu(\mathsf{X}, \mathsf{t}+\mathsf{Y})$*.*

The proof is in the full version of the paper [16].

4.2 If the Two Polygons are Incomparable

The more interesting case, is when the maximum intersection of X and Y is significantly smaller than both polygons; that is, $\mathrm{ssim}(\mathsf{X},\mathsf{Y}) \geq 1$ and $\mathrm{ssim}(\mathsf{Y},\mathsf{X}) \geq 1$. Surprisingly, in this case, we can approximate both polygons simultaneously.

Lemma 14. *Given convex polygons* X *and* Y*, such that* $\mathrm{ssim}(\mathsf{X},\mathsf{Y}) \geq 1$ *and* $\mathrm{ssim}(\mathsf{Y},\mathsf{X}) \geq 1$*, then the widths of* $\mathsf{X}_{\mathcal{T}} = \mathcal{T}(\mathsf{X})$ *and* $\mathsf{Y}_{\mathcal{T}} = \mathcal{T}(\mathsf{Y})$*, as computed by Algorithm 3, are bounded by* 7*.*

The proof is in the full version of the paper [16].

Lemma 15. *Given two convex polygons* X *and* Y*, of total complexity* n*, and a parameter* ε*, such that* $\mathrm{ssim}(\mathsf{X},\mathsf{Y}) \geq 1$ *and* $\mathrm{ssim}(\mathsf{Y},\mathsf{X}) \geq 1$*, then one can construct in* $O\big(n + 1/\varepsilon^2\big)$ *time, a* $(\varepsilon, O(1/\varepsilon), O(1/\varepsilon))$*-approximation* $\psi(\cdot)$ *to* $\mu(\mathsf{X}, \mathsf{t} + \mathsf{Y})$*.*

The proof is in the full version of the paper [16].

The result. By combining Lemma 12 and Lemma 15 (deciding which one to apply can be done by computing $\mathrm{ssim}(\mathsf{X},\mathsf{Y})$ and $\mathrm{ssim}(\mathsf{Y},\mathsf{X})$, which takes $O(n)$ time), we get the following.

Lemma 16. *Given two convex polygons* X *and* Y*, of total complexity* n*, and a parameter* ε*, one can construct in* $O\big(n/\varepsilon^2\big)$ *time, a* $\big(\varepsilon, O(1/\varepsilon^2), O(n/\varepsilon^2)\big)$*-approximation* $\psi(\cdot)$ *to* $\mu(\mathsf{X}, \mathsf{t} + \mathsf{Y})$*.*

5 Approximating the Maximum Overlap of Polygons

The input is two polygons P and Q in the plane, of total complexity n, each of them can be decomposed into at most k convex polygons. Our purpose is to find the translation that maximizes the area of overlap.

The Algorithm. We decompose the polygons P and Q into minimum number of interior disjoint convex polygons [17], in time $O\big(n + k^2 \min(k^2, n)\big)$ (some of these convex polygons can be empty). Then, for every pair P_i, Q_j, we compute an $\big(\epsilon, O(1/\epsilon^2), O(n/\epsilon^2)\big)$-approximation ψ_{ij} to the overlap function of P_i and Q_j, using Lemma 16, where $\epsilon = \varepsilon/k^2$.

Next, as each function ψ_{ij} is defined by an arrangement defined by $O(1/\epsilon^2)$ polygons, we overlay all these arrangements together, and compute for each face of the arrangement the function $\psi = \sum_{i,j} \psi_{ij}$. Inside such a face this function is the same, and it is a quadratic function. We then find the global maximum of this function, and return it as the desired approximation.

Analysis – Quality of approximation. For any translation t, we have that

$$\Big|\mu(\mathsf{P}, \mathsf{t} + \mathsf{Q}) - \psi(\mathsf{t})\Big| \leq \sum_{i=1}^{k}\sum_{j=1}^{k} \Big|\mu(\mathsf{P}_i, \mathsf{t} + \mathsf{Q}_j) - \psi_{ij}(\mathsf{t})\Big| \leq \sum_{i=1}^{k}\sum_{j=1}^{k} \epsilon\mu_{\max}(\mathsf{P}_i, \mathsf{Q}_j)$$
$$\leq \epsilon k^2 \mu_{\max}(\mathsf{P}, \mathsf{Q}) \leq \varepsilon \mu_{\max}(\mathsf{P}, \mathsf{Q}) \,.$$

Analysis – Running time. Computing each of the k^2 approximation function, takes $O\big((k/\varepsilon)^2 n\big)$ time. Each one of them is a $\big(\varepsilon/k, O(k^2/\varepsilon^2), O(k^2 n/\varepsilon^2)\big)$-approximation, which means that the final arrangement is the overlay of $O(k^4/\varepsilon^2)$ convex polygons, each of complexity $O(k^2 n/\varepsilon^2)$. In particular, any pair of such polygons can have at most $O(k^2 n/\varepsilon^2)$ intersection points, and thus the overall complexity of the arrangement of these polygons is $N = O\Big(\big(k^4/\varepsilon^2\big)^2 \big(k^2 n/\varepsilon^2\big)\Big) = O\big(k^{10}\varepsilon^{-6} n\big)$. Computing this arrangement can be done by a standard sweeping algorithm. Observing that every vertical line crosses only $O(k^4/\varepsilon^2)$ segments, imply that the sweeping can be done in $O(\log(k/\varepsilon))$ time per operation, which implies that the overall running time is

$$O\left(k^2 \frac{k^2}{\varepsilon^2} n + N \log \frac{k}{\varepsilon}\right) = O\left(\frac{k^{10}}{\varepsilon^6} n \log \frac{k}{\varepsilon}\right).$$

The result.

Theorem 4. *Given two simple polygons* P *and* Q *of total complexity* n*, one can compute a translation which* ε*-approximates the maximum area of overlap of* P *and* Q*. The time required is* $O(c'n)$ *where* $c' = \dfrac{k^{10}}{\varepsilon^6} \log \dfrac{k}{\varepsilon}$*, where* k *is the minimum number of convex polygons in the decomposition of* P *and* Q*.*

More specifically, one gets a data-structure, such that for any query translation t*, one can compute, in* $O(\log n)$ *time, an approximation* $\psi(\mathsf{t})$*, such that* $|\psi(\mathsf{t}) - \mu(\mathsf{P}, \mathsf{Q})| \leq \varepsilon \mu_{\max}(\mathsf{P}, \mathsf{Q})$*, where* $\mu_{\max}(\mathsf{P}, \mathsf{Q})$ *is the maximum area of overlap between* P *and* Q*.*

Note, that our analysis is far from tight. Specifically, for the sake of simplicity of exposition, it is loose in several places as far as the dependency on k and ε.

Acknowledgments. The authors would like to thank the anonymous referees for their insightful comments. In particular, the improved construction of Lemma 12 was suggested by an anonymous referee.

References

1. Ahn, H.K., Cheng, S.W., Kweon, H.J., Yon, J.: Overlap of convex polytopes under rigid motion. Comput. Geom. Theory Appl. 47(1), 15–24 (2014), `http://dx.doi.org/10.1016/j.comgeo.2013.08.001`
2. Ahn, H.K., Cheng, S.W., Reinbacher, I.: Maximum overlap of convex polytopes under translation. Comput. Geom. Theory Appl. 46(5), 552–565 (2013), `http://dx.doi.org/10.1016/j.comgeo.2011.11.003`
3. Ahn, H.K., Cheong, O., Park, C.D., Shin, C.S., Vigneron, A.: Maximizing the overlap of two planar convex sets under rigid motions. Comput. Geom. Theory Appl. 37(1), 3–15 (2007)
4. Alt, H., Fuchs, U., Rote, G., Weber, G.: Matching convex shapes with respect to the symmetric difference. Algorithmica 21, 89–103 (1998), `http://citeseer.nj.nec.com/267158.html`

5. Alt, H., Guibas, L.J.: Discrete geometric shapes: Matching, interpolation, and approximation. In: Sack, J.R., Urrutia, J. (eds.) Handbook of Computational Geometry, pp. 121–153. Elsevier (2000)
6. Avis, D., Bose, P., Toussaint, G.T., Shermer, T.C., Zhu, B., Snoeyink, J.: On the sectional area of convex polytopes. In: Proc. 12th Annu. Sympos. Comput. Geom., pp. 411–412 (1996)
7. Barequet, G., Har-Peled, S.: Efficiently approximating the minimum-volume bounding box of a point set in three dimensions. J. Algorithms 38, 91–109 (2001), `http://cs.uiuc.edu/~sariel/research/papers/98/bbox.html`
8. Barequet, G., Har-Peled, S.: Polygon containment and translational min-hausdorff-distance between segment sets are 3sum-hard. Internat. J. Comput. Geom. Appl. 11(4), 465–474 (2001)
9. de Berg, M., Cabello, S., Giannopoulos, P., Knauer, C., van Oostrum, R., Veltkamp, R.C.: Maximizing the area of overlap of two unions of disks under rigid motion. In: Hagerup, T., Katajainen, J. (eds.) SWAT 2004. LNCS, vol. 3111, pp. 138–149. Springer, Heidelberg (2004)
10. de Berg, M., Cheong, O., Devillers, O., van Kreveld, M., Teillaud, M.: Computing the maximum overlap of two convex polygons under translations. Theo. Comp. Sci. 31, 613–628 (1998), `http://link.springer-ny.com/link/service/journals/00224/bibs/31n5p613.html`
11. Chazelle, B., Liu, D., Magen, A.: Sublinear geometric algorithms. SIAM J. Comput. 35(3), 627–646 (2005)
12. Vigneron, A.: Geometric optimization and sums of algebraic functions. ACM Trans. Algo. 4, 1–4 (2014), `http://doi.acm.org/10.1145/2532647`
13. Cheong, O., Efrat, A., Har-Peled, S.: On finding a guard that sees most and a shop that sells most. Discrete Comput. 37(4), 545–563 (2007), `http://link.springer-ny.com/link/service/journals/00454/`
14. Gritzmann, P., Klee, V.: Inner and outer j-radii of convex bodies in finite-dimensional normed spaces. Discrete Comput. Geom. 7, 255–280 (1992), `http://link.springer-ny.com/link/service/journals/00454/`
15. Har-Peled, S.: Geometric Approximation Algorithms. Mathematical Surveys and Monographs, vol. 173. Amer. Math. Soc. (2011)
16. Har-Peled, S., Roy, S.: Approximating the maximum overlap of polygons under translation. CoRR abs/1406.5778 (2014), `http://arxiv.org/abs/1406.5778`
17. Keil, J.M., Snoeyink, J.: On the time bound for convex decomposition of simple polygons. Internat. J. Comput. Geom. Appl. 12(3), 181–192 (2002)
18. Mount, D.M., Silverman, R., Wu, A.Y.: On the area of overlap of translated polygons. Computer Vision and Image Understanding: CVIU 64(1), 53–61 (1996), `http://www.cs.umd.edu/~mount/Papers/overlap.ps`
19. Sharir, M., Toledo, S.: Extremal polygon containment problems. Comput. Geom. Theory Appl. 4, 99–118 (1994)
20. Toussaint, G.T.: Solving geometric problems with the rotating calipers. In: Proc. IEEE MELECON 1983. pp. A10.02/1–4 (1983)
21. Vigneron, A.: Geometric optimization and sums of algebraic functions. ACM Trans. Algo. 4, 1–4 (2014), `http://doi.acm.org/10.1145/2532647`

Ordering without Forbidden Patterns[⋆]

Pavol Hell, Bojan Mohar, and Arash Rafiey

Simon Fraser University, Burnaby, Canada
pavol,mohar,arashr@sfu.ca

Abstract. Let $\mathcal{F}$ be a set of ordered patterns, i.e., graphs whose vertices are linearly ordered. An $\mathcal{F}$-free ordering of the vertices of a graph H is a linear ordering of $V(H)$ such that none of the patterns in $\mathcal{F}$ occurs as an induced ordered subgraph. We denote by $\textsc{Ord}(\mathcal{F})$ the decision problem asking whether an input graph admits an $\mathcal{F}$-free ordering; we also use $\textsc{Ord}(\mathcal{F})$ to denote the class of graphs that do admit an $\mathcal{F}$-free ordering. It was observed by Damaschke (and others) that many natural graph classes can be described as $\textsc{Ord}(\mathcal{F})$ for sets $\mathcal{F}$ of small patterns (with three or four vertices). This includes recognition of split graphs, interval graphs, proper interval graphs, cographs, comparability graphs, chordal graphs, strongly chordal graphs, and so on. Damaschke also noted that for many sets $\mathcal{F}$ consisting of patterns with three vertices, $\textsc{Ord}(\mathcal{F})$ is polynomial-time solvable by known algorithms or their simple modifications. We complete the picture by proving that *all* these problems can be solved in polynomial time. In fact, we provide a single master algorithm, which solves in polynomial time the problem $\textsc{Ord}_3$ in which the input is a set $\mathcal{F}$ of patterns, each with at most three vertices, and a graph H, and the problem is to decide whether or not H admits an $\mathcal{F}$-free ordering of the vertices. Our algorithm certifies non-membership by a forbidden substructure, and thus provides a single forbidden structure characterization for all the graph classes described by some $\textsc{Ord}(\mathcal{F})$ with $\mathcal{F}$ consisting of patterns with at most three vertices. This includes bipartite graphs, split graphs, interval graphs, proper interval graphs, chordal graphs, and comparability graphs. Many of the problems $\textsc{Ord}(\mathcal{F})$ with $\mathcal{F}$ consisting of larger patterns have been shown to be NP-complete by Duffus, Ginn, and Rödl, and we add some additional examples.

We also discuss a bipartite version of the problem, $\textsc{BiOrd}(\mathcal{F})$, in which the input is a bipartite graph H with a fixed bipartition of the vertices, and we are given a set $\mathcal{F}$ of bipartite patterns. We give a unified polynomial-time algorithm for all problems $\textsc{BiOrd}(\mathcal{F})$ where $\mathcal{F}$ has at most four vertices, i.e., we solve the analogous problem $\textsc{BiOrd}_4$. This is also a certifying algorithm, and it yields a unified forbidden substructure characterization for all bipartite graph classes described by some $\textsc{BiOrd}(\mathcal{F})$ with $\mathcal{F}$ consisting of bipartite patterns with at most four vertices. This includes chordal bipartite graphs, co-circular-arc bipartite graphs, and bipartite permutation graphs. We also describe some examples of digraph ordering problems and algorithms.

We conjecture that for every set $\mathcal{F}$ of forbidden patterns, $\textsc{Ord}(\mathcal{F})$ is either polynomial or NP-complete.

[⋆] Supported by NSERC, Canada.

1 Problem Definition and Motivation

For every positive integer k we write $[k] = \{1, 2, \ldots, k\}$, $E_k = \{\{i,j\} \mid i,j \in [k], i \neq j\}$, and $\mathcal{F}_k = 2^{E_k}$. Each element in $\mathcal{F}_k$ can be viewed as a labelled graph on vertex set $[k]$ and is called a *pattern* of *order* k, or simply a k*-pattern.* Given an input graph H and a linear ordering $<$ of its vertices, we say that a pattern $F \in \mathcal{F}_k$ *occurs* in H (under the ordering $<$) if H contains vertices $v_1 < v_2 < \cdots < v_k$ such that the induced ordered subgraph on these vertices is isomorphic to F, i.e., for every $i, j \in [k]$, $v_i v_j \in E(H)$ if and only if $\{i,j\} \in F$. For convenience, we shall henceforth write ij to simplify notation for unordered pairs $\{i,j\}$.

For a set $\mathcal{F} \subseteq \mathcal{F}_k$ we say that a linear ordering $<$ of $V(H)$ is $\mathcal{F}$*-free* if none of the patterns in $\mathcal{F}$ occurs in $<$. The problem $\textsc{Ord}(\mathcal{F})$ asks whether or not the input graph H has an $\mathcal{F}$-free ordering. We also consider the problem $\textsc{Ord}_k$ that asks, for an input $\mathcal{F} \subseteq \mathcal{F}_k$ and a graph H, whether or not H has an $\mathcal{F}$-free ordering.

The problems $\textsc{Ord}(\mathcal{F})$ can be viewed as 2-satisability problems with additional ordering constraints, or as special ternary constraint satisfaction problems. In neither of these cases are there general polynomial time algorithms known for their solution. (See also [15].)

The problems $\textsc{Ord}(\mathcal{F})$ have been studied by Damaschke [5], Duffus, Ginn, and Rödl [6], and others. In particular, Damaschke lists many graph classes that can be equivalently described as $\textsc{Ord}(\mathcal{F})$. For example (see [2]), it is well known that a graph H is chordal if and only if it admits an $\mathcal{F}$-free ordering for $\mathcal{F}$ consisting of the single pattern $\{12, 13\}$, and H is an interval graph if and only if it admits an $\mathcal{F}$-free ordering for $\mathcal{F}$ consisting of the pattern $\{\{13\}, \{13, 23\}\}$. Similar sets of patterns from $\mathcal{F}_3$ describe proper interval graphs, bipartite graphs, split graphs, and comparability graphs [5]. With patterns from $\mathcal{F}_4$ we can additionally describe strongly chordal graphs [7], circular-arc graphs [22], and many other graph classes.

Analogous definitions apply to bipartite graphs: a *bipartite pattern of order* k is a bipartite graph whose vertices in each part of the bipartition are labelled by elements of $[\ell]$ and $[\ell']$, respectively, with $\ell + \ell' \leq k$. We again denote by $\mathcal{B}_k$ the set of all bipartite patterns of order k. The problem $\textsc{BiOrd}(\mathcal{F})$ for a fixed $\mathcal{F} \subseteq \mathcal{B}_k$ asks whether or not an input bipartite graph H with a given bipartition $V(H) = U \cup V$ admits an ordering of U and of V so that no pattern from $\mathcal{F}$ occurs. We also define the corresponding problem $\textsc{BiOrd}_k$ in which both $\mathcal{F} \subseteq B_k$ and H with $V(H) = U \cup V$ are part of the input.

Several known bipartite graph classes can be characterized as $\textsc{BiOrd}(\mathcal{F})$ for $\mathcal{F} \subseteq \mathcal{B}_4$. For instance, for $\mathcal{F} = \{11', 31'\}$ (here $\ell = 3, \ell' = 1$), the class $\textsc{BiOrd}(\mathcal{F})$ consists precisely of convex bipartite graphs, and $\mathcal{F} = \{\{11', 12', 21'\}, \{12', 21'\}, \{12', 21', 22'\}\}$ (here $\ell = \ell' = 2$) similarly defines bipartite permutation graphs (a.k.a., proper interval bigraphs) [14,25,26]. One can similarly obtain the classes of chordal bipartite graphs, and bipartite co-circular arc bigraphs [17].

Summary of Our Main Results

We show that $\textsc{Ord}_3$ and $\textsc{BiOrd}_4$ are solvable in polynomial time. In particular, this completes the picture analyzed by Damaschke [5], and proves that all $\textsc{Ord}(\mathcal{F})$ with $\mathcal{F} \subseteq \mathcal{F}_3$ are polynomial-time solvable; similarly, all $\textsc{BiOrd}(\mathcal{F})$ with $\mathcal{F} \subseteq \mathcal{B}_4$ are polynomial-time solvable.

We also discuss digraphs with forbidden patterns on three vertices, and present two classes of digraphs for which our algorithm can be deployed to obtain the desired ordering without forbidden patterns.

We further describe sets $\mathcal{F} \subseteq \mathcal{F}_4$ for which $\textsc{Ord}(\mathcal{F})$ is polynomial time solvable and other sets $\mathcal{F} \subseteq \mathcal{F}_4$ for which the same problem is NP-complete. Many more NP-complete cases of $\textsc{Ord}(\mathcal{F})$ are presented in [6]; in particular, the authors of [6] conjecture that any $\mathcal{F}$ consisting of a single 2-connected pattern (other than a complete graph) yields an NP-complete $\textsc{Ord}(\mathcal{F})$.

Our master algorithm for $\textsc{Ord}_3$ provides a unified approach to all recognition problems for classes $\textsc{Ord}(\mathcal{F})$ with $\mathcal{F} \subseteq \mathcal{F}_3$, including all the well known graph classes mentioned earlier. Our algorithm is a certifying algorithm, and so it also provides a unified obstruction characterization for all these graph classes. (We note that these graphs have different ad-hoc obstruction characterizations [13].) A similar situation occurs with $\textsc{BiOrd}(\mathcal{F})$ with $\mathcal{F} \subseteq \mathcal{B}_4$ and classes characterized as $\textsc{BiOrd}(\mathcal{F})$ with $\mathcal{F} \subseteq \mathcal{B}_4$, including the well known classes of bipartite graphs mentioned earlier. We note that these special graph classes received much attention in the past; efficient recognition algorithms and structural characterizations can be found in [1,3,4,10,16,23,25,27] and elsewhere, cf. [2,13].

The algorithms use a novel idea of an auxiliary digraph. We believe this will be useful in other situations, and we have used similar digraphs in [9,19]. The algorithm for $\textsc{Ord}_3$ runs in time $O(n^3)$ where n is the number of vertices of H and in several cases (when the family $\mathcal{F}$ is particularly nice) it runs in time $O(nm)$, where m is the number of edges of H. The algorithm for $\textsc{BiOrd}_4$ runs in time $O(n^4)$ and in some cases in time $O(n^2m)$. We note that several special cases have recognition algorithms whose time complexity is $O(m + n)$, so we are definitely paying a price for having a unified algorithm; we note that the auxiliary digraph we use has $\Omega(nm)$ edges, so this technique is not likely to produce a linear time unified algorithm.

We conjecture that for every set $\mathcal{F}$ of forbidden patterns, $\textsc{Ord}(\mathcal{F})$ is either polynomial or NP-complete and provide some additional evidence for this dichotomy.

2 Algorithm for $\textsc{Ord}_3$ on Undirected Graphs

Consider an input graph H and a set of patterns $\mathcal{F} \subseteq \mathcal{F}_3$. Note that $\mathcal{F}$ imposes a constraint on any three vertices x, y, z of H. This means that whenever (x, y, z) induce a subgraph isomorphic to a pattern from $\mathcal{F}$ and x is before y, then z must not be after y.

We first construct an auxiliary digraph H^+, which we call a *constraint digraph*. The vertex set of H^+ consists of the ordered pairs $(x, y) \in V(H) \times V(H)$, $x \neq y$, and the arcs of H^+ are defined as follows. There is an arc from (x, y) to (z, y) and an arc from (y, z) to (y, x) whenever the vertices x, y, z ordered as $x < y < z$ induce a forbidden pattern in $\mathcal{F}$. We say that a pair (x, y) *dominates* (x', y') and we write $(x, y) \to (x', y')$ if there is an arc from (x, y) to (x', y') in H^+.

Consider a strong component S of H^+. The dual component $\overline{S}$ of S consists of all the pairs (y, x) where $(x, y) \in S$. Note that if $(x, y) \to (x, z)$, then $(z, x) \to (y, x)$, and vice versa.

There are two operations that appear naturally when dealing with orderings and forbidden patterns [5]. If we replace each pattern in $\mathcal{F}$ with its *complement* (change edges to nonedges and vice versa), thus obtaining a set $\overline{\mathcal{F}}$, then a linear ordering of $V(H)$ is $\mathcal{F}$-free for H if and only if it is $\overline{\mathcal{F}}$-free for the complementary graph $\overline{H}$. Another equivalence is obtained by replacing $\mathcal{F}$ with patterns that represent the same induced subgraphs but with the reversed order, e.g., replacing $\{12, 13\}$ by $\{32, 31\}$. Then a linear ordering will be $\mathcal{F}$-free if and only if the reverse ordering will be free of the reversed patterns. We rely on these two properties in some of our proofs.

In general, the structure of the digraph H^+ depends on the patterns. It is easy to see that if $\{12, 23\}$ or $\{13\}$ is the only forbidden pattern in $\mathcal{F} \subset \mathcal{F}_3$, then $(u, v)(u', v')$ is an arc of H^+ if and only if $(u', v')(u, v)$ is an arc of H^+, i.e. $(u, v)(u', v')$ is a symmetric arc of H^+ and hence H^+ is a graph. On the other hand, if $\{12, 13\}$ is the only forbidden pattern in $\mathcal{F}$, then H^+ is a digraph without digons and if $(u, v)(u', v')$ is an arc, then $(u', v')(u, v)$ is not an arc of H^+.

If all pairs $(x_0, x_1), (x_1, x_2), ..., (x_{n-1}, x_n), (x_n, x_0)$, $n \geq 1$, are in the same subset D of $V(H^+)$ then we say that $(x_0, x_1), (x_1, x_2), ..., (x_{n-1}, x_n), (x_n, x_0)$ is a *circuit* in D.

Lemma 1. *Let $\mathcal{F} \subseteq \mathcal{F}_3$ and let H^+ be the constraint digraph of H with respect to $\mathcal{F}$. If there exists a circuit in a strong component S of H^+, then H has no $\mathcal{F}$-free ordering.*

Proof. For a contradiction suppose $<$ is an $\mathcal{F}$-free ordering. Consider a circuit $(x_0, x_1), \ldots, (x_{n-1}, x_n), (x_n, x_0)$ in S. Since S is strong, there is a directed path P_i from (x_i, x_{i+1}) to (x_{i+1}, x_{i+2}) in S. If $x_i < x_{i+1}$ then following the path P_i in S we conclude that we must have $x_{i+1} < x_{i+2}$. Thus, if $x_0 < x_1$, then eventually by following each P_j, $0 \leq j \leq n$, we conclude that $x_0 < x_1 < \cdots < x_n < x_0$. This is a contradiction. Thus we must have $x_1 < x_0$. Now there is a path P'_i in $\overline{S}$ and hence by following the paths P'_i we eventually conclude that $x_1 < x_0 < x_n < \cdots < x_2 < x_1$, yielding a contradiction. $\square$

Lemma 2. (a) *Suppose $\emptyset \in \mathcal{F} \subseteq \mathcal{F}_3$.*

If H contains an independent set of three vertices, then H^+ has a strong component with a circuit and H has no $\mathcal{F}$-free ordering.

Otherwise H^+ is the same for $\mathcal{F}$ and for $(\mathcal{F} \setminus \{\emptyset\})$, and H has an $\mathcal{F}$-free ordering if and only if it has an $(\mathcal{F} \setminus \{\emptyset\})$-free ordering.

(b) *Suppose $\{12, 13, 23\} \in \mathcal{F} \subseteq \mathcal{F}_3$.*

If H contains a triangle, then H^+ has a strong component with a circuit and H has no $\mathcal{F}$-free ordering.

Otherwise H^+ is the same for $\mathcal{F}$ and for $(\mathcal{F} \setminus \{\emptyset\})$, and H has an $\mathcal{F}$-free ordering if and only if it has an $(\mathcal{F} \setminus \{\{12, 13, 23\}\})$-free ordering.

Proof. We only prove part (a) since the proof of (b) is similar.

Let a, b, c be pairwise nonadjacent vertices of H. If $\emptyset \in \mathcal{F}$, then $(a, b) \to (c, b)$, and $(c, b) \to (a, b)$, thus (a, b) and (c, b) are in the same strong component of H^+. Similarly, we have $(a, b) \to (a, c)$, and $(a, c) \to (a, b)$, thus (a, b) and (a, c) are in the same strong component of H^+. By symmetry, applied to other pairs, we conclude that all ordered pairs of two distinct vertices from the set $\{a, b, c\}$ are in the same strong component S of H^+. Clearly, $(a, b), (b, a)$ is a circuit in S.

As for the second part of the claim, if H has no independent set of three vertices, then $\emptyset$ contributes no restriction to orderings of $V(H)$, so both claims follow. □

Our main result is the following theorem which implies that Ord3 is solvable in polynomial time. (In fact, its proof will amount to a polynomial-time algorithm to actually construct an $\mathcal{F}$-free ordering if one exists.)

Theorem 1. *Let $\mathcal{F} \subseteq \mathcal{F}_3$ and let H^+ be the constraint digraph of H with respect to $\mathcal{F}$. Then H has an $\mathcal{F}$-free ordering if and only if no strong component of H^+ contains a circuit.*

Theorem 1 will follow from the correctness of our algorithm for Ord3. The proof of correctness will be given in a full journal version of this note [18].

Note that Theorem 1 provides a universal forbidden substructure (namely a circuit in a strong component of H^+) characterizing the membership in graph classes as varied as chordal graphs, interval graphs, proper interval graphs, comparability graphs, and co-comparability graphs.

We say a strong component S of H^+ is a *sink component* if there is no arc from S to a vertex outside S in H^+. Consider a subset D of the pairs in $V(H^+)$. We say that a strong component S of $H^+ \setminus (D \cup \overline{D})$ is *green with respect to D* if there is no arc from an element of S to a vertex in $H^+ \setminus (D \cup \overline{D} \cup \underline{S})$. This is equivalent to the condition that S is a sink component in $H^+ \setminus (D \cup \overline{D})$.

In the algorithm below, we start with an empty set D and we construct the final set D step by step. After each step of the algorithm, D (and hence also $\overline{D}$) is the union of vertex-sets of strong components of H^+ and neither D nor $\overline{D}$ contains a circuit. Each strong component S of H^+ either belongs to D or $\overline{D}$ or $V(H^+) \setminus (D \cup \overline{D})$. At the end of the algorithm $D \cup \overline{D}$ is a partition of the vertices (pairs) in $V(H^+)$ such that whenever $(x, y), (y, z) \in D$ then $(x, z) \in D$. We will say that D satisfies *transitivity condition.* At the end of the algorithm we place x before y whenever $(x, y) \in D$ and we obtain the desired ordering. We say a strong component is *trivial* if it has only one element otherwise it is called *non-trivial.*

Ordering with forbidden 3-patterns, Ord$_3$
Input: A graph H and a set $\mathcal{F} \subseteq \mathcal{F}_3$ of forbidden patterns on three vertices
Output: An $\mathcal{F}$-free ordering of the vertices of H or report that there is no such ordering.

Algorithm for Ord$_3$

1. If a strong component S of H^+ contains a circuit then report that no solution exists and exit. Otherwise, remove $\emptyset$ and $\{12, 13, 23\}$ from $\mathcal{F}$. If $\mathcal{F}$ is empty after this step, then return any ordering of vertices of H and stop.
2. Set D to be the empty set.
3. Choose a strong component S of H^+ that is green with respect to D. The choice is made according to the following rules.
   a) If $\mathcal{F}$ contains one of the forbidden patterns $\{13, 23\}, \{12, 13\}, \{12, 23\}$, then the priority is given to strong components containing a pair (x, y) with $xy \in E(H)$. If there is a choice then it is preferred S to be a trivial component. Subject to these preferences, if there are several candidates, then priority is given to the ones that are sink components in H^+.
   b) If $\mathcal{F}$ contains one of $\{12\}, \{23\}, \{13\}$, then priority is given to a strong component S containing (x, y) with $xy \notin E(H)$. If there is a choice, then the priority is given to trivial components, and if there are several candidates for S, then preference is given to the sink components in H^+.
4. If by adding S into D we do not close a circuit, then we add S into D and discard $\overline{S}$. Otherwise we add $\overline{S}$ and its outsection (all vertices in H^+ that are reachable from $\overline{S}$) into D and discard S and its insection (the vertices that can reach S). Return to Step 3 if there are some strong components of H^+ left.
5. For every $(x, y) \in D$, place x before y in the final ordering.

Our proof of the correctness of the algorithm will, in particular, also imply Theorem 1.

Corollary 1. *Each problem* Ord($\mathcal{F}$) *with* $\mathcal{F} \subseteq \mathcal{F}_3$ *can be solved in polynomial time.*

Remark. Our algorithm is linear in the size of H^+. The number of edges in H^+ is at most n^3 since each pair (x, y) has at most n out-neighbors. Thus the algorithms runs in $O(n^3)$, where $n = |V(H)|$. In some cases, e.g., when $|\mathcal{F}| = 1$, this can be improved to $O(nm)$, where $m = |E(H)|$.

2.1 Characterization of Obstructions

Many of the known graph classes discussed here have obstruction characterizations, usually in terms of forbidden induced subgraphs or some other forbidden substructures. A typical example is chordal graphs, whose very definition is a forbidden induced subgraph description: no induced cycles of length greater than three. Interval graphs have been characterized by Lekkerkerker and Boland [21] as not having an induced cycle of length greater than three, and no substructure called an asteroidal triple. Proper interval graphs have been characterized by the absence of induced cycles of length greater than three, and three special graphs usually called net, tent, and claw [29]. Comparability graphs have a similar forbidden substructure characterization [11].

The constraint digraph offers a natural way to define a common obstruction characterization for all these graph classes. In fact, Theorem 1 can be viewed as an obstruction characterization of $\textsc{Ord}(\mathcal{F})$ for any $\mathcal{F} \subseteq \mathcal{F}_3$, i.e., each of these classes is characterized by the absence of a circuit in a strong component of the constraint digraph. Moreover, our algorithm is a certifying algorithm, in the sense that when it fails, it identifies a circuit in a strong component of H^+.

For some of the sets $\mathcal{F} \subseteq \mathcal{F}_3$, we have an even simpler forbidden substructure characterization. We say x, y is an *invertible pair* of H if (x, y) and (y, x) belong to the same strong component of H^+. (Thus an invertible pair is precisely a circuit of length two.) We say $\mathcal{F}$ is *nice* if it is one of the following sets

$$\{\{13\}\}, \{\{12, 23\}\}, \{\{13\}, \{13, 23\}\}, \{\{13\}, \{12, 13\}, \{13, 23\}\}.$$

By following the correctness proof of our algorithm, it will be seen that if $\mathcal{F}$ is nice, then the algorithm does not create a circuit as long as every strong component S of H^+ has $S \cap \overline{S} = \emptyset$. Thus we obtain the following theorem for nice sets $\mathcal{F}$.

Theorem 2. *Suppose $\mathcal{F}$ is nice. A graph H admits an $\mathcal{F}$-free ordering if an only if it does not have an invertible pair.* □

In fact the correctness proof will show that if there is any circuit in a strong component of H^+, then there is also a circuit of length two.

Theorem 2 applies to, amongst others, interval graphs, proper interval graphs, comparability graphs and co-comparability graphs.

3 Forbidden Patterns in Bipartite Graphs

In this section we consider bipartite graphs H with a fixed bipartition $U \cup V$. We prove that $\textsc{BiOrd}_4$ is polynomial-time solvable, and so $\textsc{BiOrd}(\mathcal{F})$ is polynomial-time solvable for each $\mathcal{F} \subseteq \mathcal{B}_4$. Each forbidden pattern $F \in \mathcal{F}$ imposes constraints for those 4-tuples of vertices that induce a subgraph isomorphic to $\mathcal{F}$. We construct an auxiliary digraph H^+, that we also call a *constraint digraph.* The vertex set of H^+ consists of the pairs $(x, y) \in (U \times U) \cup (V \times V)$, where $x \neq y$, and the arc-set of H^+ is defined as follows.

There is an arc from (x, y) to (z, y) and an arc from (y, z) to (y, x) whenever the vertices x, y, z from the same part (U or V) of the bipartition, ordered $x < y < z$, together with some vertex v from the other part of the bipartition (V or U), induce a forbidden pattern in $\mathcal{F}$. There is also an arc from (x, y) to (u, v) and an arc from (v, u) to (y, x) whenever the vertices x, y from the same part, ordered as $x < y$, together with vertices u, v from the other part, ordered as $v < u$, induce a pattern in $\mathcal{F}$.

We say that a pair (x, y) *dominates* (x', y') and we write $(x, y) \rightarrow (x', y')$ if there is an arc from (x, y) to (x', y') in H^+.

A *circuit* in a subset D of H^+ is a sequence of pairs $(x_0, x_1), (x_1, x_2), \dots,$ $(x_{n-1}, x_n), (x_n, x_0)$, $n \geq 1$, that all belong to D. Observe that $x_0, x_1, \dots, x_n$ belong to the same bipartition part of $V(H)$.

Ordering with bipartite forbidden 4-patterns, BiOrd$_4$
Input: A bigraph $H = (U, V)$ and a set $\mathcal{F} \subseteq \mathcal{B}_4$ of bipartite forbidden patterns on four vertices
Output: And ordering of the vertices in U and an ordering of the vertices in V that is a $\mathcal{F}$-free ordering or report that there is no such ordering.

Algorithm for BiOrd$_4$

1. If a strong component S of H^+ contains a circuit then report that no solution exists and exit. Otherwise, remove $\emptyset$, $\{11', 12', 21', 22'\}$, $\{11', 21', 31'\}$ and $\{11', 22', 13'\}$ from $\mathcal{F}$. If $\mathcal{F}$ is empty after this step, then return any ordering of vertices of H and stop.
2. Set D to be the empty set.
3. Choose a strong component S of H^+ that is green with respect to D. The choice is made according to the following rules.
   a) If $\mathcal{F}$ contains one of the forbidden patterns $\{11', 12', 21'\}, \{12', 21', 22'\}$, $\{11', 12', 22'\}, \{11', 21', 22'\}$ then priority is given to a component S containing (x, y) where x, y have a common neighbor in H. If there is a choice then it is preferred S to be a trivial component. Subject to these preferences, if there are several candidates, then priority is given to the ones that are sink components in H^+.
   b) If $\mathcal{F}$ contains one of the forbidden patterns $\{11'\}, \{22'\}), \{12'\}, \{21'\}$ then priority is given to a component S containing (x, y) where x, y have a common non-neighbor in H. If there is a choice then it is preferred S to be a trivial component. Subject to these preferences, if there are several candidates, then priority is given to the ones that are sink components in H^+.
4. If by adding S into D we do not close a circuit, then we add S into D and discard $\overline{S}$. Otherwise we add $\overline{S}$ and its outsection (all vertices in H^+ that are reachable from $\overline{S}$) into D and discard S and its insection (the vertices that can reach S). Return to Step 3 if there are some strong components of H^+ left.
5. For every $(x, y) \in D$, place x before y in the final ordering.

A polynomial-time solution to BiOrd$_4$ is implicit in the following fact, the main result of this section.

Theorem 3. *Let $\mathcal{F} \subseteq \mathcal{B}_4$ and let H^+ be the constraint digraph of H with respect to $\mathcal{F}$. Then H has an $\mathcal{F}$-free ordering of its parts if and only if no strong component of H^+ contains a circuit.*

The proof, and the correctness of the algorithm will also be proved in the full version [18].

Corollary 2. *Each problem* $\textsc{BiOrd}(\mathcal{F})$ *with* $\mathcal{F} \subseteq \mathcal{F}_4$ *can be solved in polynomial time.*

3.1 Obstruction Characterizations

It is similarly the case that the constraint digraph offers a unifying concept of an obstruction for graph classes $\textsc{BiOrd}(\mathcal{F})$, $\mathcal{F} \subseteq \mathcal{B}_4$. Namely, Theorem 3 characterizes all these classes by the absence of a circuit in a strong component of the constraint digraph. In some cases we can again simplify the obstructions to a bipartite version of invertible pairs.

An *invertible pair* of H is a pair of vertices u, v from the same part of the bipartition such that both (u, v) and (v, u) lie on the same directed cycle of H^+. Thus a circuit of length two in a strong component of H^+ corresponds precisely to an invertible pair.

For our first illustration we discuss the case of co-circular-arc bigraphs. A *co-circular-arc bigraph* is a bipartite graph whose complement is a circular arc graph. A complex characterization of co-circular-arc graphs by seven infinite families of forbidden induced subgraphs has been given in [27], later simplified to a Lekerkerker-Boland-like characterization by forbidden induced cycles and edge asteroids in [8]. These graphs seem to be the bipartite analogues of interval graphs, see [8]. One reason may be that co-circular-arc bigraphs are precisely the intersection graphs of 2-directional rays [24].

We observe the following simple characterization.

Theorem 4. *Let* $H = (B, W)$ *be a bipartite graph. Then the following are equivalent.*

(1) H is a co-circular-arc bigraph.
(2) H admits an $\mathcal{F}$*-free ordering where* $\mathcal{F} = \{\{12', 21'\}, \{12', 21', 22'\}\}$.
(3) H has no invertible pair.

Proof. It was shown in [20] that (1) and (2) are equivalent. (In [20] $\mathcal{F}$-free orderings are described by an equivalent notion of so-called min orderings.) According to proof of Theorem 3 for $\mathcal{F}$ we assume that H does not have an invertible pair. Therefore (2) and (3) are equivalent and hence the theorem is proved. □

A bipartite graph $G = (V, U)$ is called *proper interval bigraph* if the vertices in each part can be represented by an inclusion-free family of intervals, and a vertex from V is adjacent to a vertex from U if and only if their intervals intersect. They are also known as bipartite permutation graphs [14,25,26].

Theorem 5. *Let* $H = (B, W)$ *be a bipartite graph. Then the following are equivalent.*

(1) H is a proper interval bigraph
(2) H admits an $\mathcal{F}$*-free ordering where* $\mathcal{F} = \{\{12', 21'\}, \{12', 21', 22'\}, \{11', 12', 21'\}\}$.
(3) H does not have an invertible pair.

Proof. It was noted in [14] that H admits an $\mathcal{F}$-free ordering if and only if H is a bipartite permutation graph (proper interval bigraph). (In [14] $\mathcal{F}$-free orderings are described by an equivalent notion of min-max orderings.) Therefore (1) and (2) are equivalent. It is easy to see that if no strong component of H^+ contains a circuit of length two, i.e. if H has no invertible pair, then the Algorithm for BiOrd_4 does not create a circuit. Therefore (2) and (3) are equivalent and hence the theorem is proved. □

4 Remarks and Conclusions

As noted earlier, Duffus, Ginn, and Rödl have found many examples of NP-complete problems $\text{Ord}(\mathcal{F})$; in fact if $\mathcal{F}$ consists of a single ordered pattern, they offered strong evidence that $\text{Ord}(\mathcal{F})$ may be NP-complete as soon as the pattern is 2-connected. We offer just two simple examples to illustrate some NP-complete cases.

Proposition 1. *For every $k \geq 4$ there exists a set $\mathcal{F} \subseteq \mathcal{F}_k$ such that* $\text{Ord}(\mathcal{F})$ *is NP-complete.*

Proof. We show that if $\mathcal{F}$ is a set of all those forbidden patterns on k vertices, which contain $\{12, 23, 34, \ldots, (k-1)k\}$ as a subset, then Ord_k is NP-complete. We reduce the problem to $(k-1)$-colorability. Let H be an arbitrary graph. If H is $(k-1)$-colorable with color classes $X_1, X_2, \ldots, X_{k-1}$, then we put all the vertices in X_i before all the vertices in X_{i+1}, $1 \leq i \leq k-2$. This way we obtain an ordering of the vertices and it is clear that it does not contain any of the forbidden patterns in $\mathcal{F}$.

Now suppose there is an ordering $v_1, v_2, \ldots, v_n$ of the vertices in H without seeing any forbidden pattern in $\mathcal{F}$. Let X_1 be the set of vertices v_j, $1 \leq j \leq n$ that have no neighbor before v_j. Now for every $2 \leq i \leq k-1$, let X_i be the set of vertices v_j, $1 \leq j \leq n$, from the set $Y_i = V(H) \setminus (\cup_{\ell=1}^{i-1} X_\ell)$ that have no neighbor in Y_i that is before v_j. Note that by definition each X_i, $1 \leq i \leq k-1$ is an independent subset of H. Moreover $V(H) = \cup_{\ell=1}^{k-1} X_\ell$ as otherwise we obtain k vertices $u_1 < u_2 < \cdots < u_k$ where $u_j u_{j+1}$, $1 \leq j \leq k-1$, is an edge of H and hence we find a forbidden pattern from $\mathcal{F}$. Thus H is $(k-1)$-colorable. □

We note that in Damaschke's paper [5] the complexity of $\text{Ord}(\mathcal{F})$ was left open for $\mathcal{F} = \{12, 23, 34\}$. (However, other folklore solutions for this particular case have been reported since.)

In the case of bipartite graphs, we offer the following simple example.

Proposition 2. $\text{BiOrd}(\mathcal{F})$ *is NP-complete for the set $\mathcal{F} = \{\{11', 31', 51'\}\}$.*

Proof. Let M be an $m \times n$ matrix with entities 0 and 1. Finding an ordering of the columns such that in each row there are at most two sequences of consecutive 1's has been shown to be NP-complete in [12]. Now from an instance of a matrix M we construct a bipartite graph $H = (A, B, E)$ where A represents the set of columns and B represents the set of rows in M. There is an edge between $a \in A$

and $b \in B$ if the entry in M, corresponding to row a and column b is 1. Now if we were able to reorder to columns with the required property we would be able to find the ordering of H without seeing the forbidden pattern in F and vice versa. □

There are natural polynomial problems $\textsc{Ord}(\mathcal{F})$ for sets $\mathcal{F}$ of larger patterns. For instance, strongly chordal graphs are characterized as $\textsc{Ord}(\mathcal{F}), \mathcal{F} \subseteq \mathcal{F}_4$, in [7]. In fact, an algorithm similar to the one presented here can be developed for this case. (The authors will be happy to communicate the details to interested readers.)

There is a natural version of $\textsc{Ord}(\mathcal{F})$ for digraph patterns $\mathcal{F}$. We are given an input digraph H and a set $\mathcal{F}$ of forbidden digraph patterns (each digraph pattern is an ordered digraph). The decision problem asking whether an input digraph admits an ordering without forbidden patterns in $\mathcal{F}$ is denoted by $\textsc{DiOrd}(\mathcal{F})$. Let $\mathcal{D}_k$ denote the collection of sets $\mathcal{F}$ of digraph patterns with k vertices. The problem $\textsc{DiOrd}_k$ asks, for an input $\mathcal{F} \subseteq \mathcal{D}_k$ and a digraph H, whether or not H has an $\mathcal{F}$-free ordering.

The algorithms in [9,19] illustrate two cases where $\textsc{DiOrd}(\mathcal{F})$ problems have been solved by algorithms similar to the algorithm for $\textsc{Ord}_3$, and the obstructions characterized as invertible pairs. (The problems in [9,19] are not presented as $\textsc{Ord}(\mathcal{F})$, but they can easily be so reformulated.) We believe many other digraph problems can be similarly handled. In fact we wonder whether the problem $\textsc{DiOrd}(\mathcal{F})$ is polynomial for every set $\mathcal{D} \in \mathcal{D}_3$.

We conjecture that for every set $\mathcal{F}$ of forbidden patterns, $\textsc{Ord}(\mathcal{F})$ is either polynomial or NP-complete.

References

1. Booth, K.S., Lueker, G.S.: Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. Journal of Computer and System Sciences 13(3), 335–379 (1976)
2. Brandstädt, A., Le, V.B., Spinrad, J.: Graph Classes: A Survey. SIAM Monographs on Discrete Mathematics and Applications (1999)
3. Calamoneri, T., Caminiti, S., Petreschi, R., Olariu, S.: On the L(h,k)-labeling of co-comparability graphs and circular-arc graphs. Networks 53(1), 27–34 (2009)
4. Corneil, D.G., Olariu, S., Stewart, L.: The LBFS Structure and Recognition of Interval Graphs. SIAM J. Discrete Math. 23(4), 1905–1953 (2009)
5. Damaschke, P.: Forbidden Ordered Subgraphs. Topics in Combinatorics and Graph Theory, pp. 219–229 (1990)
6. Duffus, D., Ginn, M., Rödl, V.: On the computational complexity of ordered subgraph recognition. Random Structures and Algorithms 7(3), 223–268 (1995)
7. Farber, M.: Characterizations of strongly chordal graphs. Discrete Mathematics 43(2-3), 173–189 (1983)
8. Feder, T., Hell, P., Huang, J.: List homomorphisms and circular arc graphs. Combinatorica 19, 487–505 (1999)
9. Feder, T., Hell, P., Huang, J., Rafiey, A.: Interval graphs, adjusted interval graphs and reflexive list homomorphisms. Discrete Appl. Math. 160, 697–707 (2012)

10. Fulkerson, D.R., Gross, O.A.: Incidence matrices and interval graphs. Pacific J. Math., 835–855 (1965)
11. Gallai, T.: Transitiv orientierbare Graphen. Acta Math. Acad. Sci. Hung 18, 25–66 (1967)
12. Goldberg, P.W., Golumbic, M.C., Kaplan, H., Shamir, R.: Four strikes against physical mapping of DNA. J. Comput. Biol., 139–152 (1995)
13. Golumbic, M.C.: Algorithmic Graph Theory and Perfect Graphs. Academic Press (1980)
14. Gutin, G., Hell, P., Rafiey, A., Yeo, A.: A dichotomy for minimum cost graph homomorphisms. European J. Combinatorics 29, 900–911 (2008)
15. Guttmann, W., Maucher, M.: Variations on an ordering theme with constraints. In: Navarro, G., Bertossi, L., Kohayakawa, Y. (eds.) Proc. 4th IFIP International Conference on Theoretical Computer Science, TCS 2006. IFIP, vol. 209, pp. 77–90. Springer, Heidelberg (2006)
16. Habib, M., McConnell, R.: Ch. Paul and L. Viennot. Lex-BFS and Partition Refinement, with Applications to Transitive Orientation, Interval Graph Recognition, and Consecutive Ones Testing 234, 59–84 (2000)
17. Hell, P., Huang, J.: Interval bigraphs and circular arc graphs. Journal of Graph Theory 46, 313–327 (2004)
18. Hell, P., Mohar, B., Rafiey, A.: Ordering without forbidden patterns, arXiv (2014)
19. Hell, P., Rafiey, A.: Monotone Proper Interval Digraphs. SIAM J. Discrete Math. 26(4), 1576–1596 (2012)
20. Hell, P., Mastrolilli, M., Nevisi, M.M., Rafiey, A.: Approximation of Minimum Cost Homomorphisms. In: Epstein, L., Ferragina, P. (eds.) ESA 2012. LNCS, vol. 7501, pp. 587–598. Springer, Heidelberg (2012)
21. Lekkerkerker, C.G., Boland, J.C.: Representation of a finite graph by a set of intervals on the real line. Fundamenta Math. 51, 45–64 (1962)
22. Lin, M.C., Szwarcfiter, J.L.: Characterizations and recognition of circular-arc graphs and subclasses: A survey. Discrete Mathematics 309(18), 5618–5635 (2009)
23. Rose, D., Lueker, G., Tarjan, R.E.: Algorithmic aspects of vertex elimination on graphs. SIAM Journal on Computing 5(2), 266–283 (1976)
24. Shrestha, A.M.S., Tayu, S., Ueno, S.: On orthogonal ray graphs. Discrete Applied Math. 158, 1650–1659 (2010)
25. Spinrad, J.P., Brandstaedt, A., Stewart, L.: Bipartite permutation graphs. Discrete Applied Math. 18, 279–292 (1987)
26. Spinrad, J.: Efficient Graph Representations. AMS (2003)
27. Trotter, W.T., Moore, J.: Characterization problems for graphs, partially ordered sets, lattices, and families of sets. Discrete Math. 16, 361–381 (1976)
28. Trotter, W.T.: Combinatorics and Partially Ordered Sets–Dimension Theory. The Johns Hopkins University Press (1992)
29. Wegner, G.: Eigenschaften der Nerven homologisch-einfacher Familien im R^n, Ph.D. thesis, University of Göttingen (1967)

Halving Balls in Deterministic Linear Time*

Michael Hoffmann[1], Vincent Kusters[1], and Tillmann Miltzow[2]

[1] Department of Computer Science, ETH Zürich, Switzerland
{hoffmann,vincent.kusters}@inf.ethz.ch
[2] Institute of Computer Science, Freie Universität Berlin, Germany
miltzow@mi.fu-berlin.de

Abstract. Let $\mathcal{D}$ be a set of n pairwise disjoint unit balls in $\mathbb{R}^d$ and P the set of their center points. A hyperplane $\mathcal{H}$ is an *m-separator* for $\mathcal{D}$ if each closed halfspace bounded by $\mathcal{H}$ contains at least m points from P. This generalizes the notion of halving hyperplanes ($n/2$-separators). The analogous notion for point sets has been well studied. Separators have various applications, for instance, in divide-and-conquer schemes. In such a scheme any ball that is intersected by the separating hyperplane may still interact with both sides of the partition. Therefore it is desirable that the separating hyperplane intersects a small number of balls only.

We present three deterministic algorithms to bisect or approximately bisect a given set of n disjoint unit balls by a hyperplane: firstly, a linear-time algorithm to construct an αn-separator in $\mathbb{R}^d$, for $0 < \alpha < 1/2$, that intersects at most $cn^{(d-1)/d}$ balls, where c depends on d and α. The number of balls intersected is best possible up to the constant c. Secondly, we present a near-linear time algorithm to find an $(n/2-o(n))$-separator in $\mathbb{R}^d$ that intersects $o(n)$ balls. Finally, we give a linear-time algorithm to construct a halving line in $\mathbb{R}^2$ that intersects $O(n^{(5/6)+\varepsilon})$ disks.

Our results improve the runtime of a disk sliding algorithm by Bereg, Dumitrescu and Pach. In addition, our results improve and derandomize an algorithm to construct a space decomposition used by Löffler and Mulzer to construct an onion decomposition for imprecise points.

1 Introduction

Let $\mathcal{D}$ be a set of n pairwise disjoint unit balls in $\mathbb{R}^d$ and P the set of their center points. A hyperplane $\mathcal{H}$ is an *m-separator* for $\mathcal{D}$ if each closed halfspace bounded by $\mathcal{H}$ contains at least m points from P. This generalizes the notion of halving hyperplanes, which correspond to $n/2$-separators. The analogous notion of separating hyperplanes for point sets has been well studied (see, e.g, [11] for a survey). Separators have various applications, for instance in divide-and-conquer schemes In such a scheme any ball that is intersected by the separating hyperplane may still interact with both sides of the partition. Therefore it is desirable that the separating hyperplane intersects a small number of balls only.

* Partially supported by the ESF EUROCORES programme EuroGIGA, CRP GraDR and the Swiss National Science Foundation, SNF Project 20GG21-134306.

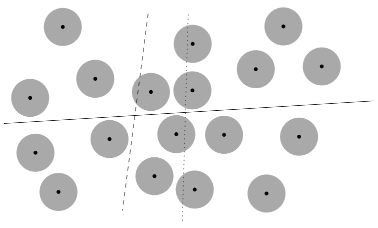


Fig. 1. A set of 18 disks and three separators. The dashed line forms a 6-separator. Both the solid line and the dotted line are halving lines. The solid line is preferable to the other two lines because it separates perfectly and does not intersect any disk.

Alon et al. [1] prove that for any set $\mathcal{D}$ in $\mathbb{R}^2$, there is a direction such that every line in this direction intersects $O(\sqrt{n \log n})$ disks. In particular, this guarantees the existence of a halving line that intersects $O(\sqrt{n \log n})$ disks. This result does not immediately extend to $\mathbb{R}^d$. Löffler and Mulzer [10] observed that this proof gives a randomized linear-time algorithm to find such a halving line. In this paper, we present three deterministic algorithms to find an m-separator that intersects $o(n)$ balls, for various m.

Theorem 1. *Given a set $\mathcal{D}$ of n pairwise disjoint unit balls in $\mathbb{R}^d$ and any $\alpha \in (0, 1/2)$, one can construct in $O(nd^2/(1-2\alpha))$ time a hyperplane $\mathcal{H}$ that intersects $O(d^3(n/(1-2\alpha))^{(d-1)/d})$ balls from $\mathcal{D}$ and such that each closed halfspace bounded by $\mathcal{H}$ contains at least αn balls from $\mathcal{D}$ (for n sufficiently large, depending on d and α).*

Theorem 2. *Given a set $\mathcal{D}$ of n pairwise disjoint unit balls in $\mathbb{R}^d$ and a function $f(n) \in \omega(1) \cap O(\log n)$, one can construct in $O(nd^2 f(n))$ time a hyperplane $\mathcal{H}$ such that each closed halfspace bounded by $\mathcal{H}$ contains at least $\frac{n}{2}(1 - 1/f(n)) = \frac{n}{2}(1 - o(1))$ balls from $\mathcal{D}$ (for n sufficiently large, depending on d).*

Theorem 3. *For any set $\mathcal{D}$ of n pairwise disjoint unit disks in $\mathbb{R}^2$ and any $\varepsilon > 0$ one can construct in $O(n)$ time a line ℓ that intersects $O(n^{(5/6)+\varepsilon})$ disks from $\mathcal{D}$ and such that each closed halfplane bounded by ℓ contains at least $n/2$ centers of disks from $\mathcal{D}$.*

We develop a generic algorithm in $\mathbb{R}^d$ that can be instantiated with different parameters to obtain the first two theorems. Note that Theorem 2 improves the separation of the center points (compared to Theorem 1) at the cost of increasing the running time slightly. Theorem 3 computes a true halving line in the plane.

Related work. Bereg at el. [4] (see also [13, Lemma 9.3.2]) strengthen the result of Alon et al. slightly by proving that there exists a direction such that any line with this direction has $O(\sqrt{n \log n})$ disks *within constant distance.* They use this lemma to prove that one can always move a set of n unit disks from a start to a target configuration in $3n/2 + O(\sqrt{n \log n})$ moves. Their algorithm runs in

$O(n^{3/2}(\log n)^{-1/2})$ time, which Theorem 3 improves to $O(n \log n)$ (simply by replacing part of their algorithm).

Held and Mitchell [7] introduced a paradigm to model data imprecision where the location of a point is not known exactly. Instead, for each point we are given a unit disk that is guaranteed to contain it. After preprocessing the disks in $O(n \log n)$ time, they can construct a triangulation of the actual point set in linear time. Löffler and Mulzer [10] follow the same model to construct the onion layer of an imprecise point set. They observed that the proof by Alon et al. immediately gives a randomized expected linear-time algorithm to find a halving line that intersects $O(\sqrt{n \log n})$ unit disks with probability at least 1/2. Using this algorithm they compute a (α, β)*-space decomposition tree*: a data structure similar to a binary space partition in which every line is an αk-separator that intersects at most k^{β} disks. They show that such a $(1/2 + \varepsilon, 1/2 + \varepsilon)$ space decomposition tree can be computed in $O(n \log n)$ expected time, for every $\varepsilon > 0$. Theorem 1 improves this to $O(n \log n)$ deterministic time. They also present a simple deterministic linear-time algorithm that guarantees that at least $n/10$ of the disks are completely on each side of some axis-parallel line. Next, they describe a more sophisticated, deterministic $O(n \log n)$ algorithm to compute a line ℓ such that there are at least $n/2 - cn^{5/6}$ disks completely to each side of ℓ. The algorithm uses an r-partition of the plane [12] to find good candidate lines. Theorem 3 can be used to improve running time of this algorithm to $O(n)$.

Tverberg [15] studies a related question. He proves that for every natural number k there is a number $K(k)$, such that given convex pairwise disjoint sets $C_1, \ldots, C_{K(k)}$, there always exists a line with some set completely on one side and k sets completely on the other side. Finally, the question has a continuous counterpart that has been solved recently [6].

Organization. In Section 2 we develop a generic algorithm to compute a separator in $\mathbb{R}^d$, where the trade-off between the number of intersected disks and the number of disk centers on each side is determined by a parameter. Theorem 1 and Theorem 2 follow from specific instances of this generic algorithm. Section 3 is devoted to the proof of Theorem 3. Our algorithm follows the approach used in the linear-time ham-sandwich cut algorithm [9]. It divides the line arrangement dual to the set of disk center points by vertical lines such that each slab (the region bounded by two consecutive vertical lines) contains at most a constant fraction of the vertices of the arrangement. In each iteration, the algorithm chooses one slab to continue with and discards the rest of the arrangement. Due to space contraints, we have to omit a number of proofs in this extended abstract.

2 Separating Balls in $\mathbb{R}^d$

In this section, we develop a generic algorithm to compute a separator for a given set of pairwise disjoint unit balls in $\mathbb{R}^d$. Using this generic algorithm, we will give two algorithms to compute an approximately halving hyperplane that intersects a sublinear number of balls.

In addition to the set $\mathcal{D}$ of n balls in $\mathbb{R}^d$, the generic algorithm has two more parameters. First, a number $b \in \{1, \ldots, n\}$ that quantifies the quality of the approximation: we will show that the hyperplane constructed by the algorithm forms an $(n-b)/2$-separator for $\mathcal{D}$. The main step of the algorithm consists in finding a direction d such that we are guaranteed to find a desired separator that is orthogonal to d. A second parameter $k \in \mathbb{N}$ of the algorithm specifies the number of different directions to generate and test during this step. As a rule of thumb, generating more directions results in a better solution, but the runtime of the algorithm increases proportionally. The algorithm works for certain combinations of these parameters only, as detailed in the following theorem.

Theorem 4. *Given a set $\mathcal{D}$ of n pairwise disjoint unit balls in $\mathbb{R}^d$ and parameters $b \in \{1, \ldots, n\}$ and $k \in \mathbb{N}$ that satisfy the conditions*

$$dn \leq kb \quad \text{and} \tag{1}$$

$$t := \left(\frac{V_d}{2d^{(d-2)/2}}\right)^{1/d} \frac{n^{1/d}}{k^{2-1/d}} > 2, \tag{2}$$

(where $V_d \sim (\sqrt{d\pi})^{-1}(2\pi e/d)^{d/2}$ is the volume of the d-dimensional unit ball), one can construct in $O(kdn)$ time a hyperplane $\mathcal{H}$ that intersects at most $2b/(t-2)$ balls from $\mathcal{D}$ and such that each closed halfspace bounded by $\mathcal{H}$ contains at least $(n-b)/2$ balls from $\mathcal{D}$.

More interesting than Theorem 4 in its full generality are the special cases stated as Theorem 1 and Theorem 2 above. Theorem 1 describes a version with running time linear in n and can be obtained by choosing $b = \lceil (1-2\alpha)n \rceil$ and $k = \lceil d/(1-2\alpha) \rceil$ for $\alpha \in (0, 1/2)$. Theorem 2 describes a version that achieves a separator that halves up to a lower order term. It can be obtained by choosing $b = \lceil n/f(n) \rfloor$ and $k = \lceil df(n) \rceil$ for a a very slowly growing function $f(n)$.

Overview of the algorithm. Our algorithm consists of two steps. In the first step, we find a direction in which the balls from $\mathcal{D}$ are "spread out nicely". More precisely, for an arbitrary (oriented) line ℓ consider the set P of points that results from orthogonally projecting all centers of balls from $\mathcal{D}$ onto ℓ. Denote by $p_1, \ldots, p_n$ the order of points from P sorted along ℓ. We want to find an $(n-b)/2$-separator orthogonal to ℓ. This means that the separating hyperplane $\mathcal{H}$ must intersect ℓ somewhere in between $p_{(n-b)/2}$ and $p_{(n+b)/2}$.

However, we also need to guarantee that not too many points from P are within distance one of $\mathcal{H}$, which may or may not be possible depending on the choice of ℓ. Therefore we try several possible directions/lines and select the first one among them that works. In order to evaluate the quality of a line, we use as a simple criterion the *spread*, defined to be the distance between $p_{(n-b)/2}$ and $p_{(n+b)/2}$. Given a line ℓ with sufficient spread, we can find a suitable $(n-b)/2$-separator orthogonal to ℓ in the second step of our algorithm, as follows (note the safety cushion of width one to the remaining disks of $\mathcal{D}$).

Lemma 1. *Given a set P of b (one-dimensional) points in an interval $[\ell, r]$ of length $w = r - \ell > 2$, we can find in $O(b)$ time a point $p \in (\ell + 1, r - 1)$ such that at most $2b/(w-2)$ points from P are within distance one of p.*

How to find a good direction. We try k different directions and stop as soon as we find a direction with spread at least t (see Theorem 4). For a given direction the spread can be computed in $O(dn)$ time using linear time rank selection [5]. In the remainder of this section, we will discuss how to select an appropriate set of directions such that one direction is guaranteed to have spread at least t.

For this we need a bound on the number of balls simultaneously within distance $w_1, \dots, w_d$ of some hyperplanes $\mathcal{H}_1, \dots, \mathcal{H}_d$. Below we give an easy formula based on a volume argument. This formula in turn motivates our choice of directions, which we will explain thereafter.

Lemma 2. *Let $\boldsymbol{v}_1, \boldsymbol{v}_2, \dots, \boldsymbol{v}_d \in S^{d-1} \subset \mathbb{R}^d$ be linearly independent directions and $\mathcal{H}_1, \mathcal{H}_2, \dots, \mathcal{H}_d$ hyperplanes with corresponding normal directions, then the maximal number of pairwise disjoint unit balls entirely within distance $w_1, \dots, w_d$ of $\mathcal{H}_1, \mathcal{H}_2, \dots, \mathcal{H}_d$, respectively, is bounded from above by*

$$\frac{2^d w_1 \dots w_d}{|\det(\boldsymbol{v}_1, \dots, \boldsymbol{v}_d)|V_d},$$

where V_d denotes the volume of the d-dimensional unit ball.

The bound in Lemma 2 depends on the determinant formed by the d direction vectors, which corresponds to the volume of the $(d-1)$-simplex spanned by them. In order to obtain a good upper bound, we must guarantee that this volume does not become too small. Ensuring this reduces to the *Heilbronn Problem*: Given $k \in \mathbb{N}$ and a compact region $P \subset \mathbb{R}^d$ of unit volume, how can we select k points from P as to maximize the area of the smallest d-simplex formed by these points? Heilbronn posed this question for $d = 2$; the natural generalization to higher dimension was studied by Barequet [3] and Lefmann [8]. We use the following simple explicit construction in $\mathbb{R}^2$ that goes back to Erdős and was generalized to higher dimension by Barequet.

Lemma 3 ([3,14]). *Given a prime k, let $P = \{p_0, \dots, p_{k-1}\} \subset [0,1]^d$ with*

$$p_i = \frac{1}{k}\left(i, i^2 \bmod k, \dots, i^d \bmod k\right).$$

Then the smallest d-simplex spanned by $d+1$ points from P has volume at least $1/(d!k^d)$.

Assuming k to be prime is not a restriction: If k is not prime, then by Bertrand's postulate there is a prime $k' \leq 2k$. We can compute k' efficiently, for instance, in $O(k/\log\log k)$ time using Atkin's sieve [2]. In order to obtain the desired direction vectors we proceed as follows: Use Lemma 3 to generate k points $p_0, \ldots, p_{k-1}$ in $[0,1]^{d-1}$. Then lift the points to $S^{d-1} \subset \mathbb{R}^d$ using the map

$$f : (x_1, \ldots, x_{d-1}) \mapsto \frac{(x_1 - \frac{1}{2}, \ldots, x_{d-1} - \frac{1}{2}, \frac{1}{2})}{||(x_1 - \frac{1}{2}, \ldots, x_{d-1} - \frac{1}{2}, \frac{1}{2})||}$$

and denote the resulting set of directions by $D = \{\boldsymbol{v}_0, \ldots, \boldsymbol{v}_{k-1}\}$ with $\boldsymbol{v}_i = f(p_i)$.

Lemma 4. *For any d vectors $\boldsymbol{v}_{i_1}, \ldots, \boldsymbol{v}_{i_d}$ from D we have $|\det(\boldsymbol{v}_{i_1}, \ldots, \boldsymbol{v}_{i_d})| \geq 2^{d-1}/((d-1)! d^{d/2} k^{d-1})$.*

We are now ready to prove Theorem 4.

Proof. Compute directions $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k$ as in Lemma 4 in time $O(kd)$. For each $i \in \{1, \ldots, k\}$ consider the sequence of center points of the disks in $\mathcal{D}$, sorted according to direction $\boldsymbol{v}_i$, and denote by S_i the middle b points in this order (rank $(n-b)/2$ up to $(n+b)/2$). We can bound

$$kb = \sum_{i=1}^{k} |S_i| \leq (d-1)n + \sum_{i_1 < \ldots < i_d} |S_{i_1} \cap \ldots \cap S_{i_d}|,$$

noting that a point that is contained in at most $d-1$ sets S_i is counted $d-1$ times on the right hand side, whereas a point that is contained in $a \geq d$ sets is counted $d - 1 + \binom{a}{d} \geq a$ times. Denote by w_i the width of S_i in direction $\boldsymbol{v}_i$ (which is the spread of $\boldsymbol{v}_i$). We claim that $w_i \geq t$, for some $i \in \{1, \ldots, k\}$. For the purpose of contradiction assume $w_i < t$, for all $i \in \{1, \ldots, k\}$. Together with Lemma 2 and Lemma 4 we get

$$\begin{aligned}
kb = \sum_{i=1}^{k} |S_i| &\leq (d-1)n + \sum_{i_1 < \ldots < i_d} \frac{2^d w_{i_1} \ldots w_{i_d}}{|\det(\boldsymbol{v}_{i_1}, \ldots, \boldsymbol{v}_{i_d})| V_d} \\
&< (d-1)n + \sum_{i_1 < \ldots < i_d} \frac{2^d t^d}{V_d} \frac{(d-1)! d^{d/2} k^{d-1}}{2^{d-1}} \\
&= (d-1)n + \binom{k}{d} \frac{2t^d (d-1)! d^{d/2} k^{d-1}}{V_d} \leq (d-1)n + \frac{2d^{(d-2)/2}}{V_d} t^d k^{2d-1}.
\end{aligned}$$

In combination with Condition (1) we get

$$dn \leq kb < (d-1)n + \frac{2d^{(d-2)/2}}{V_d} t^d k^{2d-1}$$

and so

$$t^d > \frac{V_d}{2d^{(d-2)/2}} \frac{n}{k^{2d-1}},$$

in contradiction to the definition of t in Condition (2). Therefore, our assumption $w_i < t$, for all $i \in \{1, \ldots, k\}$, was wrong and there is some $w_j \geq t$.

Project S_j to a line in direction $\boldsymbol{v}_j$ in $O(bd)$ time and use Lemma 1 to obtain a hyperplane $\mathcal{H}$ orthogonal to $\boldsymbol{v}_j$ that intersects at most $2b/(w_j - 2) \leq 2b/(t-2)$ balls from $\mathcal{D}$ in $O(b)$ time. By Lemma 1, the hyperplane $\mathcal{H}$ does not intersect any ball in $\mathcal{D}$ whose center is not in S_j, and so $\mathcal{H}$ is the desired separator.

Regarding the runtime bound: we compute the directions in $O(kd)$ time, the spread of a direction in $O(dn)$ time, which yields $O(kdn)$ time for k directions. The second step of finding $\mathcal{H}$ takes $O(bd) = O(nd)$ time. Therefore the overall runtime is $O(kdn)$.

3 A Deterministic Linear Time Algorithm in the Plane

In this section we describe a deterministic linear time algorithm to construct a halving line ℓ for a given set $\mathcal{D}$ of n disks in the plane. The line ℓ bisects $\mathcal{D}$ perfectly ($\leq n/2$ centers lie on either side) and it intersects $O(n^c)$ disks, where c may be chosen arbitrarily close to 5/6. We may assume that n is odd: If n is even, remove one arbitrary disk and observe that any halving line for the resulting set of disks is also a halving line for the original set. As our algorithm works in the dual arrangement, we first briefly review this duality and how it applies to line-disk intersections.

Point-line duality. The standard duality transform maps a point $p = (p_x, p_y)$ to the line $p^*\colon y = p_x x - p_y$ and a non-vertical line $g : y = mx + b$ to the point $g^* = (m, -b)$. This transformation is both incidence preserving ($p \in g \iff g^* \in p^*$) and order preserving (p is above $g \iff g^*$ is above p^*). Given a set P of points in the plane, the dual arrangement $\mathcal{A}(P^*)$ is defined by the lines in $P^* = \{p^* \mid p \in P\}$. In order to avoid parallel lines we assume that no two points in P have the same x-coordinate (which can be achieved by an infinitesimal rotation of the plane).

A halving line ℓ for P corresponds to a point ℓ^* in the dual arrangement that has no more than half of the lines from P^* above it and no more than half of the lines below it. The set of these points is referred to as the *median level* of the arrangement induced by P^*. Since n is odd, for any x-coordinate there is exactly one such point, and so we can regard the median level as a function from $\mathbb{R}$ to $\mathbb{R}$. The following lemma characterizes line-disk intersections in the dual plane.

Lemma 5. *Let $\ell : y = mx + b$ be a non-vertical line and let p denote the center of a unit disk D. Then D intersects ℓ if and only if the line p^* intersects the vertical segment $s = [(m, -b - \sqrt{m^2+1}), (m, -b + \sqrt{m^2+1})]$.*

If we view Lemma 5 from the perspective of a unit disk D with center p, then the set of lines that intersect D dualizes to the set of points (x, y) whose vertical distance to p^* is at most $\sqrt{1+x^2}$. We call this closed region of points the (dual) 1-*tube* of D. Note that the function $\sqrt{1+x^2}$ is strictly convex and so the 1-tube is bounded by a strictly convex function from above and by a strictly concave function from below.

Overview of the algorithm. The algorithm works in the dual arrangement and follows the prune and search paradigm. At the beginning we consider all potential halving lines, but subsequently narrow the range of potential slopes.

The successive narrowing of the range of slopes under consideration is made explicit by a parameter S, denoting the closed region bounded by at most two vertical lines. Such a region we call a *slab*. A slab $S = \{(x, y) \in \mathbb{R}^2 \colon \ell \leq x \leq r\}$ we denote by $S = <\ell, r>$. The distance $r - \ell$ between the two bounding vertical lines is the *width* of S. By Alon et al. [1] we may start with $S = <0, 1>$ as an initial slab, that is, there is always a halving line that intersects few disks and whose slope is between zero and one.

Crucial for the linear runtime bound is that a constant fraction of all lines from L be discarded after each iteration. However, by discarding some lines also our level of interest—which is the median level of the original set of lines—changes. Therefore this level also appears as a parameter of the algorithm. We denote this parameter by $\lambda \in \{1, 2, \ldots, |L|\}$. Initially $\lambda = \lceil n/2 \rceil$.

We first describe a single iteration of the algorithm, then prove some bounds for the parameters, and finally present the analysis of the whole algorithm.

A single iteration. At the beginning of each iteration we have a set L of n lines, a slab $S = <\ell, r>$ of width $w = r - \ell$, and a level parameter λ. Our goal is to find a constant fraction of lines from L that can be discarded. The outline of an iteration step is as follows.

1. Divide S in constantly many slabs $S_1, \ldots, S_m$, such that each contains at most $\alpha\binom{n}{2}$ many vertices of the arrangement $\mathcal{A}(L)$, for some appropriate constant $0 < \alpha < 1$. We define $S_i = <\ell_i, r_i>$ and $w_i = r_i - \ell_i$.
2. For each slab S_i, construct a *trapezoid* $T_i \subseteq S_i$ such that T_i contains the λ-level of $\mathcal{A}(L)$ within S_i and at most half of the lines from L intersect T_i.
3. For each trapezoid T_i, define its 1-*tube* $\tau_i \supset T_i$ as follows: Consider the two lines a_i and b_i passing through the segment bounding T_i from above and below, respectively; then τ_i is defined as the closed subset of S_i that is bounded by the upper boundary of the 1-tube of a_i from above and by the lower boundary of the 1-tube of b_i from below.
   For each slab S_i and some parameter $\gamma \in (0, 1/2)$, define the γ-*core* C_γ of S_i to be the central $(1-2\gamma)$-section of S_i, that is, $\mathrm{C}_\gamma(S_i) = <\ell_i + \gamma w_i, r_i - \gamma w_i>$.
   For each slab S_i, count the number n_i of lines that intersect τ_i within $\mathrm{C}_\gamma(S_i)$.
4. Select (in a way to be described) one of the slabs $\mathrm{C}_\gamma(S_i)$ to continue the search with. Discard all lines from L that do not intersect τ_i within $\mathrm{C}_\gamma(S_i)$ and adjust λ accordingly (decrease by #lines discarded that are below τ_i).

Observe first that discarding lines as described in Step 4 is justified: A line $\ell \in L$ that does not intersect τ_i within $\mathrm{C}_\gamma(S_i)$ by Lemma 5 corresponds to a unit disk centered at ℓ^* that within $\mathrm{C}_\gamma(S_i)$ is not intersected by any line whose dual point lies on the λ-level of $\mathcal{A}(L)$.

Next we will detail the steps listed above and analyze their runtime. For the first two steps we apply the machinery due to Lo et al. [9]. The first step can be handled in linear time using the following lemma, which follows from Lemma 3.3 of Lo et al. with $\alpha = 1/32$.

Lemma 6 ([9]). *Let L be a set of n lines in the plane in general position*[1] *and let S be a slab. In $O(n)$ time S can be subdivided into subslabs $S_1, S_2, \ldots, S_m \subset S$ (for some constant $m \leq 64$), such that each S_i contains at most $\frac{1}{32}\binom{n}{2}$ of the $\binom{n}{2}$ vertices of $\mathcal{A}(L)$.*

The trapezoids mentioned in the second step can be computed as follows. For $S_i = {<}\ell_i, r_i{>}$, let the upper left (right) corner of T_i be defined by the $(\lambda + n/8)$-level of $\mathcal{A}(L)$ at $x = \ell_i$ ($x = r_i$). Analogously, the lower corners of T_i are defined by the $(\lambda - n/8)$-level of $\mathcal{A}(L)$ at $x = \ell_i$ ($x = r_i$). Then Lemma 3.5 from the paper by Lo et al. (with $\delta = 1/8$) gives the following:

Lemma 7 ([9]). *The trapezoid T_i contains the λ-level of $\mathcal{A}(L)$ within S_i and at most half of the lines from L intersect T_i.*

All these trapezoids can be constructed in a brute-force manner in $O(n)$ time (recall that m is constant). This completes the first two steps: we have computed (in linear time) a subdivision of our initial slab S into $m \leq 64$ subslabs S_i, each of which contains a trapezoid T_i that contains the λ-level of $\mathcal{A}(L)$ within S_i and at most half of the lines from L intersect T_i.

Regarding Step 3, note that testing whether a given line intersects τ_i is a geometric predicate of constant algebraic degree. Hence this step can be executed in a straightforward manner in $O(mn) = O(n)$ time. It remains to argue how to select an appropriate slab to continue with in Step 4. It turns out that not only the number of lines matters, but it is also important to ensure that the width of the slab does not become too small. The following lemma gives a precise account for the bounds we are after.

Lemma 8. *For any $0 < \varepsilon < 1/2$ and $0 < \gamma < 1/2$ there exist an integer $n' > 0$ and constants $m \leq 64$ and $c = (8m/\gamma\varepsilon)^2$ such that for any $n \geq n'$ the following statement holds. Given a set L of n lines, an integer $\lambda \in \{1, \ldots, n\}$, and a slab $S \subseteq {<}0, 1{>}$ of width $w \geq c\log(n)/n$, there exist a set $L' \subset L$ of at most $(\frac{1}{2} + \varepsilon)n$ lines and a slab S' of width $\geq (1 - 2\gamma)w/m$ such that inside S' the λ-level of $\mathcal{A}(L)$ does not intersect any line in $L \setminus L'$.*

Analysis of the algorithm. Let us postpone the proof of Lemma 8 for now and first complete the overall analysis of the algorithm. Denote by n_t the number of lines and by w_t the width of the current slab after t iterations. We have $n_0 = n$ and $w_0 = 1$. By Lemma 8 we have $n_t \leq \left(\frac{1}{2} + \varepsilon\right)^t n$ and $w_t \geq ((1 - 2\gamma)/m)^t$, as long as $w \geq c\log(n)/n$. After some number of iterations, either we are left with a constant number of lines or a slab of width $w < c\log(n)/n$. As in the first case we can finish by brute force, let us concentrate on the second case. Suppose t^*

[1] Any two intersect in exactly one point.

is the smallest index for which $w_{t^*} < c\log(n)/n$. The following inequalities are equivalent:

$$\left(\frac{1-2\gamma}{m}\right)^{t^*} < \left(\frac{8m}{\gamma\varepsilon}\right)^2 \cdot \frac{\log n}{n}$$
$$-t^* \log\left(\frac{m}{1-2\gamma}\right) < 2\log\left(\frac{8m}{\gamma\varepsilon}\right) + \log\log n - \log n$$
$$t^* > \frac{\log n - 2\log\left(\frac{8m}{\gamma\varepsilon}\right) - \log\log n}{\log\left(\frac{m}{1-2\gamma}\right)}.$$

Since γ, ε and m are all constant, the last inequality implies that for any constant $0 < \varepsilon' < 1$ we have

$$t^* > \log n \cdot \frac{(1-\varepsilon')}{\log(\frac{m}{1-2\gamma})},$$

for sufficiently large n (depending on ε'). Hence the number of lines to be considered after t^* iterations is

$$n_{t^*} \le \left(\frac{1}{2}+\varepsilon\right)^{t^*} \cdot n < \left(\frac{1}{2}+\varepsilon\right)^{\log n \frac{1-\varepsilon'}{\log(\frac{m}{1-2\gamma})}} \cdot n = n^{\log\left(\frac{1}{2}+\varepsilon\right)\frac{1-\varepsilon'}{\log(\frac{m}{1-2\gamma})}+1} \le n^{\frac{5}{6}+\delta},$$

where the last inequality uses $\log m \le 6$ and where $\delta > 0$ can be made arbitrarily small by choosing ε, ε' and γ to be correspondingly small.

So after at most $t^* = \Theta(\log n)$ iterations we are left with a slab S and $O(n^{\frac{5}{6}+\delta})$ lines. All lines that have been discarded do not intersect the 1-tube of the level that corresponds to the original median level. Therefore any point on this level within S corresponds to a halving line for the original set of disks that intersects $o(n)$ of the disks. Such a point can easily be found in a brute force manner in $o(n)$ time.

Denote by $R(n)$ the runtime of the algorithm for n disks. Each iteration can be handled in time linear in the number of lines/disks remaining and so

$$R(n) \le \sum_{t=0}^{t^*} cn_t \le cn\sum_{t=0}^{t^*}\left(\frac{1}{2}+\varepsilon\right)^t < \frac{2c}{1-2\varepsilon}n = O(n),$$

for some constant c. This proves Theorem 3.

Proof of Lemma 8. It remains to prove that we can select a constant fraction of lines to be discarded in each iteration while at the same time the width of the current slab does not shrink too much. To begin with we need a slab whose 1-tube is not intersected by too many lines. To show that such a slab exists, we use an averaging argument: While a single 1-tube τ_i may be intersected by all lines from L, on average the number of intersecting lines per slab is sublinear.

To this end we define a function g by setting $g(x)$ to be the number of lines that intersect $\bigcup_{i=1}^{m} \tau_i$ at $x \in (\ell, r)$. The following lemma provides an upper bound on the average number of such lines.

Lemma 9. *For a slab $S = <\ell, r> \subseteq <0, 1>$ of width $w = r - \ell$, there is some constant $c \leq 4$ such that*

$$\int_{\ell}^{r} g(x)\,\mathrm{d}x \leq c\sqrt{nw\log(nw)}\,,$$

if nw is sufficiently large.

By the pigeonhole principle, the integral is small for most subslabs. But bounding the integral is not sufficient to bound the number of lines that intersect the 1-tube, because lines that do so for a very short interval only do not contribute much to the integral. To account for such lines we restrict our focus to the γ-core of the slabs instead. For a slab S_i let $d_i(x)$ denote the number of lines that intersect $\tau_i \setminus T_i$ at x, for $x \in (\ell_i, r_i)$. Clearly $d_i \leq g$. Furthermore let $\phi_{\gamma,i} = \max\{d_i(x)\colon x \times \mathbb{R} \subset \mathrm{C}_\gamma(S_i)\}$.

Proposition 1. *The number of lines from L that intersect $(\tau_i \setminus T_i) \cap \mathrm{C}_\gamma(S_i)$ is bounded by $2\phi_{\gamma,i}$, for any $i \in \{1, \ldots, m\}$ and $0 < \gamma < 1/2$.*

Proposition 2. $\phi_{\gamma,i} w_i \leq \gamma^{-1} \int_{\ell_i}^{r_i} g(x)\,\mathrm{d}x$.

Now we have all tools in place to complete the proof of Lemma 8. Combining Proposition 2 and Lemma 9 yields $\sum_{i=1}^{m} \phi_{\gamma,i} w_i \leq 4\gamma^{-1}\sqrt{nw\log(nw)}$.

We claim that we can select any slab S_j for which $w_j \geq w/m$ and continue the search within $\mathrm{C}_\gamma(S_j)$. Such a slab exists because there are m slabs in total and $w = \sum_{i=1}^{m} w_i$. We can then bound

$$\phi_{\gamma,j}\frac{w}{m} \leq \phi_{\gamma,j} w_j \leq \sum_{i=1}^{m} \phi_{\gamma,i} w_i \leq 4\gamma^{-1}\sqrt{nw\log(nw)}$$

and so

$$\phi_{\gamma,j} \leq 4\gamma^{-1} m \sqrt{\frac{n\log(nw)}{w}} \leq 4\gamma^{-1} m \sqrt{\frac{n\log(n)}{w}}\,.$$

The slab we continue to search in (the core $\mathrm{C}_\gamma(S_j)$ of S_j) has width at least $(1-2\gamma)w/m$. Lemma 7 and Proposition 1 bound the number $n_{\gamma,j}$ of lines that intersect τ_j within $\mathrm{C}_\gamma(S_j)$ by

$$n_{\gamma,j} \leq \frac{n}{2} + 2\phi_{\gamma,j} \leq \frac{n}{2} + \frac{8m}{\gamma}\sqrt{\frac{n\log(n)}{w}}\,.$$

Given any $0 < \varepsilon < 1/2$ and $0 < \gamma < 1/2$, we have $n_{\gamma,j} \leq \left(\frac{1}{2} + \varepsilon\right) n$, as long as $w \geq \left(\frac{8m}{\gamma\varepsilon}\right)^2 \cdot \frac{\log n}{n}$, which is stated as an assumption. □

4 Conclusions

We studied the construction of separators for balls in *deterministic* linear time. The aim is to intersect as few balls as possible while (approximately) bisecting the set of center points. We presented essentially two ways to compute such seperators. The first algorithm is simple and obtains an arbitrarily good bisection in combination with an asymptotically optimal number of intersections. The second algorithm bisects the center points *exactly*, but works in the plane only.

Throughout the paper we assumed the balls to be disjoint, but for our algorithms it is enough to have some density lower bound on the objects under consideration and some bound on the size of the objects (e.g. disjoint fat objects). Also note that, in contrast to the continuous case, we do not make use of the fact that the hyperplane to be constructed is bisecting. Therefore it is easy to adapt the algorithm to, for instance, have $n/3$ of the points on one side and $2n/3$ of the other side of the hyperplane.

There are point sets for which the number of balls intersected by *every* halving hyperplane is $\Omega(n^{(d-1)/d})$. But already for dimension three it is not clear if a halving plane with $o(n^{3/4})$ intersections always exists ($O(n^{3/4})$ is not difficult). In dimension two it is open if $o(\sqrt{n \log n})$ can be achieved. So let us ask the following question: Is it true that for every set of n disjoint unit balls in $\mathbb{R}^d$ there exists a halving hyperplane that intersects $O(n^{(d-1)/d})$ of the balls?

Acknowledgments. We want to thank Marek Elias, Jiřka Matoušek, Edgardo Roldán-Pensado and Zuzana Safernová for interesting discussions on the conjecture for higher dimensions and referring us to related work. We thank the anonymous reviewers for their constructive comments and suggestions.

References

1. Alon, N., Katchalski, M., Pulleyblank, W.R.: Cutting disjoint disks by straight lines. Discrete & Computational Geometry 4, 239–243 (1989)
2. Atkin, A.O.L., Bernstein, D.J.: Prime sieves using binary quadratic forms. Math. Comput. 73(246), 1023–1030 (2004)
3. Barequet, G.: A lower bound for Heilbronn's triangle problem in d dimensions. SIAM Journal on Discrete Mathematics 14(2), 230–236 (2001)
4. Bereg, S., Dumitrescu, A., Pach, J.: Sliding disks in the plane. International Journal of Computational Geometry & Applications 18(05), 373–387 (2008)
5. Blum, M., Floyd, R.W., Pratt, V., Rivest, R.L., Tarjan, R.E.: Time bounds for selection. Journal of Computer and System Sciences 7(4), 448–461 (1973)
6. Esposito, L., Ferone, V., Kawohl, B., Nitsch, C., Trombetti, C.: The longest shortest fence and sharp Poincaré–sobolev inequalities. Archive for Rational Mechanics and Analysis 206(3), 821–851 (2012)
7. Held, M., Mitchell, J.S.: Triangulating input-constrained planar point sets. Information Processing Letters 109(1), 54–56 (2008)
8. Lefmann, H.: On Heilbronn's problem in higher dimension. Combinatorica 23(4), 669–680 (2003)

9. Lo, C.-Y., Matoušek, J., Steiger, W.L.: Algorithms for ham-sandwich cuts. Discrete & Computational Geometry 11, 433–452 (1994)
10. Löffler, M., Mulzer, W.: Unions of onions: preprocessing imprecise points for fast onion layer decomposition. In: Algorithms and Data Structures, pp. 487–498. Springer, Heidelberg (2013)
11. Martini, H., Schöbel, A.: Median hyperplanes in normed spaces – a survey. Discrete Applied Mathematics 89(1), 181–195 (1998)
12. Matoušek, J.: Efficient partition trees. Discrete & Computational Geometry 8(1), 315–334 (1992)
13. Pach, J., Sharir, M.: Combinatorial geometry and its algorithmic applications: The Alcalá lectures. Mathematical Surveys and Monographs, vol. 152. Amer. Math. Soc. (2009)
14. Roth, K.F.: On a problem of Heilbronn. J. London Math. Soc. 26(3), 198–204 (1951)
15. Tverberg, H.: A seperation property of plane convex sets. Mathematica Scandinavica 45, 255–260 (1979)

Turing Kernelization for Finding Long Paths and Cycles in Restricted Graph Classes*

Bart M.P. Jansen

University of Bergen, Norway
Bart.Jansen@ii.uib.no

Abstract. We analyze the potential for provably effective preprocessing for the problems of finding paths and cycles with at least k edges. Several years ago, the question was raised whether the existing superpolynomial kernelization lower bounds for k-Path and k-Cycle can be circumvented by relaxing the requirement that the preprocessing algorithm outputs a single instance. To this date, very few examples are known where the relaxation to *Turing kernelization* is fruitful. We provide a novel example by giving polynomial-size Turing kernels for k-Path and k-Cycle on planar graphs, graphs of maximum degree t, claw-free graphs, and $K_{3,t}$-minor-free graphs, for each constant $t \geq 3$. The result for planar graphs solves an open problem posed by Lokshtanov. Our kernelization schemes are based on a new methodology called *Decompose-Query-Reduce.*

1 Introduction

Motivation. Kernelization is a formalization of efficient and provably effective data reduction originating from parameterized complexity theory. In this setting, each instance $x \in \Sigma^*$ of a decision problem is associated with a parameter $k \in \mathbb{N}$ that measures some aspect of its complexity. Work on kernelization over the last few years has resulted in deep insights into the possibility of reducing an instance (x, k) of a parameterized problem to an equivalent instance (x', k') of size polynomial in k, in polynomial time. By now, many results are known concerning problems that admit such *polynomial kernelization algorithms*, versus problems for which the existence of a polynomial kernel is unlikely because it implies the complexity-theoretic collapse NP $\subseteq$ coNP/poly. (See Section 2 for formal definitions of parameterized complexity.)

In this work we study the potential of effectively preprocessing instances of the problems of finding long paths or cycles in a graph. In the model of (Karp) kernelization described above, in which the output of the preprocessing algorithm is a single, small instance, we cannot guarantee effective polynomial-time preprocessing for these problems. Indeed, the k-Path (is there a path of *at least* k edges) and k-Cycle (is there a cycle of *at least* k edges) problems are *or-compositional* [6] since the disjoint union of graphs $G_1, \ldots, G_t$ contains a path

* This work was supported by the European Research Council through Starting Grant 306992 "Parameterized Approximation".

(cycle) of length k if and only if there is at least one input graph with such a structure. Using the framework of Bodlaender et al. [6] this proves that k-Path and k-Cycle do not admit Karp kernelizations of polynomial size unless NP $\subseteq$ coNP/poly and the polynomial hierarchy collapses to its third level.

More than five years ago, the question was raised how fragile this *bad news* is: what happens if we relax the requirement that the preprocessing algorithm outputs a single instance? Does a polynomial-time preprocessing algorithm exist that, given an instance (G, k) of k-Path, builds a list of instances $(x_1, k_1), \ldots, (x_t, k_t)$, each of size polynomial in k, such that G has a length-k path if and only if there is at least one yes-instance among the outputs? Such a *cheating kernelization* is possible for the k-Leaf Out-Tree problem [3] while it does not admit a polynomial Karp kernelization unless NP $\subseteq$ coNP/poly. Hence it is natural to ask whether this can be done for k-Path or k-Cycle.

A robust definition of such relaxed forms of preprocessing was given by Lokshtanov [19] under the name *Turing kernelization.* It is phrased in terms of algorithms that can query an oracle for the answer to instances of specific decision problem in a single computation step.[1] Observe that the existence of an $f(k)$-size kernel for a parameterized problem $\mathcal{Q}$ shows that $\mathcal{Q}$ can be solved in polynomial time if we allow the algorithm to make a single size-$f(k)$ query to an oracle for $\mathcal{Q}$: apply the kernelization to input (x, k) to obtain an equivalent instance (x', k') of size $f(k)$, query the $\mathcal{Q}$-oracle for this instance and output its answer. A natural relaxation, which encompasses the *cheating kernelization* mentioned above, is to allow the polynomial-time algorithm to query the oracle more than once for the answers to $f(k)$-size instances. This motivates the definition of Turing kernelization.

Definition 1. *Let $\mathcal{Q}$ be a parameterized problem and let $f\colon \mathbb{N} \to \mathbb{N}$ be a computable function. A* Turing kernelization for $\mathcal{Q}$ of size f *is an algorithm that decides whether a given instance $(x, k) \in \Sigma^* \times \mathbb{N}$ is contained in $\mathcal{Q}$ in time polynomial in $|x| + k$, when given access to an oracle that decides membership in $\mathcal{Q}$ for any instance (x', k') with $|x'|, k' \leq f(k)$ in a single step.*

For practical purposes the role of oracle is fulfilled by an external computing cluster that computes the answers to the queries. A Turing kernelization gives the means of efficiently splitting the work on a large input into manageable chunks, which may be solvable in parallel depending on the nature of the Turing kernelization. Moreover, Turing kernelization is a natural relaxation of Karp kernelization that facilitates a theoretical analysis of the nature of preprocessing.

At first glance, it seems significantly easier to develop a Turing kernelization than a Karp kernelization. However, to this date there are only a handful of parameterized problems known for which polynomial-size Turing kernelization is possible but polynomial-size Karp kernelization is unlikely [1,8,22,23].

[1] Formally, such algorithms are *oracle Turing machines* (cf. [13, Appendix A.1]).

Recently, the first *adaptive*[2] Turing kernelization was given by Thomassé et al. [23] for the k-Independent Set problem restricted to bull-free graphs. Although this forms an interesting step forwards in harnessing the power of Turing kernelization, the existence of polynomial-size Turing kernels for k-Path and related subgraph-containment problems remains a major open problem in the area of kernelization [3,5,14]. Since many graph problems that are intractable in general admit polynomial-size (Karp) kernels when restricted to planar graphs, it is natural to consider whether such a restriction makes it easier to obtain polynomial Turing kernels for k-Path. Consequently, Lokshtanov [19] and Misra et al. [20] posed the existence of a polynomial Turing kernel for k-Path on planar graphs as an open problem. Observe that, by the or-composition argument given above, planar k-Path does not have a polynomial-size (Karp) kernel unless NP $\subseteq$ coNP/poly.

Our Results. In this paper we introduce the *Decompose-Query-Reduce* framework for obtaining adaptive polynomial-size Turing kernelizations for the k-Path and k-Cycle problems on various restricted graph families, including planar graphs and bounded-degree graphs. The three steps of the framework consist of (i) decomposing the input (G, k) into parts of size $k^{\mathcal{O}(1)}$ with constant-size interfaces between the various parts; (ii) querying the oracle to determine how a solution can intersect such bounded-size parts, and (iii) reducing to an equivalent but smaller instance using this information. In our case, we use a classic result by Tutte [24] concerning the decomposition of a graph into its triconnected components, made algorithmic by Hopcroft and Tarjan [15], to find a tree decomposition of adhesion two of the input graph G such that all torsos of the decomposition are triconnected minors of G. We complement this with various known graph-theoretic lower bounds on the circumference of triconnected graphs belonging to restricted graph families to deduce that if this Tutte decomposition has a bag of size $\Omega(k^{\mathcal{O}(1)})$, then there must be a cycle (and therefore path) of length at least k in G. If we have not already found the answer to the problem we may therefore assume that all bags of the decomposition have polynomial size. Consequently we may query the oracle for solutions involving only $k^{\mathcal{O}(1)}$ parts of the decomposition. Observe that if G is a graph and $S \subseteq V(G)$, then a simple path in G cannot visit more than $|S|+1$ different components of $G-S$. Hence for each node of the Tutte decomposition, which represents a bag of size $k^{\mathcal{O}(1)}$, at most $k^{\mathcal{O}(1)}$ of the triconnected components represented by its children are used in a solution. Using ideas from earlier work [7], this allows us to invoke the oracle to instances of size $k^{\mathcal{O}(1)}$ to obtain the information that is needed to safely discard some pieces of the input, thereby shrinking it. Iterating this procedure, we arrive at a final equivalent instance of size $k^{\mathcal{O}(1)}$, whose answer is queried from the oracle and given as the output of the Turing kernelization. In this way we obtain polynomial Turing kernels for k-Path and the related k-Cycle problem

[2] The algorithm is adaptive because it uses the answers to earlier oracle queries to formulate its next query. In contrast, the cheating kernelization for k-Leaf Out-Tree constructs all its queries without having to know a single answer. In the language of classical computability theory [21], the adaptive algorithm is a Turing reduction whereas the cheating kernelization is a truth-table reduction.

in planar graphs, graphs that exclude $K_{3,t}$ as a minor for some $t \geq 3$, graphs of maximum degree bounded by $t \geq 3$, and claw-free graphs. We remark that the k-Path and k-Cycle problems remain NP-complete in all these cases [18].

Our results raise a number of interesting challenges and shed some light on the possibility of polynomial Turing kernelization for the unrestricted k-Path problem. A completeness program for classifying Turing kernelization complexity was recently introduced by Hermelin et al. [14]. They proved that a *colored* variant of the k-Path problem is complete for a class called WK[1] and conjectured that WK[1]-hard problems do not admit polynomial Turing kernels. We give evidence that the classification of the colored variant may have little to do with the kernelization complexity of the base problem: Multicolored k-Path remains WK[1]-hard on bounded-degree graphs, while our framework yields a polynomial Turing kernel for uncolored k-Path in this case.

Related Work. Non-adaptive Turing kernels of polynomial size are known for k-Leaf Out-Tree [3], k-Colorful Motif on comb graphs [1], and s-Club [22] (see also [14]). Trotignon et al. [23] gave an adaptive Turing kernel of polynomial size for k-Independent Set on bull-free graphs. Weller [25, Chapter 5] discusses various variants of Turing kernelization.

Organization. In Section 2 we give preliminaries on parameterized complexity and graph theory. In Section 3 we present Turing kernels for the k-Cycle problem. These are technically somewhat less involved than the analogues for k-Path that are described in Section 4. In Section 5 we briefly consider Multicolored k-Path. Due to length restrictions, the proofs of statements marked by a star (★) have been deferred to the full version [17].

2 Preliminaries

Parameterized Complexity and Kernels. The set $\{1, 2, \ldots, n\}$ is abbreviated as $[n]$. For a set X and non-negative integer n we use $\binom{X}{n}$ to denote the collection of size-n subsets of X. A parameterized problem $\mathcal{Q}$ is a subset of $\Sigma^* \times \mathbb{N}$, the second component of a tuple $(x, k) \in \Sigma^* \times \mathbb{N}$ is called the *parameter*. A parameterized problem is (strongly uniformly) *fixed-parameter tractable* if there exists an algorithm to decide whether $(x, k) \in \mathcal{Q}$ in time $f(k)|x|^{\mathcal{O}(1)}$ where f is a computable function. A *Karp kernelization algorithm* (or *Karp kernel*) of size $f \colon \mathbb{N} \to \mathbb{N}$ for a parameterized problem $\mathcal{Q} \subseteq \Sigma^* \times \mathbb{N}$ is a polynomial-time algorithm that, on input $(x, k) \in \Sigma^* \times \mathbb{N}$, outputs an instance (x', k') with $|x'|, k' \leq f(k)$ such that $(x, k) \in \mathcal{Q} \Leftrightarrow (x', k') \in \mathcal{Q}$. If $f(k) \in \mathcal{O}(k^{\mathcal{O}(1)})$ then this is a *polynomial kernel* (cf. [4]). We refer to a textbook [13] for more background on parameterized complexity.

Graphs and Tree Decompositions. All graphs we consider are finite, simple, and undirected. An undirected graph G consists of a vertex set $V(G)$ and an edge set $E(G) \subseteq \binom{V(G)}{2}$. We write $G \subseteq H$ if graph G is a subgraph of graph H. The subgraph of G induced by a set $X \subseteq V(G)$ is denoted $G[X]$. We use $G - X$

as a shorthand for $G[V(G) \setminus X]$. When deleting a single vertex v, we write $G-v$ rather than $G-\{v\}$. The *open neighborhood* of a vertex v in graph G is $N_G(v)$. The open neighborhood of a set $X \subseteq V(G)$ is $\bigcup_{v \in X} N_G(v) \setminus X$. Graph H is a *minor* of graph G if H can be obtained from a subgraph of G by edge contractions. A *cut vertex* in a connected graph G is a vertex v such that $G-v$ is disconnected. A vertex is a cutvertex in a disconnected graph if it forms such a structure for a connected component. A graph G is *biconnected* if it is connected and contains no cut vertices. The *biconnected components* of G partition the edges of G into biconnected subgraphs of G. A graph G is triconnected if removing at most three vertices from G cannot result in a disconnected graph.[3] A *walk* in G is a sequence of vertices $v_1, \ldots, v_k$ such that $\{v_i, v_{i+1}\} \in E(G)$ for $i \in [k-1]$. An xy-walk is a walk with $v_1 = x$ and $v_k = y$. A *path* is a walk in which all vertices are distinct. Similarly, an xy-path is an xy-walk consisting of distinct vertices. The vertices x and y are the *endpoints* of an xy-path. The *length* of a path $v_1, \ldots, v_k$ is the number of edges on it: $k-1$. A *cycle* is a sequence of three or more vertices $v_1, \ldots, v_k$ that forms a $v_1 v_k$-path such that, additionally, the edge $\{v_1, v_k\}$ is contained in G. The *length* of a cycle is the number of edges on it: k. For an integer k, a k-cycle in a graph is a cycle with at least k edges; similarly a k-path is a path with at least k edges. The *claw* is the complete bipartite graph $K_{1,3}$ with partite sets of size one and three. A graph is *claw-free* if it does not contain the claw as an induced subgraph. When analyzing the running time of graph algorithms, we use n to denote the number of vertices and m to denote the number of edges.

Proposition 1 (★). *If the graph G contains a cycle (path) of length at least k as a minor, then it contains a cycle (path) of length at least k as a subgraph.*

Definition 2. *A* tree decomposition *of a graph G is a pair $(T, \mathcal{X})$, where T is a tree and $\mathcal{X}\colon V(T) \to 2^{V(G)}$ assigns to every node of T a subset of $V(G)$ called a* bag*, such that: (a) $\bigcup_{i \in V(T)} \mathcal{X}(i) = V(G)$, (b) for each edge $\{u, v\} \in E(G)$ there is a node $i \in V(T)$ with $\{u, v\} \subseteq \mathcal{X}(i)$, and (c) for each $v \in V(G)$ the nodes $\{i \mid v \in \mathcal{X}(i)\}$ induce a connected subtree of T.*

The *width* of the tree decomposition is $\max_{i \in V(T)} |\mathcal{X}(i)| - 1$. The *adhesion* of a tree decomposition is $\max_{\{i,j\} \in E(T)} |\mathcal{X}(i) \cap \mathcal{X}(j)|$. If T has no edges, we define the adhesion to be zero. If $(T, \mathcal{X})$ is a tree decomposition of a graph G, then the *torso* of a bag $\mathcal{X}(i)$ for $i \in V(T)$ is the graph $\textsc{torso}(G, \mathcal{X}(i))$ obtained from $G[\mathcal{X}(i)]$ by adding an edge between each pair of vertices in $\mathcal{X}(i)$ that are connected by a path in G whose internal vertices do not belong to $\mathcal{X}(i)$.

Tutte Decomposition. The following theorem is originally due to Tutte, but has been reformulated in the language of tree decompositions. The full paper [17] contains a proof for completeness. Recall that a *minimal separator* in a connected graph G is a vertex set $S \subseteq V(G)$ such that $G-S$ is disconnected and $G-S'$

[3] Some authors require a triconnected graph to contain more than three vertices; the present definition allows us to omit some case distinctions.

is connected for all $S' \subsetneq S$. A vertex set of a disconnected graph is a minimal separator if it is a minimal separator for one of the connected components.

Theorem 1 ([24], see [12, Exercise 12.20]). *For every graph G there is a tree decomposition $(T, \mathcal{X})$ of adhesion at most two, called a* Tutte decomposition, *such that (i) for each node $i \in V(T)$, the graph* $\textsc{torso}(G, \mathcal{X}(i))$ *is a triconnected minor of G, and (ii) for each edge $\{i, j\}$ of T the set $\mathcal{X}(i) \cap \mathcal{X}(j)$ is a minimal separator in G or the empty set.*

Circumference of Restricted Classes of Triconnected Graphs. The *circumference* of a graph is the length of a longest cycle. Several results are known that give a lower bound on the circumference of a triconnected graph in terms of its order. We will use these lower bounds to deduce that if a Tutte decomposition of a graph has large width, then the graph contains a long cycle (and therefore also a long path).

Theorem 2. *Let G be a triconnected graph on n vertices and let ℓ be its circumference.*

(a) If G is planar, then $\ell \geq n^{\log_3 2}$. [10]
(b) If G is $K_{3,t}$-minor free, then $\ell \geq (1/2)^{t(t-1)} n^{\log_{1729} 2}$. [11]
(c) If G is claw-free, then $\ell \geq (n/12)^{0.753} + 2$. [2]
(d) If G has maximum degree at most $\Delta \geq 4$, then $\ell \geq n^{\log_r 2}/2 + 3$, where $r := \max(64, 4\Delta + 1)$. [9].

3 Turing Kernelization for Finding Cycles

In this section we show how to obtain polynomial Turing kernels for k-Cycle on various restricted graph families. After discussing some properties of cycles in Section 3.1, we start with the planar case in Section 3.2. In Section 3.3 we show how to adapt the strategy for $K_{3,t}$-minor-free, claw-free, and bounded-degree graphs.

3.1 Properties of Cycles

We present several properties of cycles that will be used in the Turing kernelization. Recall that a k-cycle is a cycle with *at least* k edges.

A *separation* of a graph G is a pair (A, B) of subsets of $V(G)$ such that $A \cup B = V(G)$ and G has no edges between $A \setminus B$ and $B \setminus A$. The latter implies that $A \cap B$ separates the vertices $A \setminus B$ from the vertices $B \setminus A$. The *order* of the separation is $|A \cap B|$. The following lemma shows that, after testing one side of an order-two separation for having a k-cycle, we may safely remove vertices from that side as long as we preserve a maximum-length path connecting the two vertices in the separator.

Lemma 1 (★). *Let $A, B \subseteq V(G)$ be a separation of order two of a graph G with $A \cap B = \{x, y\}$. Let $V(\mathcal{P}_A)$ be the vertices on a maximum-length xy-path $\mathcal{P}_A$ in $G[A]$, or $\emptyset$ if no such path exists. If G has a k-cycle, then $G[A]$ has a k-cycle or $G[V(\mathcal{P}_A) \cup B]$ has a k-cycle.*

We show how to use an oracle for the *decision* version of k-Cycle to *construct* longest xy-paths by self-reduction. These paths can be used with the previous lemma to find vertices that can be removed from the graph while preserving a k-cycle, if one exists. We will invoke Lemma 2 only for graphs with $n \in k^{\mathcal{O}(1)}$.

Lemma 2 (★). *There is an algorithm that, given a graph G with distinct vertices x and y, and an integer k, either (i) determines that G contains a k-cycle, (ii) determines that G contains an xy-path of length at least k, or (iii) outputs the (unordered) vertex set of a maximum-length xy-path in G (or $\emptyset$ if no such path exists). The algorithm runs in $\mathcal{O}((n+k)(n+m+k))$ time when given access to an oracle that decides the k-Cycle problem. The oracle is queried for instances (G', k) with $|V(G')| \leq n + k$, where G' is obtained from an induced subgraph of G by adding an xy-path that is either a single edge or consists of new vertices of degree two.*

When the self-reduction algorithm detects a long xy-path for a minimal separator $\{x, y\}$, the following proposition proves that there is in fact a long cycle.

Proposition 2 (★). *If $\{x, y\}$ is a minimal separator of a graph G and G contains an xy-path of length $k \geq 2$, then G contains a $(k+1)$-cycle.*

3.2 k-Cycle in Planar Graphs

Before presenting the Turing kernelization algorithm we sketch the main ideas. Suppose that the input (G, k) contains a separation $A, B \subseteq V(G)$ of order two such that $k < |A| < k^{\mathcal{O}(1)}$, and the separator $\{x, y\} := A \cap B$ is minimal. As the size of $G[A]$ is polynomial in the parameter, we can invoke the oracle to determine the existence of a k-cycle in $G[A]$. If such a cycle exists, then we are done. If not, then using Lemma 2 we can find the vertices on a maximum-length xy-path $\mathcal{P}_A$ in $G[A]$. If the path has at least k vertices (implying the length is at least $k-1$), then by Proposition 2 there is a k-cycle in G and again we are done. Otherwise, by Lemma 1, we can safely remove the vertices of $G[A] \setminus V(\mathcal{P}_A)$ from G. Since $k < |A|$ we discard at least one vertex in this way. Hence as long as a minimal order-two separation with $k < |A| < k^{\mathcal{O}(1)}$ exists, we can reduce to a smaller equivalent instance after querying the oracle for polynomial-size subgraphs. The main insight is that for biconnected planar graphs, either the graph contains a k-cycle or such a separation exists. The proof given below exploits this insight implicitly in its recursive procedure to solve the k-Cycle problem.

Theorem 3. *The planar k-Cycle problem has a polynomial Turing kernel: it can be solved in polynomial-time using an oracle that decides planar k-Cycle instances with at most $(3k+1)k^{\log_2 3} + k$ vertices and parameter value k.*

Proof. We present the Turing kernel for k-Cycle on planar graphs following the three steps of the kernelization framework.

Decompose. Consider an input (G, k) of planar k-Cycle. First observe that a cycle in G is contained within a single biconnected component of G. We may therefore compute the biconnected components of G in linear time using the algorithm by Hopcroft and Tarjan [16] and work on each biconnected component separately. In the remainder we therefore assume that the input graph G is biconnected. By another algorithm of Hopcroft and Tarjan [15] we can compute a Tutte decomposition $(T, \mathcal{X})$ of G in linear time. For each edge $\{i, j\} \in E(T)$ of the decomposition tree, the definition of a Tutte decomposition ensures that $\mathcal{X}(i) \cap \mathcal{X}(j)$ is a minimal separator in G. Since T has adhesion at most two by Theorem 1, these minimal separators have size at most two. Using the biconnectedness of G it follows that the intersection of the bags of adjacent nodes in T has size exactly two. Since each torso of the decomposition is a triconnected minor of G, and therefore planar, the following claim follows by combining Theorem 1, Proposition 1, and Theorem 2.

Claim 1 (★). *If there is a node $i \in V(T)$ of the Tutte decomposition such that $|\mathcal{X}(i)| \geq k^{\log_2 3}$, then G has a k-cycle.*

The claim shows that we may safely output YES if the width of $(T, \mathcal{X})$ exceeds $k^{\log_2 3}$. For the remainder of the kernelization we may therefore assume that $(T, \mathcal{X})$ has width at most $k^{\log_2 3}$. We start the reduction phase by making a copy G' of G and a copy $(T', \mathcal{X}')$ of the decomposition. During the reduction phase we will repeatedly remove vertices from the graph G' to reduce its size. Removing these vertices from the bags of the decomposition $(T', \mathcal{X}')$, we may violate the property of a Tutte decomposition that all torsos of bags are triconnected. However, we will maintain the fact that $(T', \mathcal{X}')$ is a tree decomposition of adhesion at most two and width at most $k^{\log_2 3}$ of G'. We root the decomposition tree T' at an arbitrary vertex to complete the decomposition phase. We use the following terminology. For $i \in V(T')$ we write $T'[i]$ for the subtree of T' rooted at i. For a subtree $T'' \subseteq T'$ we write $\mathcal{X}'(T'')$ for the union $\bigcup_{i \in V(T'')} \mathcal{X}'(i)$ of the bags of the nodes in T''. For a node i in the rooted tree T' we write $N^+_{T'}(i)$ to denote the children (the out-neighbors) of node i.

Query and reduce. In the next phase we repeatedly query the k-Cycle oracle to reduce the size of G' while preserving a k-cycle, if one exists. At any point in the process we may find a k-cycle and halt. The procedure is given as Algorithm 1. It is initially called for the root node r of T'. Intuitively, Algorithm 1 processes the decomposition tree T' bottom-up, applying Lemma 1 to justify two types of size reductions for a node $i \in V(T')$. During the first **foreach** loop the sizes $|\mathcal{X}'(T'[j])|$ of the subtrees $T'[j]$ rooted at children j of i are reduced, by removing vertices that are avoided by some maximum-length xy-path. The second **foreach** loop applies the same lemma to reduce the number of children of i by effectively deleting connected components C of $G'[A] - \{x, y\}$ when a maximum-length xy-path can be obtained through a different component $C' \neq C$.

Algorithm 1. QueryReduceCycle($G', (T', \mathcal{X}'), i, k$)

Precondition: G' is an induced subgraph of G with a tree decomposition $(T', \mathcal{X}')$ of adhesion at most two. A node i of T' is specified.

Postcondition: The existence of a k-cycle in G is reported, or the graph G' and decomposition $(T', \mathcal{X}')$ are updated by removing vertices of $\mathcal{X}'(T'[i]) \setminus \mathcal{X}'(i)$, resulting in $|\mathcal{X}'(T'[i])| \leq k \cdot |E(\textsc{torso}(G, \mathcal{X}(i)))| + |\mathcal{X}(i)|$. If G' initially contained a k-cycle, then the modifications preserve this fact.

```
for each $j \in N^+_{T'}(i)$ do
    QueryReduceCycle($G', (T', \mathcal{X}'), j, k$)
    Let $\{x, y\} := \mathcal{X}'(i) \cap \mathcal{X}'(j)$, let $A' := \mathcal{X}'(T'[j])$, and let $B := (V(G') \setminus A) \cup \{x, y\}$
    Invoke the $k$-Cycle oracle on $(G'[A], k)$ and apply Lemma 2 to $(G'[A], k, x, y)$
    if the oracle answers yes or Lemma 2 reports an $xy$-path of length $\geq k$ then
        Report the existence of a $k$-cycle and halt
    else
        Let $S$ be the vertex set computed by Lemma 2
        Remove the vertices $\mathcal{X}'(T'[j]) \setminus S$ from $G'$ and $T'$
for each pair $\{x, y\} \in \binom{\mathcal{X}'(i)}{2}$ do
    while $\exists j, j' \in N^+_{T'}(i)$ with $j \neq j'$ and $\mathcal{X}'(i) \cap \mathcal{X}'(j) = \mathcal{X}'(i) \cap \mathcal{X}'(j') = \{x, y\}$ do
        Let $A := \mathcal{X}'(T'[j]) \cup \mathcal{X}'(T'[j'])$, let $B := (V(G') \setminus A) \cup \{x, y\}$
        Invoke the $k$-Cycle oracle on $(G'[A], k)$ and apply Lemma 2 to $(G'[A], k, x, y)$
        if the oracle answers yes or Lemma 2 reports an $xy$-path of length $\geq k$ then
            Report the existence of a $k$-cycle and halt
        else
            Let $S$ be the vertex set computed by Lemma 2
            Choose $j_{\overline{S}} \in \{j, j'\}$ such that $\mathcal{X}'(T'[j_{\overline{S}}]) \setminus \{x, y\}$ contains no vertex of $S$
            Remove $T'[j_{\overline{S}}]$ from $(T', X')$ and remove $\mathcal{X}'(T'[j_{\overline{S}}]) \setminus \{x, y\}$ from $G'$
```

After the procedure terminates, we make a final call to the planar k-Cycle oracle for the remaining graph G' and parameter k. The output of the oracle is given as the output of the procedure. By the postcondition of the procedure, each modification step preserves the existence of a k-cycle. The oracle answer to the final reduced graph G' is therefore the correct answer to the original input instance (G, k). It is easy to see that the algorithm runs in polynomial time using constant-time access to the oracle: note that each iteration of the **while**-loop removes at least one child subtree of node i from the decomposition. Assuming that the algorithm acts according to its specification, it is also easy to see that the overall approach is correct. To establish Theorem 3 it therefore remains to prove that the algorithm adheres to its specifications and that it only queries the oracle for small instances of the k-Path problem *on planar graphs.*

Claim 2 (★). *If Algorithm 1 reports a k-cycle, then G has a k-cycle.*

Claim 3 (★). *If the input to Algorithm 1 satisfies the precondition, then the output satisfies the postcondition.*

Claim 4 (★). *Algorithm 1 only queries the k-*Cycle *oracle with parameter k on planar graphs of order at most $(3k+1)k^{\log_2 3} + k$.*

This concluses the proof of Theorem 3. □

3.3 k-Cycle in Other Graph Families

There are two obstacles when generalizing the Turing kernel for k-Cycle on planar graphs to the other restricted graph families. In the decompose step we have to ensure that each torso of the Tutte decomposition still belongs to the restricted graph family, so that Theorem 2 may be used to deduce the existence of a k-cycle if the width of the Tutte decomposition is sufficiently large. Lemma 3 is used for this purpose. In the query step we have to deal with the fact that the alterations made to the graph by the self-reduction procedure may violate the defining property of the graph class, which can be handled by using an NP-completeness transformation before querying the oracle. Besides these issues, the kernelization is the same as in the planar case.

Lemma 3 (★). *Let $(T, \mathcal{X})$ be a tree decomposition of adhesion at most two of a graph G and let $i \in V(T)$.*

1. *If G is claw-free, then* torso$(G, \mathcal{X}(i))$ *is claw-free.*
2. *If G has maximum degree Δ, then* torso$(G, \mathcal{X}(i))$ *has degrees at most Δ.*
3. *If G is H-minor-free, then* torso$(G, \mathcal{X}(i))$ *is H-minor-free.*

Theorem 4 (★). *The k-*Cycle *problem has a polynomial Turing kernel when restricted to graphs of maximum degree t, claw-free graphs, or $K_{3,t}$-minor-free graphs, for each constant $t \geq 3$.*

4 Turing Kernelization for Finding Paths

Now we turn our attention to the k-Path problem. While the main ideas are the same as in the k-Cycle case, the details are a bit more technical, for two reasons. First of all, since a path may cross several biconnected components, we can no longer restrict ourselves to biconnected graphs and therefore the minimal separators formed by the intersections of adjacent bags of the Tutte decomposition may now have size one or two. Additionally, there are six structurally different ways in which a path may cross a separation of order two and we have to account for all possible options. To query for the relevant information, we need a more robust self-reduction algorithm. Due to space restrictions, the necessary material is developed in the full version [17]. It results in the following theorem.

Theorem 5 (★). *The k-*Path *problem has a polynomial-size Turing kernel when restricted to planar graphs, graphs of maximum degree t, claw-free graphs, or $K_{3,t}$-minor-free graphs, for each constant $t \geq 3$.*

5 Multicolored Paths in Bounded-Degree Graphs

An input for the Multicolored k-Path problem consists of a graph G, an integer k and a (generally not proper) coloring $f\colon V(G) \to [k+1]$ of its vertices. The question is whether there is a path of length k (which must contain $k+1$ vertices) that contains exactly one vertex of each color. Hermelin et al. [14] showed that Multicolored k-Path is WK[1]-complete under polynomial-parameter transformations. They conjectured that WK[1]-hard problems do not have polynomial-size Turing kernels.

Theorem 6 (★). *The* Multicolored k-Path *problem on graphs of maximum degree at most three is WK[1]-hard.*

The theorem shows that the Multicolored k-Path problem remains WK[1]-hard on bounded-degree graphs. However, Theorem 5 shows that the uncolored k-Path problem admits a polynomial Turing kernel on bounded-degree graphs. This indicates that the colored problem may be significantly harder to preprocess than the uncolored version.

6 Conclusion

We presented polynomial-size Turing kernels for k-Path and k-Cycle on restricted graph families using the *Decompose-Query-Reduce* framework, thereby answering an open problem posed by Lokshtanov [19] and Misra et al. [20]. Our results form the second [23] example of adaptive Turing kernelization of polynomial size. (We remark that our results were obtained independently without knowing of the work of Thomassé et al [23].)

The question remains whether k-Path admits a polynomial-size Turing kernel in general graphs. Theorem 6 indicates that the WK[1]-hardness of Multicolored k-Path [14] may not be relevant for the k-Path problem. The Tutte decomposition is of little use in general graphs: in contrast to Theorem 2, the circumference of a general triconnected graph may be as low as $\mathcal{O}(\log n)$ (e.g., for the join of a triangle with a complete binary tree). Analyzing k-Path on chordal graphs may be an intermediate step: the example above shows that even for triconnected chordal graphs the circumference may be $\mathcal{O}(\log n)$.

Our results also prompt the investigation of other subgraph and minor testing problems. For example, does the problem of testing whether H is isomorphic to a subgraph of a planar graph G admit a polynomial Turing kernel, parameterized by $|H|$? The simplest unresolved case of this problem seems to be the Exact k-Cycle problem of finding a cycle of length *exactly*, rather than at least, k. The present approach fails on this problem since it is already unclear how to deal with triconnected planar graphs. Similar questions can be asked for the problem of finding a graph H as a minor in a planar graph G, parameterized by $|H|$. To further understand the nature of Turing kernelization, one might also investigate whether the adaptive Turing kernel given here can be transformed into a non-adaptive Turing kernel, whose queries only depend on the input but not on the answers to earlier queries. Since the queries in a non-adaptive Turing kernel can be executed in parallel, this might offer practical advantages.

Acknowledgments. We are grateful to Michał Pilipczuk for suggesting Theorem 6 and Dániel Marx for suggesting its current easy proof.

References

1. Ambalath, A.M., Balasundaram, R., Koppula, C.R.H.V., Misra, N., Philip, G., Ramanujan, M.S.: On the kernelization complexity of colorful motifs. In: Proc. 5th IPEC, pp. 14–25 (2010)
2. Bilinski, M., Jackson, B., Ma, J., Yu, X.: Circumference of 3-connected claw-free graphs and large Eulerian subgraphs of 3-edge-connected graphs. J. Comb. Theory, Ser. B 101(4), 214–236 (2011)
3. Binkele-Raible, D., Fernau, H., Fomin, F.V., Lokshtanov, D., Saurabh, S., Villanger, Y.: Kernel(s) for problems with no kernel: On out-trees with many leaves. ACM Trans. Algorithms 8(4), 38 (2012)
4. Bodlaender, H.L.: Kernelization: New upper and lower bound techniques. In: Chen, J., Fomin, F.V. (eds.) IWPEC 2009. LNCS, vol. 5917, pp. 17–37. Springer, Heidelberg (2009)
5. Bodlaender, H.L., Demaine, E.D., Fellows, M.R., Guo, J., Hermelin, D., Lokshtanov, D., Müller, M., Raman, V., Rooij, J.V., Rosamond, F.A.: Open problems in parameterized and exact computation - IWPEC 2008. Technical Report UU-CS-2008-017, Utrecht University (2008)
6. Bodlaender, H.L., Downey, R.G., Fellows, M.R., Hermelin, D.: On problems without polynomial kernels. J. Comput. Syst. Sci. 75(8), 423–434 (2009)
7. Bodlaender, H.L., Jansen, B.M.P., Kratsch, S.: Kernel bounds for path and cycle problems. In: Proc. 6th IPEC, pp. 145–158 (2011)
8. Bodlaender, H.L., Jansen, B.M.P., Kratsch, S.: Kernelization lower bounds by cross-composition. SIAM J. Discrete Math. 28(1), 277–305 (2014)
9. Chen, G., Gao, Z., Yu, X., Zang, W.: Approximating longest cycles in graphs with bounded degrees. SIAM J. Comput. 36(3), 635–656 (2006)
10. Chen, G., Yu, X.: Long cycles in 3-connected graphs. J. Comb. Theory, Ser. B 86(1), 80–99 (2002)
11. Chen, G., Yu, X., Zang, W.: The circumference of a graph with no $K_{3,t}$-minor, II. J. Comb. Theory, Ser. B 102(6), 1211–1240 (2012)
12. Diestel, R.: Graph Theory, 4th edn. Springer, Heidelberg (2010)
13. Flum, J., Grohe, M.: Parameterized Complexity Theory. Springer-Verlag New York, Inc. (2006)
14. Hermelin, D., Kratsch, S., Sołtys, K., Wahlström, M., Wu, X.: A completeness theory for polynomial (Turing) kernelization. In: Proc. 8th IPEC, pp. 202–215 (2013)
15. Hopcroft, J.E., Tarjan, R.E.: Dividing a graph into triconnected components. SIAM J. Comput. 2(3), 135–158 (1973), doi:10.1137/0202012
16. Hopcroft, J.E., Tarjan, R.E.: Efficient algorithms for graph manipulation [H] (algorithm 447). Commun. ACM 16(6), 372–378 (1973)

17. Jansen, B.M.P.: Turing kernelization for finding long paths and cycles in restricted graph classes. arXiv, abs/1305.3102 (2014)
18. Li, M.-C., Corneil, D.G., Mendelsohn, E.: Pancyclicity and NP-completeness in planar graphs. Discrete Appl. Math. 98(3), 219–225 (2000)
19. Lokshtanov, D.: New Methods in Parameterized Algorithms and Complexity. PhD thesis, University of Bergen, Norway (2009)
20. Misra, N., Raman, V., Saurabh, S.: Lower bounds on kernelization. Discrete Optim. 8(1), 110–128 (2011)
21. Rogers Jr., H.: Theory of Recursive Functions and Effective Computability. MIT Press, Cambridge (1987)
22. Schäfer, A., Komusiewicz, C., Moser, H., Niedermeier, R.: Parameterized computational complexity of finding small-diameter subgraphs. Optim. Lett. 6(5), 883–891 (2012)
23. Thomassé, S., Trotignon, N., Vuskovic, K.: A polynomial Turing-kernel for weighted independent set in bull-free graphs. In: Proc. 40th WG (2014) (in press)
24. Tutte, W.T.: Connectivity in graphs. Mathematical expositions. University of Toronto Press (1966)
25. Weller, M.: Aspects of Preprocessing Applied to Combinatorial Graph Problems. PhD thesis, Technische Universität Berlin (2013)

Optimal Parallel Quantum Query Algorithms

Stacey Jeffery[1], Frederic Magniez[2], and Ronald de Wolf[3]

[1] IQC, University of Waterloo, Canada
sjeffery@uwaterloo.ca
[2] CNRS, LIAFA, Univ Paris Diderot, Sorbonne Paris-Cité, 75205 Paris, France
frederic.magniez@univ-paris-diderot.fr
[3] CWI and University of Amsterdam, The Netherlands
rdewolf@cwi.nl

Abstract. We study the complexity of quantum query algorithms that make p queries in parallel in each timestep. We show tight bounds for a number of problems, specifically $\Theta((n/p)^{2/3})$ p-parallel queries for element distinctness and $\Theta((n/p)^{k/(k+1)})$ for k-sum. Our upper bounds are obtained by parallelized quantum walk algorithms, and our lower bounds are based on a relatively small modification of the adversary lower bound method, combined with recent results of Belovs et al. on learning graphs. We also prove some general bounds, in particular that quantum and classical p-parallel complexity are polynomially related for all total functions f when p is small compared to f's block sensitivity.

1 Introduction

Using quantum effects to speed up computation has been a prominent research-topic for the past two decades. Most known quantum algorithms have been developed in the model of quantum query complexity, the quantum generalization of decision tree complexity. Here an algorithm is charged for each "query" to the input, while intermediate computation is free (see [15] for more details). This model facilitates the proof of lower bounds, and often, though not always, quantum query upper bounds carry over to quantum time complexity. For certain functions one can obtain large quantum-speedups in this model. For example, Grover's algorithm [21] can search an n-bit database (looking for a bit-position of a 1) using $O(\sqrt{n})$ queries. In contrast, any classical algorithm needs $\Omega(n)$ queries. For some partial functions we know exponential and even unbounded speed-ups [18,34,33,7].

A more recent crop of quantum speed-ups come from algorithms based on *quantum walks*. Such algorithms solve a search problem by embedding the search on a graph, and doing a quantum walk on this graph that converges rapidly to a superposition over only the "marked" vertices, which are the ones containing a solution. An important example is Ambainis's quantum algorithm for solving the *element distinctness* problem [3]. In this problem one is given an input $x \in [q]^n$, and the goal is to find a pair of distinct i and j in $[n]$ such that $x_i = x_j$, or report that none exists. Ambainis's quantum walk solves this in

$O(n^{2/3})$ queries, which is optimal [1]. Classically, $\Theta(n)$ queries are required. Two generalizations of this are the *k-distinctness* problem, where the objective is to find distinct $i_1, \ldots, i_k \in [n]$ such that $x_{i_1} = \cdots = x_{i_k}$, and the *$k$-sum* problem, where the objective is to find distinct $i_1, \ldots, i_k \in [n]$ such that $x_{i_1} + \cdots + x_{i_k} = 0 \bmod q$. Ambainis's approach solves both problems using $O(n^{k/(k+1)})$ quantum queries. Recently, Belovs gave a $o(n^{3/4})$-query algorithm for k-distinctness for any fixed k [8] (which can also be made time-efficient for $k = 3$ [11]). In contrast, Ambainis's $O(n^{k/(k+1)})$-query algorithm is optimal for k-sum [10,14].

Here we consider to what extent such algorithms can be *parallelized*. Doing operations in parallel is a well-known way to trade hardware for time, speeding up computations by distributing the work over many processors that run in parallel. This is becoming ever more prominent in classical computing due to multi-core processors and grid computing. In the case of quantum computing there is an additional reason to consider parallelization, namely the limited lifetime of qubits due to *decoherence*: because of unintended interaction with their environment, qubits tend to lose their quantum properties over a limited amount of time, called the *decoherence time*, and degrade to classical random bits. One way to fight this is to apply quantum error-correction[1], which can counteract the effects of certain models of decoherence. Another way is to try to parallelize as much as possible, completing the computation before the qubits decohere too much (this may of course increase the width of the computation, creating other problems).

We know of only a few results about parallel quantum algorithms, most of them in the circuit model where "time" is measured by the depth of the circuit. A particularly important and beautiful example is the work of Cleve and Watrous [16], who showed how to implement the n-qubit quantum Fourier transform using a quantum circuit of depth $O(\log n)$. As a consequence, they were able to parallelize the quantum component of Shor's algorithm: they showed that one can factor n-bit integers by means of an $O(\log n)$-depth quantum circuit with polynomial-time classical pre- and post-processing. There have also been a number of papers about quantum versions of small-depth classical Boolean circuit classes like AC and NC [29,19,23,35]. Beals et al. [5] show how the quantum circuit model can be efficiently simulated by the more realistic model of a distributed quantum computer (see also [20]). The setting of *measurement-based* quantum computing (see [25] and references therein) in some cases allows more parallelization than the usual circuit model. Another example, the only one we know of in the setting of query complexity, is Zalka's tight analysis of parallelizing quantum search [36, Section 4]. Suppose one wants to search an n-bit database, with the ability to do p queries in parallel in one time-step. An easy way to make use of this parallelism is to view the database as p databases of n/p bits each, and to run a separate copy of Grover's algorithm on each of those. This finds a 1-position with high probability using $O(\sqrt{n/p})$ p-parallel queries, and Zalka showed that this is optimal.

[1] Parallelism is in fact *necessary* to do quantum error-correction against a constant noise rate: sequential operations cannot keep up with the parallel build-up of errors.

Our Results. We focus on parallel quantum algorithms in the setting of quantum query complexity. Consider a function $f : \mathcal{D} \to \{0,1\}$, with $\mathcal{D} \subseteq [q]^n$. For standard (sequential) query complexity, let $Q(f)$ denote the bounded-error quantum query complexity of f, i.e., the minimal number of queries needed among all quantum algorithms that (for every input $x \in \mathcal{D}$) output $f(x)$ with probability at least 2/3. In the p-parallel query model, for some integer $p \geq 1$, an algorithm can make up to p quantum queries in parallel in each timestep. In that case, we let $Q^{p\|}(f)$ denote the bounded-error p-parallel complexity of f. As always in query complexity, all intermediate input-independent computation is free. For every function, we have $Q(f)/p \leq Q^{p\|}(f) \leq Q(f)$.

An extreme case of the parallel model is where p large enough so that $Q^{p\|}(f)$ becomes 1; such algorithms are called "nonadaptive," because all queries are made in parallel. Montanaro [28] showed that for total functions, such nonadaptive quantum algorithms cannot improve much over classical algorithms: every Boolean function that depends on n input bits needs $p \geq n/2$ nonadaptive quantum queries for exact computation, and $p = \Omega(n)$ for bounded-error.

Here we prove matching upper and lower bounds on the p-parallel complexity $Q^{p\|}(f)$ for a number of problems: $\Theta((n/p)^{2/3})$ queries for element distinctness and $\Theta((n/p)^{k/(k+1)})$ for the k-sum problem for any constant $k > 1$. Our upper bounds are obtained by parallelized quantum walk algorithms, and our lower bounds are based on a modification of the adversary lower bound method combined with some recent results by Belovs et al. about using so-called "learning graphs," both for upper and for lower bounds [9,13,10,14]. The modification we need to make is surprisingly small, and technically we need to do little more than adapt recent progress on sequential algorithms to the parallel case. Still, we feel this extension is important because: (1) our techniques may be useful for proving future lower bounds; (2) parallel quantum algorithms are important and yet have received little attention before; and (3) the fact that the extension is easy and natural increases our confidence that the adversary method is the "right" approach in the parallel as well as the sequential case.

In Section 5 we prove some more "structural" results, i.e., bounds for $Q^{p\|}(f)$ that hold for all Boolean functions $f : \{0,1\}^n \to \{0,1\}$. Specifically, based on earlier results in the sequential model due to Beals et al. [6], we show that if p is not too large then $Q^{p\|}(f)$ is polynomially related to its classical deterministic p-parallel counterpart. We also observe that $Q^{p\|}(f) \approx n/2p$ for almost all f.

2 Preliminaries

Sequential and Parallel Query Complexity. We use $[n] := \{1,\dots,n\}$, $\binom{[n]}{k} := \{S \subseteq [n] : |S| = k\}$, $\binom{[n]}{\leq k} := \{S \subseteq [n] : |S| \leq k\}$, and $\binom{n}{\leq k} := \sum_{s=0}^{k} \binom{n}{s}$.

We will consider algorithms in the p-parallel quantum query model. A quantum query to an input $x \in [q]^n$ corresponds to the unitary map $|i,b\rangle \mapsto |i,b+x_i\rangle$. Here the first n-dimensional register contains the index $i \in [n]$ of the queried element, and the value of that element is added (in $\mathbb{Z}_q$) to the contents of the second (q-dimensional) register. In order to enable an algorithm to not make a query on part of its state, we extend the previous unitary map to the case $i = 0$ by $|0, b\rangle \mapsto |0, b\rangle$. In each timestep we can make up to p quantum queries in parallel by applying the map $|i_1, b_1, \ldots, i_p, b_p\rangle \mapsto |i_1, b_1 + x_{i_1}, \ldots, i_p, b_p + x_{i_p}\rangle$ at unit cost. All intermediate input-independent computation is free.

Consider a function $f : \mathcal{D} \to \{0, 1\}$, with $\mathcal{D} \subseteq [q]^n$. When $p = 1$ we have the standard sequential query complexity, and we let $Q_\varepsilon(f)$ denote the quantum query complexity of f with error probability $\leq \varepsilon$ on every input $x \in \mathcal{D}$. For general p, let $Q_\varepsilon^{p\|}(f)$ be the p-parallel complexity of f. Note that $Q_\varepsilon(f)/p \leq Q_\varepsilon^{p\|}(f) \leq Q_\varepsilon(f)$ for every function. The exact value of the error probability ε does not matter, as long as it is a constant $< 1/2$. We usually fix $\varepsilon = 1/3$, abbreviating $Q(f) = Q_{1/3}(f)$ and $Q^{p\|}(f) = Q_{1/3}^{p\|}(f)$ as in the introduction.

We will use an extension of the adversary bound for the usual sequential (1-parallel) quantum query model. An *adversary matrix* Γ for f is a real-valued matrix whose rows are indexed by $f^{-1}(0)$ and whose columns by $f^{-1}(1)$. Let Δ_j be the Boolean matrix whose rows and columns are indexed by $x \in f^{-1}(0)$ and $y \in f^{-1}(1)$, such that $\Delta_j[x, y] = 1$ if $x_j \neq y_j$, and $\Delta_j[x, y] = 0$ otherwise. The (negative-weights) adversary bound for f is given by:

$$\mathrm{ADV}(f) = \max_{\Gamma} \frac{\|\Gamma\|}{\max_{j\in[n]} \|\Gamma \circ \Delta_j\|}, \tag{1}$$

where Γ ranges over all adversary matrices for f, '$\circ$' denotes entry-wise product of two matrices, and '$\|\cdot\|$' denotes the operator norm associated to the ℓ_2 norm. This lower bound (often denoted $\mathrm{ADV}^{\pm}(f)$ instead of $\mathrm{ADV}(f)$) was introduced by Høyer et al. [22], generalizing Ambainis [2]. They showed $Q_\varepsilon(f) \geq \frac{1}{2}(1 - \sqrt{\varepsilon(1-\varepsilon)})\mathrm{ADV}(f)$ for all f. Reichardt et al. [32,26] showed this is tight: $Q(f) = \Theta(\mathrm{ADV}(f))$ for all f.

Quantum Walks. We will construct and analyze our algorithms in the quantum walk framework of [27], which we now briefly describe. Given a reversible Markov process P on state space V, and a subset $M \subset V$ of marked elements, we define three costs: the setup cost, S, is the cost to construct a superposition over all states $\sum_{v\in V} \sqrt{\pi_v}|v\rangle$, where π_v is the probability of vertex v in the stationary distribution π of P; the checking cost, C, is the cost to check if a state $v \in V$ is in M; and the update cost, U, is the cost to perform the map $|v\rangle|0\rangle \mapsto |v\rangle \sum_{u\in V} \sqrt{P_{vu}}|u\rangle$, where P_{vu} is the transition probability in P to go from v to u. Then, if δ is the spectral gap of P, and ε is a lower bound on $\sum_{v\in M} \pi_v$ whenever M is nonempty, we can determine if M is nonempty with bounded error probability in cost $O\left(\mathsf{S} + \frac{1}{\sqrt{\varepsilon}}\left(\frac{1}{\sqrt{\delta}}\mathsf{U} + \mathsf{C}\right)\right)$. If S, U and C denote query complexities, then the above expression gives the bounded-error query complexity of the quantum walk algorithm. If they denote *p-parallel* query complexities, the above expression gives the bounded-error p-parallel complexity.

3 Lower Bounds for Parallel Quantum Query Complexity

3.1 Adversary Bound for Parallel Algorithms

We start by extending the adversary bound for the usual sequential quantum query algorithms to p-parallel algorithms. For $J \subseteq [n]$, let x_J be the string x restricted to the entries in J. Let Δ_J be the Boolean matrix whose rows are indexed by $x \in f^{-1}(0)$ and whose columns are indexed by $y \in f^{-1}(1)$, and that has a 1 at position (x, y) iff $x_J \neq y_J$ (i.e., $x_j \neq y_j$ for at least one $j \in J$). For $J = \emptyset$, Δ_J is the all-0 matrix. Define the following quantity:

$$\mathrm{ADV}^{p\|}(f) = \max_\Gamma \frac{\|\Gamma\|}{\max_{J \in \binom{[n]}{\leq p}} \|\Gamma \circ \Delta_J\|}. \tag{2}$$

The following fact (proved in our full version [24]) implies we only need to consider sets $J \in \binom{[n]}{p}$ in the above definition: $\mathrm{ADV}^{p\|}(f)$ equals $\max\limits_{\Gamma} \frac{\|\Gamma\|}{\max_{J \in \binom{[n]}{p}} \|\Gamma \circ \Delta_J\|}$ up to a factor of 2. We could even use the latter as an alternative definition of $\mathrm{ADV}^{p\|}(f)$.

Fact 1. *For every set $J \subseteq K \subseteq [n]$, we have $\|\Gamma \circ \Delta_J\| \leq 2\|\Gamma \circ \Delta_K\|$.*

Theorem 2. *For every $f : \mathcal{D} \to \{0, 1\}$ and $\mathcal{D} \subseteq [q]^n$, $Q^{p\|}(f) = \Theta(\mathrm{ADV}^{p\|}(f))$.*

Proof. In order to derive p-parallel lower bounds from sequential lower bounds, observe that we can make a bijection between input $x \in [q]^n$ and a larger string X indexed by all sets $J \in \binom{[n]}{\leq p}$, such that $X_J = (x_j)_{j \in J}$. That is, each index J of X corresponds to up to p indices j of x. We now define a new function $F : \mathcal{D}' \to \{0, 1\}$, where $\mathcal{D}'$ is the set of X as above, in 1-to-1 correspondence with the elements of $x \in \mathcal{D}$, and $F(X)$ is defined as $f(x)$. One query to X can be simulated by p parallel queries to x, and vice versa, so we have $Q^{p\|}(f) = Q(F)$. We have $Q(F) = \Theta(\mathrm{ADV}(F))$ by [32,26]. Now Eq. (1) applied to F gives the claimed lower bound of Eq. (2) on $Q^{p\|}(f)$. □

Sometimes we can even use the same adversary matrix Γ to obtain optimal lower bounds for F and f. An example is the n-bit OR-function. Let Γ be the all-ones $1 \times n$ matrix, with the row corresponding to input 0^n and the columns indexed by all weight-1 inputs. Then $\|\Gamma\| = \sqrt{n}$ and $\|\Gamma \circ \Delta_j\| = 1$ for all $j \in [n]$, and hence $Q(\mathrm{OR}) = \Omega(\sqrt{n})$. To get p-parallel lower bounds, we define a new function $F : X \mapsto \{0, 1\}$ as in the proof of Theorem 2. We can use the same Γ, with the n columns still indexed by the weight-1 inputs to f (which induce 1-inputs to F). Now J ranges over subsets of $[n]$ of size at most p, and Δ_J will be the matrix whose (x, y)-entry is 1 if there is at least one $j \in J$ such that $x_j \neq y_j$. Note that $\|\Gamma \circ \Delta_J\| = \sqrt{|J|}$ for all J. Hence $Q^{p\|}(\mathrm{OR}) = \Omega(\mathrm{ADV}(F)) = \Omega(\sqrt{n/p})$. This is optimal and was already proved (in a different way) by Zalka [36, Section 4].

3.2 Belovs's Learning Graph Approach

Recently Belovs [9] gave a new approach to designing quantum algorithms, introducing the model of *learning graphs* to prove upper bounds on the adversary bound, and hence on quantum query complexity. We state it here for *certificate structures.* We define these below, slightly simpler and less general than Definitions 1 and 3 of Belovs and Rosmanis [13] (for us M denotes a minimal certificate, while in [13] it denotes the set of supersets of a minimal certificate).

Definition 1. *Let $\mathcal{C}$ be a set of incomparable subsets of $[n]$. We say $\mathcal{C}$ is a* 1-certificate structure *for a function $f : \mathcal{D} \to \{0,1\}$, with $\mathcal{D} \subseteq [q]^n$, if for every $x \in f^{-1}(1)$ there exists an $M \in \mathcal{C}$ such that for all $y \in \mathcal{D}$, $y_M = x_M$ implies $f(y) = 1$. We say $\mathcal{C}$ is k-*bounded *if $|M| \leq k$ for all $M \in \mathcal{C}$.*

The learning graph complexity of $\mathcal{C}$ is defined in the following in its primal formulation as a minimization problem (we will see an equivalent dual formulation soon). Let $\mathcal{E} = \{(S, j) : S \subseteq [n], j \in [n] \backslash S\}$. For $e = (S, j) \in \mathcal{E}$, we use $s(e) = S$ and $t(e) = S \cup \{j\}$.

$$\mathrm{LGC}(\mathcal{C}) = \min \sqrt{\textstyle\sum_{e\in\mathcal{E}} w_e} \quad \text{such that} \tag{3}$$

$$\sum_{e\in\mathcal{E}} \frac{\theta_e(M)^2}{w_e} \leq 1 \qquad \text{for all } M \in \mathcal{C} \tag{4}$$

$$\sum_{e\in\mathcal{E}:t(e)=S} \theta_e(M) = \sum_{e\in\mathcal{E}:s(e)=S} \theta_e(M) \quad \text{for all } M \in \mathcal{C}, \emptyset \neq S \subseteq [n], M \not\subseteq S \tag{5}$$

$$\sum_{e\in\mathcal{E}:s(e)=\emptyset} \theta_e(M) = 1 \qquad \text{for all } M \in \mathcal{C} \tag{6}$$

$$\theta_e(M) \in \mathbb{R}, w_e \geq 0 \qquad \text{for all } e \in \mathcal{E} \text{ and } M \in \mathcal{C} \tag{7}$$

For each M, $\theta_e(M)$ is a *flow* from $\emptyset$ to M on the graph with vertices $\{S \subseteq [n]\}$ and edges $\{\{S, S \cup \{j\}\} : (S, j) \in \mathcal{E}\}$ if $\theta_e(M)$ satisfies condition (5). Moreover, $\theta_e(M)$ is a *unit flow* if it also satisfies condition (6).

The learning graph complexity of $\mathcal{C}$ is an upper bound on $\mathrm{ADV}(f)$, and hence on $Q(f)$, for any function f with certificate structure $\mathcal{C}$. This bound is not always optimal, since it only depends on the certificate structure of f. E.g. k-distinctness has quantum query complexity $o(n^{3/4})$ even though it has the same 1-certificate structure as k-sum, whose quantum query complexity is $\Theta(n^{k/(k+1)})$ [10,14]. However, Belovs and Rosmanis [13] proved that for a special class of functions, it turns out the upper bound $\mathrm{LGC}(\mathcal{C})$ is optimal.

Definition 2. *An* orthogonal array *of length k is a set $T \subseteq [q]^k$, such that for every $i \in [k]$ and every $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_k$ there exists exactly one $x_i \in [q]$ such that $(x_1, \ldots, x_k) \in T$.*

Theorem 3 (Belovs-Rosmanis). *Let $\mathcal{C}$ be a k-bounded 1-certificate structure for some constant k, $q \geq 2|\mathcal{C}|$, and let each $M \in \mathcal{C}$ be equipped with an orthogonal array T_M of length $|M|$. Define a Boolean function $f : [q]^n \to \{0,1\}$ by: $f(x) = 1$ iff there exists an $M \in \mathcal{C}$ such that $x_M \in T_M$. Then $Q(f) = \Theta(\mathrm{LGC}(\mathcal{C}))$.*

For example, the element distinctness problem ED on input $x \in [q]^n$ is induced by the 2-bounded 1-certificate structure $\mathcal{C} = \binom{[n]}{2}$, equipped with associated orthogonal arrays $T_{\{i,j\}} = \{(v,v) : v \in [q]\}$. Hence $Q(\mathrm{ED}) = \Theta(\mathrm{LGC}(\mathcal{C}))$.

Belovs and Rosmanis [13] show that an equivalent dual definition of the learning graph complexity as a maximization problem is the following:

$$\begin{aligned} \mathrm{LGC}(\mathcal{C}) = &\max \sqrt{\textstyle\sum_{M\in\mathcal{C}} \alpha_\emptyset(M)^2} && && (8)\\ &\text{s.t. } \textstyle\sum_{M\in\mathcal{C}}(\alpha_{s(e)}(M) - \alpha_{t(e)}(M))^2 \le 1 && \text{for all } e \in \mathcal{E} && (9)\\ &\alpha_S(M) = 0 && \text{whenever } M \subseteq S &&\\ &\alpha_S(M) \in \mathbb{R} && \text{for all } S \subseteq [n] \text{ and } M \in \mathcal{C} && \end{aligned}$$

In particular, that means we can prove *lower* bounds on $\mathrm{LGC}(\mathcal{C})$ (and hence, for the functions described in Theorem 3, on $Q(f)$) by exhibiting a feasible solution $\{\alpha_S(M)\}$ for this maximization problem and calculating its objective value.

Before stating a similar result for p-parallel query complexity, we first adapt learning graphs. Edges, which were of type $e = (S, j)$ with $S \subseteq [n]$ and $j \in [n] \setminus S$, are now of type $e = (S, J)$ with $S \subseteq [n]$, $J \subseteq [n] \setminus S$ and $|J| \le p$.

Definition 3. *The* p-parallel learning graph complexity $\mathrm{LGC}^{p\|}(\mathcal{C})$ *of* $\mathcal{C}$ *is defined as* $\mathrm{LGC}(\mathcal{C})$ *where we replace the edge set* $\mathcal{E}$ *with* $\mathcal{E}_p = \{(S,J) : S \subseteq [n], J \subseteq [n] \setminus S, |J| \le p\}$. *Its dual is analogous. In particular, we replace* (9) *by*

$$\textstyle\sum_{M\in\mathcal{C}}(\alpha_{s(e)}(M) - \alpha_{t(e)}(M))^2 \le 1 \textit{ for all } e = (S,J) \in \mathcal{E}_p,$$

where $s(e) = S$ *and* $t(e) = S \cup J$. *We call this modified constraint "parallel-*(9)*."*

As in the special case of $p = 1$, the p-parallel learning graph complexity of $\mathcal{C}$ provides an upper bound on $\mathrm{ADV}^{p\|}(f)$, and hence on $Q^{p\|}(f)$, for any function f having that same certificate structure. The proof is given in our full version [24].

Lemma 1. *Let* $\mathcal{C}$ *be a certificate structure for* f. *Then* $\mathrm{ADV}^{p\|}(f) \le \mathrm{LGC}^{p\|}(\mathcal{C})$.

We now generalize Theorem 3 to the p-parallel case. The proof, given in [24], is an adaptation of the proof of [13, Theorem 5].

Theorem 4. *Let* $\mathcal{C}$ *be a* k*-bounded 1-certificate structure for some constant* k, $q \ge 2|\mathcal{C}|$, *and let each* $M \in \mathcal{C}$ *be equipped with an orthogonal array* T_M *of length* $|M|$. *Define a Boolean function* $f : [q]^n \to \{0,1\}$ *as follows:* $f(x) = 1$ *iff there exists an* $M \in \mathcal{C}$ *such that* $x_M \in T_M$. *Then* $Q^{p\|}(f) = \Theta(\mathrm{LGC}^{p\|}(\mathcal{C}))$.

4 Parallel Quantum Query Complexity of Specific Functions

4.1 Algorithms

In this section we give upper bounds for element distinctness and k-sum in the p-parallel quantum query model, by way of quantum walk algorithms.

Our p-parallel algorithm for element distinctness is based on the sequential query algorithm for element distinctness of Ambainis [3]. Ambainis's algorithm uses a quantum walk on a Johnson graph, $J(n, r)$, which has vertex set $V = \{S \subseteq [n] : |S| = r\}$ and edge set $\{\{S, S'\} \subseteq V : |S \setminus S'| = 1\}$. Each state $S \in V$ represents a set of queried indices. The algorithm seeks a state S containing (i, x_i) and (j, x_j) such that $i \neq j$ and $x_i = x_j$. Such a state is said to be *marked.*

Theorem 5. *Element distinctness on* $[q]^n$ *has* $Q^{p\|}(\mathrm{ED}) = O((n/p)^{2/3})$.

Proof. We modify Ambainis's quantum walk algorithm slightly. Consider a walk $J(n, r/p)^p$, on p copies of the Johnson graph $J(n, r/p)$. Vertices are p-tuples $(S_1, S_2, \ldots, S_p)$ where, for each $i \in [p]$, $S_i \subseteq [n]$ and $|S_i| = r/p$. Two vertices $(S_1, S_2, \ldots, S_p)$ and $(S'_1, S'_2, \ldots, S'_p)$ are adjacent if, for each $i \in [p]$, $|S_i \setminus S'_i| = 1$. We call a state $(S_1, S_2, \ldots, S_p)$ *marked* if there are $j, j' \in \bigcup_{i=1}^p S_i$ such that $x_j = x_{j'}$. Since the stationary distribution is μ^p, where μ is the uniform distribution on $\binom{[n]}{r/p}$, the probability that a state is marked is at least $\varepsilon = \Omega(r^2/n^2)$.

The setup cost is only $\mathsf{S} = O(r/p)$ p-parallel queries, since it suffices to query r elements in the initial superposition over all states. Similarly, the update requires that we query and unquery p elements, but we can accomplish this in two p-parallel queries, so $\mathsf{U} = O(1)$. Also, $\mathsf{C} = 0$. Finally, the eigenvalues of the product of p copies of a graph are exactly the products of p eigenvalues of that graph. Hence if the largest eigenvalue of a graph is 1 and the second-largest is $1 - \delta$, then the same will be true for the product graph. Accordingly, the spectral gap δ of p copies of $J(n, r/p)$ is exactly the spectral gap of one copy of $J(n, r/p)$, which is $\Omega(p/r)$. We can now calculate the p-parallel query complexity of element distinctness as $O\left(\mathsf{S} + \frac{1}{\sqrt{\varepsilon}}\left((\frac{1}{\sqrt{\delta}})\mathsf{U} + \mathsf{C}\right)\right) = O\left(\frac{r}{p} + (\frac{n}{r})\left(\sqrt{\frac{r}{p}}\right)\right) = O\left(\frac{r}{p} + \frac{n}{\sqrt{rp}}\right)$. Setting r to the optimal $n^{2/3}p^{1/3}$ gives an upper bound of $O((n/p)^{2/3})$. □

It is easy to generalize our element distinctness upper bound to k-sum:

Theorem 6. *k-sum on* $[q]^n$ *has* $Q^{p\|}(k\text{-sum}) = O((n/p)^{k/(k+1)})$.

Proof. Again, we walk on p copies of $J(n, r/p)$, but now we consider a state $(S_1, S_2, \ldots, S_p)$ marked if there are distinct indices $i_1, \ldots, i_k \in \bigcup_{i=1}^p S_i$ such that $\sum_{j=1}^k x_{i_j} = 0 \pmod q$. The proportion of marked states in a 1-instance is $\varepsilon = \Omega(r^k/n^k)$. All other parameters are as in Theorem 5. We get the following upper bound for k-sum: $O\left(\mathsf{S} + \frac{1}{\sqrt{\varepsilon}}\left(\frac{1}{\sqrt{\delta}}\mathsf{U} + \mathsf{C}\right)\right) = O\left(\frac{r}{p} + \frac{n^{k/2}}{r^{k/2}}\left(\sqrt{\frac{r}{p}}\right)\right) = O\left(\frac{r}{p} + \frac{n^{k/2}}{r^{(k-1)/2}\sqrt{p}}\right)$. Setting $r = n^{k/(k+1)}p^{1/(k+1)}$ gives $O((n/p)^{k/(k+1)})$. □

4.2 Lower Bounds

We now use the ideas from Section 3.2 to prove p-parallel lower bounds for ED and k-sum, matching our upper bounds if the alphabet size q is sufficiently large. Our proofs are generalizations of the sequential lower bounds in [13, Section 4].

Theorem 7. *For $q \geq 2\binom{n}{2}$,* ED *on* $[q]^n$ *has* $Q^{p\|}(\mathrm{ED}) = \Omega((n/p)^{2/3})$.

Proof. Recall that element distinctness is induced by the 1-certificate structure $\mathcal{C} = \binom{[n]}{2}$, equipped with associated orthogonal arrays $T_{\{i,j\}} = \{(v,v) : v \in [q]\}$. By Theorem 4, it suffices to prove the lower bound on the p-parallel learning graph complexity of ED. For this, it suffices to exhibit a feasible solution to the dual (8) and to lower bound its objective function. Note that the elements of $\mathcal{E}$ are now of the form (S, J), where $S \subseteq [n]$ and $J \subseteq [n] \setminus S$ with $|J| \leq p$. Define

$$\alpha_j = \tfrac{1}{2n}\max((n/p)^{2/3} - j/p, 0), \quad \text{and} \quad \alpha_S(M) = \begin{cases} 0 & \text{if } M \subseteq S \\ \alpha_{|S|} & \text{otherwise.} \end{cases}$$

To show that this is a feasible solution, the only constraint we need to verify is parallel-(9). Fix $S \subseteq [n]$ of some size s, and a set $J \subseteq [n] \setminus S$ with $|J| \leq p$. Let L denote the left-hand side of parallel-(9), which is a sum over all $\binom{n}{2}$ certificates $M \in \mathcal{C}$. With respect to $e = (S, J)$, there are four kinds of $M = \{i, j\}$:

1. $i, j \in S$. Then $\alpha_{t(e)}(M) = \alpha_{s(e)}(M) = 0$, so these M contribute 0 to L.
2. $i \in S, j \in J$. There are $s|J| \leq sp$ such M, and each contributes α_s^2 to L because $\alpha_{s(e)}(M) = \alpha_s$ and $\alpha_{t(e)}(M) = 0$.
3. $i, j \notin S$, $i, j \in J$. There are $\binom{|J|}{2} \leq \binom{p}{2}$ such M, each contributes α_s^2 to L.
4. i and/or $j \notin S \cup J$. There are $\binom{n}{2} - \binom{s+|J|}{2} \leq n^2$ such M, each contributes $|\alpha_s - \alpha_{s+|J|}|^2$ to L.

Hence, using $\alpha_s = 0$ if $s \geq n^{2/3}p^{1/3}$, $\alpha_s \leq \alpha_0 = \frac{1}{2p^{2/3}n^{1/3}}$, and $|\alpha_s - \alpha_{s+|J|}|^2 \leq 1/4n^2$, we can establish constraint parallel-(9):

$$L \leq \left(sp + \tbinom{p}{2}\right)\alpha_s^2 + n^2|\alpha_s - \alpha_{s+|J|}|^2 \leq p(n^{2/3}p^{1/3} + p/2)\frac{1}{4p^{4/3}n^{2/3}} + n^2\frac{1}{4n^2} \leq 1.$$

Hence our solution is feasible. Its objective value is $\sqrt{\binom{n}{2}\alpha_0^2} = \Omega((n/p)^{2/3})$. □

The lower bound proof for k-sum is similar. Here we use certificate structure $\mathcal{C} = \binom{[n]}{k}$ with the orthogonal array $T = \{(v_1, \ldots, v_k) : \sum_{i=1}^k v_i = 0 \mod q\}$, which induces k-sum. In [24], we show that the following has objective value $\sqrt{\binom{n}{k}\alpha_0^2} = \Omega\left((n/p)^{k/(k+1)}\right)$ and is feasible for $\mathrm{LGC}^{p\|}(\mathcal{C})$:

$$\alpha_j = \tfrac{1}{2n^{k/2}}\max((n/p)^{k/(k+1)} - j/p, 0) \quad \text{and} \quad \alpha_S(M) = \begin{cases} 0 & \text{if } M \subseteq S \\ \alpha_{|S|} & \text{otherwise;} \end{cases}$$

Theorem 8. *For $q \geq 2\binom{n}{k}$, k-sum on $[q]^n$ has* $Q^{p\|}(k\text{-sum}) = \Omega\left((n/p)^{k/(k+1)}\right)$.

5 Some General Bounds

In this section we will relate quantum and classical p-parallel complexity. For the sequential model ($p = 1$) it is known that quantum bounded-error query complexity is no more than a 6th power less than classical deterministic complexity, for all total Boolean functions [6]. Here we will see to what extent we can prove a similar result for the p-parallel model.

We start with a few definitions, referring to [15] for more details. Let $f : \{0,1\}^n \to \{0,1\}$ be a total Boolean function. For $b \in \{0,1\}$, a *b-certificate* for f is an assignment $C : S \to \{0,1\}$ to a subset S of the n variables, such that $f(x) = b$ whenever x is consistent with C. The *size* of C is $|S|$. The *certificate complexity $C_x(f)$ of f on x* is the size of a smallest $f(x)$-certificate that is consistent with x. The *certificate complexity* of f is $C(f) = \max_x C_x(f)$. The 1-*certificate complexity* of f is $C^{(1)}(f) = \max_{\{x:f(x)=1\}} C_x(f)$. Given an input $x \in \{0,1\}^n$ and subset $B \subseteq [n]$ of indices of variables, let x^B denote the n-bit input obtained from x by negating all bits x_i whose index i is in B. The *block sensitivity* $bs(f,x)$ of f at input x, is the maximal integer k such that there exist disjoint sets $B_1, \ldots, B_k$ satisfying $f(x) \neq f(x^{B_i})$ for all $i \in [k]$. The *block sensitivity* of f is $bs(f) = \max_x bs(f,x)$. Nisan [30] proved that

$$bs(f) \leq C(f) \leq bs(f)^2. \tag{10}$$

Via a standard reduction [31], Zalka's $\Theta(\sqrt{n/p})$ bound for OR implies:

Theorem 9. *For every $f : \{0,1\}^n \to \{0,1\}$, $Q^{p\|}(f) = \Omega(\sqrt{bs(f)/p})$.*

We now prove a general upper bound on deterministic p-parallel complexity:

Theorem 10. *For every $f : \{0,1\}^n \to \{0,1\}$, $D^{p\|}(f) \leq \lceil C^{(1)}(f)/p \rceil bs(f)$.*

Proof. Beals et al. [6, Lemma 5.3] give a deterministic decision tree for f that runs for at most $bs(f)$ rounds, and in each round queries all variables of a 1-certificate, and substituting their values into the function. This reduces the function to a constant. By parallelizing the querying of the certificate we can implement every round using $\lceil C^{(1)}(f)/p \rceil$ p-parallel steps. □

$D^{p\|}(f)$ and $Q^{p\|}(f)$ are polynomially related if p is not too big:

Theorem 11. *For every $f : \{0,1\}^n \to \{0,1\}$, $c > 1$, $p \leq bs(f)^{1/c}$, we have $D^{p\|}(f) \leq O(Q^{p\|}(f)^{6+4/(c-1)})$.*

Proof. We can assume $C(f) = C^{(1)}(f)$ (else consider $1-f$). By Eq. (10) we have $p \leq bs(f)^{1/c} \leq C^{(1)}(f)$. We also have $C^{(1)}(f) \leq bs(f)^2$. The assumption on p is equivalent to $p \leq (bs(f)/p)^{1/(c-1)}$. Using Theorems 9 and 10, we obtain

$$\begin{aligned} D^{p\|}(f) &\leq \lceil C^{(1)}(f)/p \rceil bs(f) \leq O(bs(f)^3/p) = O((bs(f)/p)^3 p^2) \\ &\leq O((bs(f)/p)^{3+2/(c-1)}) \leq O(Q^{p\|}(f)^{6+4/(c-1)}). \end{aligned}$$ □

For example, if $p \leq bs(f)^{1/3}$ then $Q^{p\|}(f)$ is at most an 8th power smaller than $D^{p\|}(f)$. Whether superpolynomial gaps exist for large p remains open.

We end with an observation about random functions. Van Dam [17] showed that an n-bit input string x can be recovered with high probability using $n/2 + O(\sqrt{n})$ quantum queries, hence $Q(f) \leq n/2 + O(\sqrt{n})$ for all $f : \{0,1\}^n \to \{0,1\}$.

His algorithm already applies its queries in parallel, so allows us to compute x using roughly $n/2p$ p-parallel quantum queries. Ambainis et al. [4] proved an essentially optimal lower bound for random functions: almost all f have $Q(f) \geq (1/2 - o(1))n$. Since trivially $Q(f) \leq pQ^{p\|}(f)$, we obtain the p-parallel lower bound $Q^{p\|}(f) \geq (1/2 - o(1))n/p$ for almost all f.

6 Conclusion and Future Work

This paper is the first to systematically study the power and limitations of parallelism for quantum query algorithms. We leave open many interesting questions:

– There are many other computational problems whose p-parallel complexity is unknown, for example finding a triangle in a graph or deciding whether two given matrices multiply to a third one. For both of these problems, however, even the sequential quantum query complexity is still open.
– We suspect Theorem 11 is non-optimal, and conjecture that $D^{p\|}(f)$ and $Q^{p\|}(f)$ are polynomially related for large p as well. Montanaro's result [28] about the weakness of maximally parallel quantum algorithms is evidence.
– Can we find relations with quantum communication complexity? Nonadaptive quantum query algorithms induce one-way communication protocols, while fully adaptive ones induce protocols that are very interactive. Our p-parallel algorithms would sit somewhere in between.

Acknowledgment. We are partially supported by the French ANR Blanc project ANR-12-BS02-005 (RDAM), a Vidi grant from the Netherlands Organization for Scientific Research (NWO), an ERC Consolidator grant, the European Commission IST STREP project Quantum Algorithms (QALGO) 600700 and the US ARO. We thank Jérémie Roland for helpful discussions.

References

1. Aaronson, S., Shi, Y.: Quantum lower bounds for the collision and the element distinctness problems. J. of the ACM 51(4), 595–605 (2004)
2. Ambainis, A.: Quantum lower bounds by quantum arguments. J. of Computer and System Sciences 64(4), 750–767 (2002)
3. Ambainis, A.: Quantum walk algorithm for element distinctness. SIAM J. on Computing 37(1), 210–239 (2007), Earlier version in FOCS 2004. quant-ph/0311001
4. Ambainis, A., Bačkurs, A., Smotrovs, J., de Wolf, R.: Optimal quantum query bounds for almost all Boolean functions. In: Proc. 30th STACS, pp. 446–453 (2013)
5. Beals, R., Brierley, S., Gray, O., Harrow, A., Kutin, S., Linden, N., Shepherd, D., Stather, M.: Efficient distributed quantum computing. Proc. of the Royal Society A469, 2153 (2013)
6. Beals, R., Buhrman, H., Cleve, R., Mosca, M., de Wolf, R.: Quantum lower bounds by polynomials. J. of the ACM 48(4), 778–797 (2001)
7. de Beaudrap, N., Cleve, R., Watrous, J.: Sharp quantum vs. classical query complexity separations. Algorithmica 34(4), 449–461 (2002)

8. Belovs, A.: Learning-graph-based quantum algorithm for k-distinctness. In: Proc. of 53rd IEEE FOCS, pp. 207–216 (2012)
9. Belovs, A.: Span programs for functions with constant-sized 1-certificates. In: Proc. of 43rd ACM STOC, pp. 77–84 (2012)
10. Belovs, A.: Adversary lower bound for element distinctness, arXiv:1204.5074 (2012)
11. Belovs, A., Childs, A.M., Jeffery, S., Kothari, R., Magniez, F.: Time-efficient quantum walks for 3-distinctness. In: Fomin, F.V., Freivalds, R., Kwiatkowska, M., Peleg, D. (eds.) ICALP 2013, Part I. LNCS, vol. 7965, pp. 105–122. Springer, Heidelberg (2013)
12. Belovs, A., Lee, T.: Quantum algorithm for k-distinctness with prior knowledge on the input. Technical Report arXiv:1108.3022, arXiv (2011)
13. Belovs, A., Rosmanis, A.: On the power of non-adaptive learning graphs. In: Proc. of 28th IEEE CCC, pp. 44–55 (2013) References are to arXiv:1210.3279v2
14. Belovs, A., Špalek, R.: Adversary lower bound for the k-sum problem. In: Proc. of 4th ITCS, pp. 323–328 (2013)
15. Buhrman, H., de Wolf, R.: Complexity measures and decision tree complexity: A survey. Theoretical Computer Science 288(1), 21–43 (2002)
16. Cleve, R., Watrous, J.: Fast parallel circuits for the quantum Fourier transform. In: Proc. of 41st IEEE FOCS, pp. 526–536 (2000)
17. van Dam, W.: Quantum oracle interrogation: Getting all information for almost half the price. In: Proc. of 39th IEEE FOCS, pp. 362–367 (1998)
18. Deutsch, D., Jozsa, R.: Rapid solution of problems by quantum computation. Proc. of the Royal Society of London A439, 553–558 (1992)
19. Green, F., Homer, S., Moore, C., Pollett, C.: Counting, fanout and the complexity of quantum ACC. Quantum Inf. and Comp. 2(1), 35–65 (2002)
20. Grover, L., Rudolph, T.: How significant are the known collision and element distinctness quantum algorithms? Quantum Inf. and Comp. 4(3), 201–206 (2004)
21. Grover, L.K.: A fast quantum mechanical algorithm for database search. In: Proc. of 28th ACM STOC, pp. 212–219 (1996)
22. Høyer, P., Lee, T., Špalek, R.: Negative weights make adversaries stronger. In: Proc. of 39th ACM STOC, pp. 526–535 (2007)
23. Høyer, P., Špalek, R.: Quantum fan-out is powerful. Th. Comp. 1, 81–103 (2005)
24. Jeffery, S., Magniez, F., de Wolf, R.: Optimal parallel quantum query algorithms. arXiv:1309.6116 (2013)
25. Jozsa, R.: An introduction to measurement based quantum computation. In: Angelakis, D.G., Christandl, M., Ekert, A. (eds.) Quantum Information Processing, pp. 137–158. IOS Press (2006) arXiv:0508124
26. Lee, T., Mittal, R., Reichardt, B., Špalek, R., Szegedy, M.: Quantum query complexity of state conversion. In: Proc. of 52nd IEEE FOCS, pp. 344–353 (2011) References are to arXiv:1011.3020v2
27. Magniez, F., Nayak, A., Roland, J., Santha, M.: Search via quantum walk. SIAM J. on Computing 40(1), 142–164 (2011)
28. Montanaro, A.: Nonadaptive quantum query complexity. Information Processing Letters 110(24), 1110–1113 (2010)
29. Moore, C., Nilsson, M.: Parallel quantum computation and quantum codes. SIAM J. on Computing 31(3), 799–815 (2002)
30. Nisan, N.: CREW PRAMs and decision trees. SIAM J. on Computing 20(6), 999–1007 (1991)
31. Nisan, N., Szegedy, M.: On the degree of Boolean functions as real polynomials. Computational Complexity 4(4), 301–313 (1994)

32. Reichardt, B.: Span programs and quantum query complexity: The general adversary bound is nearly tight for every Boolean function. In: Proc. of 50th IEEE FOCS, pp. 544–551 (2009)
33. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM J. on Computing 26(5), 1484–1509 (1997)
34. Simon, D.: On the power of quantum computation. SIAM J. on Computing 26(5), 1474–1483 (1997)
35. Takahashi, Y., Tani, S.: Collapse of the hierarchy of constant-depth exact quantum circuits. In: Proc. of 28th IEEE CCC (2013)
36. Zalka, C.: Grover's quantum searching algorithm is optimal. Physical Review A 60, 2746–2751 (1999)

Sublinear Space Algorithms for the Longest Common Substring Problem

Tomasz Kociumaka[1,⋆], Tatiana Starikovskaya[2,⋆⋆], and Hjalte Wedel Vildhøj[3]

[1] Institute of Informatics, University of Warsaw
[2] National Research University Higher School of Economics (HSE)
[3] Technical University of Denmark, DTU Compute

Abstract. Given m documents of total length n, we consider the problem of finding a longest string common to at least $d \geq 2$ of the documents. This problem is known as the *longest common substring (LCS) problem* and has a classic $\mathcal{O}(n)$ space and $\mathcal{O}(n)$ time solution (Weiner [FOCS'73], Hui [CPM'92]). However, the use of linear space is impractical in many applications. In this paper we show that for any trade-off parameter $1 \leq \tau \leq n$, the LCS problem can be solved in $\mathcal{O}(\tau)$ space and $\mathcal{O}(n^2/\tau)$ time, thus providing the first smooth deterministic time-space trade-off from constant to linear space. The result uses a new and very simple algorithm, which computes a τ-additive approximation to the LCS in $\mathcal{O}(n^2/\tau)$ time and $\mathcal{O}(1)$ space. We also show a time-space trade-off lower bound for deterministic branching programs, which implies that any deterministic RAM algorithm solving the LCS problem on documents from a sufficiently large alphabet in $\mathcal{O}(\tau)$ space must use $\Omega(n\sqrt{\log(n/(\tau\log n))/\log\log(n/(\tau\log n))})$ time.

1 Introduction

The *longest common substring (LCS) problem* is a fundamental and classic string problem with numerous applications. Given m strings $T_1, T_2, \ldots, T_m$ (the documents) from an alphabet Σ and a parameter $2 \leq d \leq m$, the LCS problem is to compute a longest string occurring in least d of the m documents. We denote such a string by LCS and use $n = \sum_{i=1}^{m} |T_i|$ to refer to the total length of the documents.

The classic text-book solution to this problem is to build the (generalized) suffix tree of the documents and find the node that corresponds to LCS [11,17,9]. While this can be achieved in linear time, it comes at the cost of using $\Omega(n)$ space[1] to store the suffix tree. In applications with large amounts of data or strict space constraints, this renders the classic solution impractical. A recent example of this challenge is automatic generation of signatures for identifying

⋆ Supported by Polish budget funds for science in 2013-2017 as a research project under the 'Diamond Grant' program.

⋆⋆ Partly supported by Dynasty Foundation.

[1] Throughout the paper, we measure space as the number of words in the standard unit-cost word-RAM model with word size $w = \Theta(\log n)$ bits.

zero-day worms by solving the LCS problem on internet packet data [1,12,16]. The same challenge is faced if the length of the longest common substring is used as a measure for plagiarism detection in large document collections.

To overcome the space challenge of suffix trees, succinct and compressed data structures have been subject to extensive research [8,13]. Nevertheless, these data structures still use $\Omega(n)$ bits of space in the worst-case, and are thus not capable of providing truly sublinear space solutions to the LCS problem.

1.1 Our Results

We give new sublinear space algorithms for the LCS problem. They are designed for the word-RAM model with word size $w = \Omega(\log n)$, and work for integer alphabets $\Sigma = \{1, 2, \ldots, \sigma\}$ with $\sigma = n^{\mathcal{O}(1)}$. Throughout the paper, we regard the output to the LCS problem as a pair of integers referring to a substring in the input documents, and thus the output fits in $\mathcal{O}(1)$ machine words.

As a stepping stone to our main result, we first show that an additive approximation of *LCS* can be computed in constant space. We use $|LCS|$ to denote the length of the longest common substring.

Theorem 1. *There is an algorithm that given a parameter τ, $1 \le \tau \le n$, runs in $\mathcal{O}(n^2/\tau)$ time and $\mathcal{O}(1)$ space, and outputs a string, which is common to at least d documents and has length at least $|LCS| - \tau + 1$.*

The solution is very simple and essentially only relies on a constant space pattern matching algorithm as a black-box. We expect that it could be of interest in applications where an approximation of *LCS* suffices.

For $\tau = 1$ we obtain the corollary:

Corollary 1. *LCS can be computed in $\mathcal{O}(1)$ space and $\mathcal{O}(n^2)$ time.*

To the best of our knowledge, this is the first constant space $\mathcal{O}(n^2)$-time algorithm for the LCS problem. Given that it is a simple application of a constant space pattern matching algorithm, it is an interesting result on its own.

Using Theorem 1 we are able to establish our main result, which gives the first deterministic time-space trade-off from constant to linear space:

Theorem 2. *There is an algorithm that given a parameter τ, $1 \le \tau \le n$, computes LCS in $\mathcal{O}(\tau)$ space and $\mathcal{O}(n^2/\tau)$ time.*

Previously, no deterministic trade-off was known except in the restricted setting of $n^{2/3} < \tau \le n$, where two of the authors showed that the problem allows an $\mathcal{O}((n^2/\tau) d \log^2 n(\log^2 n + d))$-time and $\mathcal{O}(\tau)$-space trade-off [15]. Our new solution is also strictly better than the $\mathcal{O}((n^2/\tau)\log n)$-time and $\mathcal{O}(\tau)$-space randomized trade-off, which correctly outputs *LCS* with high probability (see [15] for a description).

Finally, we prove a time-space trade-off lower bound for the LCS problem over large-enough alphabets, which remains valid even restricted to two documents.

Theorem 3. *Given two documents of total length n from an alphabet Σ of size at least n^2, any deterministic RAM algorithm, which uses $\tau \leq \frac{n}{\log n}$ space to compute the longest common substring of both documents, must use time $\Omega(n\sqrt{\log(n/(\tau \log n))/\log\log(n/(\tau \log n))})$.*

We prove the bound for non-uniform deterministic branching programs, which are known to simulate deterministic RAM algorithms with constant overhead. The lower bound of Theorem 3 implies that the classic linear-time solution is close to asymptotically optimal in the sense that there is no hope for a linear-time and $o(n/\log n)$-space algorithm that solves the LCS problem on polynomial-sized alphabets.

2 Upper Bounds

Let T be a string of length $n > 0$. Throughout the paper, we use the notation $T[i..j]$, $1 \leq i \leq j \leq n$, to denote the substring of T starting at position i and ending at position j (both inclusive). We use the shorthand $T[..i]$ and $T[i..]$ to denote $T[1..i]$ and $T[i..n]$ respectively.

A suffix tree of T is a compacted trie on suffixes of T appended with a unique letter (sentinel) \$ to guarantee one-to-one correspondence between suffixes and leaves of the tree. The suffix tree occupies linear space. Moreover, if the size of the alphabet is polynomial in the length of T, then the suffix tree can be constructed in linear time [7]. We refer to nodes of the suffix tree as *explicit nodes*, and to nodes of the underlying trie, which are not preserved in the suffix tree, as *implicit nodes*. Note that each substring of T corresponds to a unique explicit or implicit node, the latter can be specified by the edge it belongs to and its distance to the upper endpoint of the edge.

A generalized suffix tree of strings $T_1, T_2, \ldots, T_m$ is a trie on all suffixes of these strings appended with sentinels $\$_i$. It occupies linear space and for polynomial-sized alphabets can also be constructed in linear time.

Classic solution. As a warm-up, we briefly recall how to solve the LCS problem in linear time and space. Consider the generalized suffix tree of the documents $T_1, T_2, \ldots, T_m$, where leaves corresponding to suffixes of T_i, $i = 1, 2, \ldots, m$, are painted with color i. The main observation is that LCS is the label of a deepest explicit node with leaves of at least d distinct colors in its subtree. Hui [11] showed that given a tree with $\mathcal{O}(n)$ nodes where some leaves are colored, it is possible to compute the number of distinctly colored leaves below all nodes in $\mathcal{O}(n)$ time. Consequently, we can locate the node corresponding to LCS in $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ space.

2.1 Approximating LCS in Constant Space

Given a pattern and a string, it is possible to find all occurrences of the pattern in the string using constant space and linear time (see [5] and references therein). We use this result in the following $\mathcal{O}(1)$-space additive approximation algorithm.

Lemma 1. *There is an algorithm that given integer parameters ℓ, r satisfying $1 \leq \ell < r \leq n$, runs in $\mathcal{O}\big(\frac{n^2}{r-\ell}\big)$ time and constant space, and returns NO if $|LCS| < \ell$, YES if $|LCS| \geq r$, and an arbitrary answer otherwise.*

Proof. Let $S = T_1\$_1T_2\$_2 \dots T_m\$_m$ and $\tau = r - \ell$. Consider substrings $S_k = S[k\tau + 1..k\tau + \ell]$ for $k = 0, \dots, \lfloor\frac{|S|}{\tau}\rfloor$. For each S_k we use a constant-space pattern matching algorithm to count the number of documents T_i containing an occurrence of S_k. We return YES if for any S_k this value is at least d and NO otherwise.

If $|LCS| < \ell$, then any substring of S of length ℓ — in particular, any S_k — occurs in less than d documents. Consequently, in this case the algorithm will return NO. On the other hand, any substring of S of length r contains some S_k, so if $|LCS| \geq r$, then some S_k occurs in at least d documents, and in this case the algorithm will return YES. □

To establish Theorem 1 we perform a ternary search using Lemma 1 with the modification that if the algorithm returns YES, it also outputs a string of length ℓ common to at least d documents. We maintain an interval R containing $|LCS|$; initially $R = [1, n]$. In each step we set ℓ and r (approximately) in 1/3 and 2/3 of R, so that we can reduce R by $\lfloor|R|/3\rfloor$. We stop when $|R| \leq \tau$. The time complexity bound forms a geometric progression dominated by the last term, which is $\mathcal{O}(n^2/\tau)$. This concludes the proof of the following result.

Theorem 1. *There is an algorithm that given a parameter τ, $1 \leq \tau \leq n$, runs in $\mathcal{O}(n^2/\tau)$ time and $\mathcal{O}(1)$ space, and outputs a string, which is common to at least d documents and has length at least $|LCS| - \tau + 1$.*

2.2 An $\mathcal{O}(\tau)$-Space and $\mathcal{O}(n^2/\tau)$-Time Solution

We now return to the main goal of this section. Using Theorem 1, we can assume to know ℓ such that $\ell \leq |LCS| < \ell + \tau$. Organization of the text below is as follows. First, we explain how to compute LCS if $\ell = 1$. Then we extend our solution so that it works with larger values of ℓ. Here we additionally assume that the alphabet size is constant and later, in Section 2.3, we remove this assumption.

Case $\ell = 1$. From the documents $T_1, T_2, \dots, T_m$ we compose two lists of strings. First, we consider "short" documents T_j with $|T_j| < \tau$. We split them into groups of total length in $[\tau, 1 + 2\tau]$ (except for the last group, possibly). For each group we add a concatenation of the documents in this group, appended with sentinels $\$_j$, to a list $\mathcal{L}_1$. Separately, we consider "long" documents T_j with $|T_j| \geq \tau$. For each of them we add to a list $\mathcal{L}_2$ its substrings starting at positions of the form $k\tau + 1$ for integer k and in total covering T_j. These substrings are chosen to have length 2τ, except for the last whose length is in $[\tau, 2\tau]$. We assume that substrings of the same document T_j occur contiguously in $\mathcal{L}_2$ and append them with $\$_j$. The lists $\mathcal{L}_1$ and $\mathcal{L}_2$ will not be stored explicitly but will be generated on the fly while scanning the input. Note that $|\mathcal{L}_1 \cup \mathcal{L}_2| = \mathcal{O}(n/\tau)$.

Observation 4. *Since the length of LCS is between* 1 *and* τ*,* LCS *is a substring of some string* $S_k \in \mathcal{L}_1 \cup \mathcal{L}_2$*. Moreover, it is a label of an explicit node of the suffix tree of* S_k *or of a node where a suffix of some* $S_i \in \mathcal{L}_1 \cup \mathcal{L}_2$ *branches out of the suffix tree of* S_k*.*

We process candidate substrings in groups of τ, using the two lemmas below.

Lemma 2. *Consider a suffix tree of* S_k *with* τ *marked nodes (explicit or implicit). There is an* $\mathcal{O}(n)$*-time and* $\mathcal{O}(\tau)$*-space algorithm that counts the number of short documents containing an occurrence of the label of each marked node.*

Proof. For each marked node we maintain a counter $c(v)$ storing the number of short documents the label of v occurs in. Counters are initialized with zeros. We add each string $S_i \in \mathcal{L}_1$ to the suffix tree of S_k in $\mathcal{O}(\tau)$ time. By *adding* a string to the suffix tree of another string, we mean constructing the generalized suffix tree of both strings and establishing pointers from explicit nodes of the generalized suffix tree to the corresponding nodes of the original suffix tree. We then paint leaves representing suffixes of S_i: namely, we paint a leaf with color j if the corresponding suffix of S_i starts within a document T_j (remember that S_i is a concatenation of short documents). Then the label of a marked node occurs in T_j iff this node has a leaf of color j in its subtree. Using Hui's algorithm we compute the number of distinctly colored leaves in the subtree of each marked node v and add this number to $c(v)$. After updating the counters, we remove colors and newly added nodes from the tree. Since all sentinels in the strings in $\mathcal{L}_1$ are distinct, the algorithm is correct. It runs in $\mathcal{O}(|\mathcal{L}_1|\tau+\tau) = \mathcal{O}(n)$ time. □

Lemma 3. *Consider a suffix tree of* S_k *with* τ *marked nodes (explicit or implicit). There is an* $\mathcal{O}(n)$*-time and* $\mathcal{O}(\tau)$*-space algorithm that counts the number of long documents containing an occurrence of the label of each marked node.*

Proof. For each of the marked nodes we maintain a variable $c(v)$ counting the documents where the label of v occurs. A single document might correspond to several strings S_i, so we also keep an additional variable $m(v)$, which prevents increasing $c(v)$ several times for a single document. As in Lemma 2, we add each string $S_i \in \mathcal{L}_2$ to the suffix tree of S_k. For each marked node v whose subtree contains a suffix of S_i ending with $\$_j$, we compare $m(v)$ with j. We increase $c(v)$ only if $m(v) \neq j$, also setting $m(v) = j$ to prevent further increases for the same document. Since strings corresponding to the T_j occur contiguously in $\mathcal{L}_2$, the algorithm is correct. Its running time is $\mathcal{O}(|\mathcal{L}_2|\tau + \tau) = \mathcal{O}(n)$. □

Let $\mathcal{L} = \mathcal{L}_1 \cup \mathcal{L}_2$. If LCS is a substring of $S_k \in \mathcal{L}$, we can find it as follows: we construct the suffix tree of S_k, mark its explicit nodes and nodes where suffixes of $S_i \in \mathcal{L}$ $(i \neq k)$ branch out, and determine the deepest of them which occurs in at least d documents. Repeating for all $S_k \in \mathcal{L}$, this allows us to determine LCS. To reduce the space usage to $\mathcal{O}(\tau)$, we use Lemma 2 and Lemma 3 for batches of $\mathcal{O}(\tau)$ marked nodes in the suffix tree of S_k at a time. Labels of all marked node are also labels of explicit nodes in the generalized suffix tree of $T_1, \ldots, T_m$. In order to achieve good running time we will make sure that marked nodes have, over all $S_k \in \mathcal{L}$, distinct labels. This will imply that we use Lemma 2 and Lemma 3 only $\mathcal{O}(n/\tau)$ times, and hence spend $\mathcal{O}(n^2/\tau)$ time overall.

We consider each of the substrings $S_k \in \mathcal{L}$ in order. We start by constructing a suffix tree for S_k. To make sure the labels of marked nodes are distinct, we shall exclude some (explicit and implicit) nodes of S_k. Each node is going to be excluded together with all its ancestors or descendants, so that it is easy to test whether a particular node is excluded. (It suffices to remember the highest and the lowest non-excluded node on each edge, if any, $\mathcal{O}(\tau)$ nodes in total.)

First of all, we do not need to consider substrings of $S_1, \ldots, S_{k-1}$. Therefore we add each of strings $S_1, S_2, \ldots, S_{k-1}$ to the suffix tree (one by one) and exclude nodes common to S_k and these strings from consideration. Note that in this case a node is excluded with all its ancestors.

Then we consider all strings $S_k, S_{k+1}, S_{k+2}, \ldots$ in turn. For each string we construct the generalized suffix tree of S_k and the current S_i and iterate over explicit nodes of the tree whose labels are substrings of S_k. If a node has not been excluded, we mark it. Once we have τ marked nodes (and if any marked nodes are left at the end), we apply Lemma 2 and Lemma 3. If the label of a marked node occurs in at least d documents, then we can exclude the marked node and all its ancestors. Otherwise, we can exclude it with all its descendants.

Recall that LCS is a label of one of the explicit inner nodes of the generalized suffix tree of $T_1, T_2, \ldots, T_m$, i.e., there are $\mathcal{O}(n)$ possible candidates for LCS. Moreover, we are only interested in candidates of length at most τ, and each such candidate corresponds to an explicit node of the generalized suffix tree of a pair of strings from $\mathcal{L}$. The algorithm process each such candidate exactly once due to node exclusion. Thus, its running time is $\mathcal{O}(\frac{n}{\tau}n + \tau) = \mathcal{O}(n^2/\tau)$. At any moment it uses $\mathcal{O}(\tau)$ space.

General Case. If $\ell < 10\tau$ we can still use the technique above, adjusting the multiplicative constants in the complexity bounds. Thus, we can assume $\ell > 10\tau$.

Documents shorter than ℓ cannot contain LCS and we ignore them. For each of the remaining documents T_j we add to a list $\mathcal{L}$ its substrings starting at positions of the form $k\tau+1$ for integer k and in total covering T_j. The substrings are chosen to have length $\ell+2\tau$, except for the last whose length is in the interval $[\ell, \ell+2\tau]$. Each substring is appended with $\$_j$, and we assume that the substrings of the same document occur contiguously.

Observation 5. *Since the length of LCS is between ℓ and $\ell + \tau$, LCS is a substring of some string $S_k \in \mathcal{L}$. Moreover, it is the label of a node of the suffix tree of S_k where a suffix of another string $S_i \in \mathcal{L}$ branches out. (We do not need to consider explicit nodes of the suffix tree as there are no short documents.)*

As before, we consider strings $S_k \in \mathcal{L}$ in order and check all candidates which are substrings of S_k but not any S_i for $i < k$. However, in order to make the algorithm efficient, we replace all strings S_i, including S_k, with strings $r_k(S_i)$, each of length $\mathcal{O}(\tau)$. To define the mapping r_k we first introduce some necessary notions.

We say that $S[1..p]$ is a period of a string S if $S[i] = S[i+p]$, $1 \le i \le |S| - p$. The length of the shortest period of S is denoted as $\mathrm{per}(S)$. We say that a string S is *primitive* if its shortest period is not a proper divisor of $|S|$. Note that $\rho = S[1..\mathrm{per}(S)]$ is primitive and therefore satisfies the following lemma:

Lemma 4 (Primitivity Lemma [6]). *Let ρ be a primitive string. Then ρ has exactly two occurrences in a string $\rho\rho$.*

Let $Q_k = S_k[1 + 2\tau..\ell]$; note that $|Q_k| = \ell - 2\tau \ge 8\tau$. Let $\mathrm{per}(Q_k)$ be the length of the shortest period ρ of Q. If $\mathrm{per}(Q_k) > 4\tau$, we define $Q'_k = \#$, where $\#$ is a special letter that does not belong to the main alphabet. Otherwise Q_k can be represented as $\rho^t\rho'$, where ρ' is a prefix of ρ. We set $Q'_k = \rho^{t'}\rho'$ for $t' \le t$ chosen so that $8\tau \le |Q'_k| < 12\tau$. For any string S we define $r_k(S) = \varepsilon$ if S does not contain Q_k, and a string obtained from S by replacing the first occurrence of Q_k with Q'_k otherwise. Below we explain how to compute Q'_k.

Lemma 5. *One can decide in linear time and constant space if* $\mathrm{per}(Q_k) \le 4\tau$ *and provided that this condition holds, compute* $\mathrm{per}(Q_k)$.

Proof. Let P be the prefix of Q_k of length $\lceil |Q_k|/2 \rceil$ and p be the starting position of the second occurrence of P in Q_k, if any. The position p can be found in $\mathcal{O}(|Q_k|)$ time by a constant-space pattern matching algorithm.

We claim that if $\mathrm{per}(Q_k) \le 4\tau \le \lceil |Q_k|/2 \rceil$, then $p = \mathrm{per}(Q_k)+1$. Observe first that in this case P occurs at a position $per(Q_k)+1$, and hence $p \le \mathrm{per}(Q_k)+1$. Furthermore, p cannot be smaller than $\mathrm{per}(Q_k)+1$, because otherwise $\rho = Q_k[1..\mathrm{per}(Q_k)]$ would occur in $\rho\rho = Q_k[1..2\,\mathrm{per}(Q_k)]$ at the position p. The shortest period ρ is primitive, so this is a contradiction with Lemma 4.

The algorithm compares p and $4\tau+1$. If $p \le 4\tau+1$, it uses letter-by-letter comparison to determine whether $Q_k[1..p-1]$ is a period of Q_k. If so, by the discussion above $\mathrm{per}(Q_k) = p - 1$, and the algorithm returns it. Otherwise $\mathrm{per}(Q_k) > 4\tau$. The algorithm runs in $\mathcal{O}(|Q_k|)$ time and uses constant space. □

Fact 6. *Suppose that a string S, $|S| \le |Q_k| + 4\tau$, contains Q_k as a substring. Then*

(a) replacing with Q'_k any occurrence of Q_k in S results in $r_k(S)$,
(b) replacing with Q_k any occurrence of Q'_k in $r_k(S)$ results in S.

Proof. We start with (a). Let i and i' be the positions of the first and last occurrence of Q_k in S. We have $1 \le i \le i' \le |S| - |Q_k| + 1$, so $i' - i \le |S| - |Q_k| \le 4\tau$. If $\mathrm{per}(Q_k) > 4\tau$ this implies that $i' - i = 0$, or, in other words, that Q_k has just one occurrence in S.

On the other hand, if $\mathrm{per}(Q_k) \le 4\tau$, we observe that $i' - i \le 4\tau = 8\tau - 4\tau \le |Q_k| - \mathrm{per}(Q_k)$. Therefore the string $\rho = S[i'..i' + \mathrm{per}(Q_k) - 1]$ fits within $Q_k = S[i..i + |Q_k| - 1]$. It is primitive and Lemma 4 implies that ρ occurs in $\rho^t\rho'$ only t times, so $i' = i + j \cdot per(Q_k)$ for some integer $j \le t$. Therefore all occurrences of Q_k lie in the substring of S of the form $\rho^s\rho'$ for some $s \ge t$. Thus, replacing any of these occurrences with Q'_k leads to the same result, $r_k(S)$.

Now, let us prove (b). Note that if we replace an occurrence of Q'_k in $r_k(S)$ with Q_k, by (a) we obtain a string S' such that $r_k(S') = r_k(S)$. Moreover all such strings S' can be obtained by replacing some occurrence of Q'_k, in particular this is true for S.

If $\text{per}(Q_k) > 4\tau$, since # does not belong to the main alphabet, Q'_k has exactly one occurrence in $r_k(S)$ and the statement holds trivially. For the other case we proceed as in the proof of (a) showing that all occurrences of Q'_k are in fact substrings of a longer substring of S of the form $\rho^{s'}\rho'$ for some $s' \geq t'$. □

Lemma 6. *Consider strings P and S, such that $|S| \leq |Q_k| + 4\tau$ and P contains Q_k as a substring. Then P occurs in S at position p if and only if $r_k(P)$ occurs in $r_k(S)$ at position p.*

Proof. First, assume that P occurs in S at a position p. This induces an occurrence of Q_k in S within the occurrence of P, and replacing this occurrence of Q_k with Q'_k gives $r_k(S)$ by Theorem 6(a). This replacement also turns the occurrence of P at the position p into an occurrence of $r_k(P)$.

Now, assume $r_k(P)$ occurs in $r_k(S)$ at the position p. Since $r_k(P) \neq \varepsilon$, this means that $r_k(S) \neq \varepsilon$ and that Q'_k occurs in $r_k(S)$ (within the occurrence of $r_k(P)$). By Theorem 6(b) replacing this occurrence of Q'_k with Q_k turns $r_k(S)$ into S and the occurrence of $r_k(P)$ at the position p into an occurrence of P. □

Observe that applied for $S = S_k$, Lemma 6 implies that r_k gives a bijection between substrings of S_k of length $\geq \ell = |Q_k| + 2\tau$ and substrings of $r_k(S_k)$ of length $\geq |Q'_k| + 2\tau$. Moreover, it shows that any substring of S_k of length $\geq \ell$ occurs in S_i iff the corresponding substring of $r_k(S_k)$ occurs in $r_k(S_i)$.

This lets us apply the technique described in the previous section to find *LCS* provided that it occurs in S_k but not S_i with $i < k$. Strings $r_k(S_i)$ are computed in parallel with a constant-space pattern matching algorithm for a pattern Q_k in the documents of length ℓ or more, which takes $\mathcal{O}(n)$ time in total. The list $\mathcal{L}$ is composed $r_k(S_i)$ obtained from long documents, and we use Lemma 3 to compute the number of documents each candidates occurs in.

Compared to the arguments of the previous section, we additionally exclude nodes of depth less than $|Q'_k| + 2\tau$ to make sure that each marked node is indeed $r_k(P)$ for some substring P of S_k of length at least $\ell = |Q_k| + 2\tau$. This lets us use the amortization by the number of explicit nodes in the generalized suffix tree of $T_1, \ldots, T_m$. More precisely, if a node with label $r_k(P)$ is marked, we charge P, which is guaranteed to be explicit in the generalized suffix tree. This implies $\mathcal{O}(n^2/\tau)$-time and $\mathcal{O}(\tau)$-space bounds.

2.3 Large Alphabets

In this section we describe how to adapt our solution so that it works for alphabets of size $n^{\mathcal{O}(1)}$. Note that we have used the constant-alphabet assumption only to make sure that suffix trees can be efficiently constructed. If the alphabet is not constant, a suffix tree of a string can be constructed in linear time plus the time of sorting its letters [7]. If $\tau > \sqrt{n}$, the size of the alphabet is $n^{\mathcal{O}(1)} = \tau^{\mathcal{O}(1)}$ and hence any suffix tree used by the algorithm can be constructed in $\mathcal{O}(\tau)$ time.

Suppose now that $\tau \leq \sqrt{n}$ and $\ell = 1$. Our algorithm uses suffix trees in a specific pattern: in a single phase it builds the suffix tree of S_k and then constructs the generalized suffix tree of S_k and S_i for each i. Note that the algorithm only needs information about the nodes of the suffix tree of S_k, the nodes where suffixes of $S_i \in \mathcal{L}$ branch out, and leaves of the generalized suffix tree. None of these changes if we replace each letter of Σ occurring in S_i, but not in S_k, with a special letter which does not belong to Σ.

Thus our approach is as follows: first we build a deterministic dictionary, mapping letters of S_k to integers of magnitude $\mathcal{O}(|S_k|) = \mathcal{O}(\tau)$ and any other letter of the main alphabet to the special letter. The dictionary can be constructed in $\mathcal{O}(\tau \log^2 \log \tau)$ time [14,10]. Then instead of building the generalized suffix tree of S_k and S_i we build it for the corresponding strings with letters mapped using the dictionary. In general, when ℓ is large, we apply the same idea with $r_k(S_k)$ and $r_k(S_i)$ instead of S_k and S_i respectively.

In total, the running time is $\mathcal{O}(n^2/\tau + n \log^2 \log \tau)$. For $\tau \leq \sqrt{n}$ the first term dominates the other, i.e. we obtain an $\mathcal{O}(n^2/\tau)$-time solution.

Theorem 2. *There is an algorithm that given a parameter τ, $1 \leq \tau \leq n$, computes LCS in $\mathcal{O}(n^2/\tau)$ time using $\mathcal{O}(\tau)$ space.*

3 A Time-Space Trade-Off Lower Bound

Given n elements over a domain D, the *element distinctness problem* is to decide whether all n elements are distinct. Beame et al. [3] showed that if $|D| \geq n^2$, then any RAM algorithm solving the element distinctness problem in τ space, must use at least $\Omega(n\sqrt{\log(n/(\tau \log n))/\log\log(n/(\tau \log n))})$ time.[1]

The element distinctness (ED) problem can be seen as a special case of the LCS problem where we have $m = n$ documents of length 1 and want to find the longest string common to at least $d = 2$ documents. Thus, the lower bound for ED also holds for this rather artificial case of the LCS problem. Below we show that the same bound holds with just $m = 2$ documents. The main idea is to show an analogous bound for a two-dimensional variant of the element distinctness problem, which we call the *element bidistinctness problem.* The LCS problem on two documents naturally captures this problem. The steps are similar to those for the ED lower bound by Beame et al. [3], but the details differ. We start by introducing the necessary definitions of branching programs and embedded rectangles. We refer to [3] for a thorough overview of this proof technique.

Branching Programs. A n-variate branching program $\mathcal{P}$ over domain D is an acyclic directed graph with the following properties: (1) there is a unique source node denoted s, (2) there are two sink nodes, one labelled by 0 and one labelled

[1] Note that in [3,4] the space consumption is measured in bits. The version of RAM used there is unit-cost with respect to time and log-cost with respect to space.

by 1, (3) each nonsink node v is assigned an index $i(v) \in [1,n]$ of a variable, and (4) there are exactly $|D|$ arcs out of each nonsink node, labelled by distinct elements of D. A branching program is executed on an input $x \in D^n$ by starting at s, reading the variable $x_{i(s)}$ and following the unique arc labelled by $x_{i(s)}$. This process is continued until a sink is reached and the output of the computation is the label of the sink. For a branching program $\mathcal{P}$, we define its *size* as the number of nodes, and its *length* as the length of the longest path from s to a sink node.

Lemma 7 (see page 2 of [4]). *If $f : D^n \to \{0,1\}$ has a word-RAM algorithm with running time $T(n)$ using $S(n)$ w-bit words, then there exists an n-variate branching program $\mathcal{P}$ over D computing f, of length $O(T(n))$ and size $2^{O(wS(n)+\log n)}$.*

Embedded Rectangles. If $A \subseteq [1,n]$, a point $\tau \in D^A$ (i.e. a function $\tau : A \to D$) is called a *partial input* on A. If τ_1, τ_2 are partial inputs on $A_1, A_2 \in [1,n]$, $A_1 \cap A_2 = \emptyset$, then $\tau_1\tau_2$ is the partial input on $A_1 \cup A_2$ agreeing with τ_1 on A_1 and with τ_2 on A_2. For sets $B \subseteq D^{[1,n]}$ and $A \subseteq [1,n]$ we define B_A, the *projection* of B onto A, as the set of all partial inputs on A which agree with some input in B. An embedded rectangle R is a triple (B, A_1, A_2), where A_1 and A_2 are disjoint subsets of $[1,n]$, and $B \subseteq D^{[1,n]}$ satisfies: (i) $B_{[1,n]\setminus A_1 \cup A_2}$ consists of a single partial input σ, (ii) if $\tau_1 \in B_{A_1}$, and $\tau_2 \in B_{A_2}$, then $\tau_1\tau_2\sigma \in B$. For an embedded rectangle $R = (B, A_1, A_2)$, and $j \in \{1,2\}$ we define:

$$m_j(R) = |A_j| \qquad m(R) = \min(m_1(R), m_2(R))$$
$$\alpha_j(R) = |B_{A_j}|/|D|^{|A_j|} \qquad \alpha(R) = \min(\alpha_1(R), \alpha_2(R))$$

Given a small branching program $\mathcal{P}$ it can be shown that $\mathcal{P}^{-1}(1)$, the set of all YES-inputs, contains a relatively large embedded rectangle. Namely,

Lemma 8 (Corollary 5.4 (i) [3]). *Let $k \geq 8$ be an integer, $q \leq 2^{-40}k^{-8}$, $n \geq r \geq q^{-5k^2}$. Let $\mathcal{P}$ be a n-variate branching program over domain D of length at most $(k-2)n$ and size 2^S. Then there is an embedded rectangle R contained in $\mathcal{P}^{-1}(1)$ satisfying $m(R) = m_1(R) = m_2(R) \geq q^{2k^2}n/2$ and $\alpha(R) \geq 2^{-q^{1/2}m(R)-Sr}|\mathcal{P}^{-1}(1)|/|D^n|$.*

Element Bidistinctness. We say that two elements $x = (x_1, x_2)$ and $y = (y_1, y_2)$ of the Cartesian product $D \times D$ are *bidistinct* if both $x_1 \neq y_2$ and $x_2 \neq y_1$. The element bidistinctness function $EB : (D \times D)^n \to \{0,1\}$ is defined to be 1 iff for every pair of indices $1 \leq i, j \leq n$ the i-th and j-th pair are bidistinct. Note that computing EB for $(s_1, t_1), \ldots, (s_n, t_n)$ is equivalent to deciding if $LCS(s_1 \ldots s_n, t_1 \ldots t_n) \geq 1$. Thus the problem of computing the longest common substring of two strings over $\Sigma = D$ is at least as hard as the EB problem. Below we show a time-space trade-off lower bound for element bidistinctness.

Lemma 9. *If $|D| \geq 2n^2$, at least a fraction $1/e$ of inputs belong to $EB^{-1}(1)$.*

Proof. The size of $EB^{-1}(1)$ is at least $(|D|-1)^2 \cdot (|D|-2)^2 \cdot \ldots \cdot (|D|-n)^2$. Hence, $|EB^{-1}(1)| = |D|^{2n} \prod_{i=1}^{n} (1-\frac{i}{|D|})^2 \geq |D|^{2n}(1-\frac{1}{2n})^{2n} \geq |D|^{2n}/e$. $\square$

Lemma 10. *For any embedded rectangle $R = (B, A_1, A_2) \subseteq EB^{-1}(1)$ we have $\alpha(R) \leq 2^{-2m(R)}$.*

Proof. Let S_j be the subset of $D \times D$ that appear on indices in A_j, i.e., $S_j = \bigcup_{\tau \in B_{A_j}} \{\tau(i) : i \in A_j\}$, $j = 1, 2$. Clearly, all elements in S_1 must be bidistinct from all elements in S_2. If this was not the case B would contain a vector with two non-bidistinct elements of $D \times D$. We will prove that $\min(|S_1|, |S_2|) \leq |D|^2/4$. Let us first argue that this implies the lemma. For $j = 1$ or $j = 2$, we get that $|B_{A_j}| \leq (|D|^2/4)^{|A_j|}$, and thus $\alpha_j(R) \leq (|D|^2/4)^{|A_j|}/(|D|^2)^{|A_j|} = 4^{-|A_j|} \leq 4^{-m(R)} = 2^{-2m(R)}$.

It remains to prove that $\min(|S_1|, |S_2|) \leq |D|^2/4$. For $j \in \{1, 2\}$ let X_j and Y_j denote the set of first and second coordinates that appear in S_j. Note that by bidistinctness $X_1 \cap Y_2 = X_2 \cap Y_1 = \emptyset$. Moreover $|S_j| \leq |X_j||Y_j|$ and therefore $\sqrt{|S_j|} \leq \sqrt{|X_j||Y_j|} \leq \frac{1}{2}(|X_j| + |Y_j|)$. Consequently $2(\sqrt{|S_1|} + \sqrt{|S_2|}) \leq |X_1| + |Y_1| + |X_2| + |Y_2| = (|X_1| + |Y_2|) + (|Y_1| + |X_2|) \leq 2|D|$ and thus $\min(\sqrt{|S_1|}, \sqrt{|S_2|}) \leq |D|/2$, i.e. $\min(|S_1|, |S_2|) \leq |D|^2/4$ as claimed. $\square$

Theorem 7. *Any n-variate branching program $\mathcal{P}$ of length T and size 2^S over domain D, $|D| \geq 2n^2$, which computes the element bidistinctness function EB, requires $T = \Omega(n\sqrt{\log(n/S)/\log\log(n/S)})$ time.*

Proof. The proof repeats the proof of Theorem 6.13 [3]. We restore the details omitted in [3] for the sake of completeness. Suppose that the length of $\mathcal{P}$ is $T = (k-2)n/2$ and size 2^S. Apply Lemma 8 with $q = 2^{-40}k^{-8}$ and $r = \left\lceil q^{-5k^2} \right\rceil$. We then obtain an embedded rectangle $R \in EB^{-1}(1)$ such that $m(R) \geq q^{2k^2}n/4$ and $\alpha(R) \geq 2^{-q^{1/2}m(R)-Sr}/e = 2^{-q^{1/2}m(R)-Sr-\log e}$. From Lemma 10 we have $2^{-2m(R)} \geq 2^{-q^{1/2}m(R)-Sr-\log e}$ and thus $Sr \geq m(R)(2-q^{1/2}) - \log e \geq m(R)/2$. Consequently, $S \geq q^{2k^2}n/(8r)$. Remember that $q = 2^{-40}k^{-8}$ and $r = \left\lceil q^{-5k^2} \right\rceil$, which means that $\mathcal{P}$ requires at least $k^{-ck^2}n$ space for some constant $c > 0$. That is, $k^{ck^2} \geq n/S$, which implies $k = \Omega(\sqrt{\log(n/S)/\log\log(n/S})$. Substituting $k = 2T/n + 2$, we obtain the claimed bound. $\square$

Corollary 2. *Any deterministic RAM algorithm that solves the element bidistinctness (EB) problem on inputs in $(D \times D)^n$, $|D| \geq 2n^2$, using $\tau \leq \frac{n}{\log n}$ space, must use at least $\Omega(n\sqrt{\log(n/(\tau \log n))/\log\log(n/(\tau \log n))})$ time.*

Corollary 3 (Theorem 3). *Given two documents of total length n from an alphabet Σ of size at least n^2, any deterministic RAM algorithm, which uses $\tau \leq \frac{n}{\log n}$ space to compute the longest common substring of both documents, must use time $\Omega(n\sqrt{\log(n/(\tau \log n))/\log\log(n/(\tau \log n))})$.*

4 Conclusions

The main problem left open by our work is to settle the optimal time-space product for the LCS problem. While it is tempting to guess that the answer lies in the vicinity of $\Theta(n^2)$, it seems really difficult to substantially improve our lower bound. Strong time-space product lower bounds have so far only been established in weaker models (e.g., the comparison model) or for multi-output problems (e.g., sorting an array, outputting its distinct elements and various pattern matching problems). Proving an $\Omega(n^2)$ time-space product lower bound in the RAM model for *any* problem where the output fits in a constant number of words (e.g., the LCS problem) is a major open problem.

References

1. Afek, Y., Bremler-Barr, A., Landau Feibish, S.: Automated signature extraction for high volume attacks. In: Proc. 9th ANCS, pp. 147–156 (2013)
2. Beame, P.: Clifford, R., Machmouchi, W.: Element Distinctness, Frequency Moments, and Sliding Windows. In: Proc. 54th FOCS, pp. 290–299 (2013)
3. Beame, P., Saks, M., Sun, X., Vee, E.: Time-Space Trade-Off Lower Bounds for Randomized Computation of Decision Problems. Journal of the ACM 50(2), 154–195 (2003)
4. Borodin, A., Cook, S.A.: A Time-Space Tradeoff for Sorting on a General Sequential Model of Computation. SIAM Journal on Computing 11(2), 287–297 (1982)
5. Breslauer, D., Grossi, R., Mignosi, F.: Simple Real-Time Constant-Space String Matching. Theor. Comput. Sci. 483, 2–9 (2013)
6. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on Strings. Cambridge University Press (2007)
7. Farach-Colton, M.: Optimal Suffix Tree Construction with Large Alphabets. In: Proc. 38th FOCS, pp. 137–143 (1997)
8. Grossi, R., Vitter, J.S.: Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. SIAM Journal on Computing 35(2), 378–407 (2005)
9. Gusfield, D.: Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge University Press (1997)
10. Han, Y.: Deterministic sorting in $O(n \log \log n)$ time and linear space. Journal of Algorithms 50(1), 96–105 (2004)
11. Hui, L.C.K.: Color Set Size Problem with Applications to String Matching. In: Apostolico, A., Galil, Z., Manber, U., Crochemore, M. (eds.) CPM 1992. LNCS, vol. 644, pp. 230–243. Springer, Heidelberg (1992)
12. Kreibich, C., Crowcroft, J.: Honeycomb: Creating Intrusion Detection Signatures Using Honeypots. ACM SIGCOMM Comput. Commun. Rev. 34(1), 51–56 (2004)
13. Navarro, G., Mäkinen, V.: Compressed Full-Text Indexes. ACM Computing Surveys (CSUR) 39(1), 2 (2007)
14. Ružić, M.: Constructing Efficient Dictionaries in Close to Sorting Time. In: Aceto, L., Damgrard, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part I. LNCS, vol. 5125, pp. 84–95. Springer, Heidelberg (2008)

15. Starikovskaya, T., Vildhøj, H.W.: Time-Space Trade-Offs for the Longest Common Substring Problem. In: Fischer, J., Sanders, P. (eds.) CPM 2013. LNCS, vol. 7922, pp. 223–234. Springer, Heidelberg (2013)
16. Wang, K., Cretu, G.F., Stolfo, S.J.: Anomalous Payload-Based Worm Detection and Signature Generation. In: Valdes, A., Zamboni, D. (eds.) RAID 2005. LNCS, vol. 3858, pp. 227–246. Springer, Heidelberg (2006)
17. Weiner, P.: Linear Pattern Matching Algorithms. In: Proc. 14th FOCS (SWAT), pp. 1–11 (1973)

Nested Set Union

Daniel H. Larkin[1,⋆] and Robert E. Tarjan[1,2]

[1] Princeton University Department of Computer Science
{dhlarkin,ret}@cs.princeton.edu
[2] MSR SVC

Abstract. We consider a version of the classic disjoint set union (union-find) problem in which there are two partitions of the elements, rather than just one, but restricted such that one partition is a refinement of the other. We call this the *nested set union problem*. This problem occurs in a new algorithm to find dominators in a flow graph. One can solve the problem by using two instances of a data structure for the classical problem, but it is natural to ask whether these instances can be combined. We show that the answer is yes: the nested problem can be solved by extending the classic solution to support two nested partitions, at the cost of at most a few bits of storage per element and a small constant overhead in running time. Our solution extends to handle any constant number of nested partitions.

Keywords: data structures, disjoint set union, union-find.

1 Introduction

The *disjoint set union problem* is to maintain a partition under set union. More precisely, the problem is to maintain a collection of disjoint sets (the parts of the partition), each with a canonical element called its *root*, under an intermixed sequence of UNITE and FIND operations, defined as follows:

FIND (x): Return the root of the set containing x.

UNITE (x, y): If x and y are in the same set, return **false**; otherwise, unite the two sets containing x and y, choose a root for the new set, and return **true**.

Initially each set is a singleton whose root is its only element. During a UNITE, the implementation is free to choose any element in the set as the new root. The classic solution to this problem is to use a *compressed tree* [6]. Each set is represented by a rooted tree with a node for each element of the set; the tree root is the canonical element. Each node x has a pointer $x.p$ to its parent. The root points to itself. This representation makes implementing FIND and UNITE very simple. To execute FIND (x), follow parent pointers from x until reaching the root. The path of ancestors from x to the root is called the FIND *path*. To execute UNITE (x, y), first FIND the roots of the sets containing x and y. If they are the same, return **false**. Otherwise make one the parent of the other and return **true**.

⋆ Research at Princeton University partially supported by NSF grant CCF-0832797.

This representation allows for a great deal of flexibility, and a rich body of literature explores the design space. The tree may be restructured freely during the traversal of a FIND path, and a variety of efficient methods for compacting each FIND path have been proposed. Furthermore, the root of a new set can be chosen arbitrarily, and several methods of making this choice, called *linking rules*, have been proposed. The most efficient algorithms combine a good form of compaction with a good linking rule.

Among the good compaction methods, arguably the simplest conceptually is *compression*, which replaces the parent of each node along the FIND path by the root [8]. Other good methods include *splitting* [13], *halving* [14], and *splicing* [2,12].

Two simple linking rules are *naïve linking*, which when doing UNITE (x, y) chooses as the new root the root of the old tree containing x, and *linking by index* which requires that the elements be totally ordered and chooses as the new root the larger of the two old roots. Two more-efficient rules are *linking by size*, which chooses as the new root the root of the tree containing more nodes, and *linking by rank*, which maintains a non-negative integer rank for each node, initially 0, and chooses as the new root the old root of larger rank, increasing the rank of the new root by 1 if there is a tie. All these rules are deterministic. An efficient randomized rule is *randomized linking*, which totally orders the elements by choosing a fixed permutation uniformly at random and then does linking by index.

Tarjan and van Leeuwen [12] proved that compression, splitting, halving, or splicing in combination with linking by rank or compression, splitting, or halving in combination with linking by size takes $\mathrm{O}\left(m\alpha\left(n, {}^{m}/_{n}\right)\right)$ time to execute $m \geq n$ operations on sets containing a total of n elements, where α is a functional inverse of Ackermann's function. Goel et al. [7] recently obtained the same bound in expectation for randomized linking in combination with any of the four compaction methods. We will reference their paper extensively, and henceforth refer to it as GKLT. These results match the lower bound of Fredman and Saks [5] and thus are optimal to within a constant factor.

Some applications require the maintenance of more than one partition of the same collection of elements. An example is a recent algorithm of Fraczak et al. [4] for finding dominators in a flow graph, which needs to maintain two nested partitions of the node set. Of course one can maintain each partition using a separate data structure, but it is natural to ask whether one can save space by using a single compressed tree to represent both partitions, without sacrificing the inverse-Ackermann-function amortized time bound. We show that the answer is yes.

Let us define the problem precisely. For simplicity we consider the case of two nested partitions, but it is easy to extend our results to any fixed nesting depth. Given a set of n elements, we wish to maintain two nested partitions of the elements, the *coarse partition* and the *fine partition*, under intermixed FIND and UNITE operations on either partition. The only requirement is that the UNITE operations must maintain the nesting of the partitions: each fine UNITE must combine sets contained in the same set of the coarse partition. Initially both partitions are the same, the partition into singletons.

Our solution to this problem represents both partitions by a single compressed forest, requiring one parent pointer and a few extra bits of information per node. Each coarse set is represented by a tree whose nodes are the elements of the set, with the root of the tree equal to the root of the set. We call such a tree a *coarse tree*. Each fine set is represented by a subtree whose nodes are the elements of the set, with the root of the subtree equal to the root of the set. We call such a subtree a *fine tree*; we call its root a *subroot*. The root of a coarse tree is also the root of a fine tree and hence also a subroot.

To complete the solution, we need algorithms for coarse and fine FIND and coarse and fine UNITE. We want to adapt algorithms for the one-level case, specifically path compaction methods and linking rules, to the two-level case. In doing this, we encounter two technical challenges, in the implementation of coarse FIND and that of fine UNITE. We cannot implement coarse FIND using any of the standard compaction methods because they can destroy the partition of the coarse tree into fine trees. To overcome this problem we do coarse FIND operations using two-level compression. First, we compress each part of the coarse FIND path that is within a single fine tree. This compresses the coarse FIND path into a path of subroots. Then we compress the path of subroots.

We call this method *segmented compression*. The idea of first compressing subpaths between subroots and then compressing the resulting path of subroots was used by Farrow in an unpublished algorithm for combining values along paths in rooted trees subject to arc additions. For a definition and some results on this problem see [3,11]. In that application, there is no freedom in the way links are done. Farrow introduced a complicated definition of subroots and only managed to prove an $\mathrm{O}(\log^* n)$ amortized time bound per operation. In our application, on the other hand, we can adapt standard linking rules, and we are able to obtain an inverse-Ackermann time bound per operation. Whether Farrow's algorithm has a $\mathrm{o}(\log^* n)$ bound is an intriguing open problem.

The second challenge arises from the fine UNITE operations. If a fine UNITE is done on two fine sets whose subroots are children of the root of a coarse set, then making one of the subroots a child of the other uncompacts the coarse tree. This complicates the extension of standard linking rules to nested partitions, and it makes the analysis of the resulting algorithms much more than a straightforward extension of existing results.

We prove an asymptotically optimal bound for segmented compression used with an extended version of linking by rank. Our analysis is unconventional. Most analyses of disjoint set union structures use the "union forest" idea, which measures the effect of compactions on a forest built by the UNITE operations without any compaction. The fine UNITE operations prevent the use of this idea, requiring us to cope with non-fixed ranks and non-ancestral parent changes.

The remainder of our paper contains five sections. We begin in Section 2 by detailing our algorithms. In Section 3 we establish key properties of node ranks. In Section 4, we use these rank properties and adapt the analysis from GKLT of standard compression to prove amortized bounds for segmented compression. We conclude in Section 5 with further discussion of addional results related to the design space, the extension to deeper nesting, and open problems.

2 Algorithms

In order to guarantee that the operations remain asymptotically efficient, we will adapt clever forms of linking and path compaction from other disjoint set union structures. In particular, we will adapt segmented compression and linking by rank. We call the root of each set in the coarse partition a *root* and one of a set in the fine partition a *subroot*. One simple property of our algorithms is that all roots are also subroots, but not vice-versa. Another is that fine sets are represented by contiguous subtrees.

For a node x, we denote its parent by $x.p$ and its rank by $x.r$. We also store a flag, $x.\gamma$ which is set to **true** if x is a subroot and **false** otherwise. We have one additional field for the purpose of analysis and which is not maintained by the algorithm. The *maximum rank* of a node, denoted $x.r_m$, is equal to the maximum value of $x.r$ over the sequence of operations.

Segmented compression was initially developed by Farrow to provide an efficient algorithm for path evaluation problems on unbalanced trees [3]. Farrow's work was quite complicated. There was no natural notion of a subroot in his application, so he used artificial functions to calculate subroot status. Furthermore, the path evaluation problem does not allow for the use of intelligent linking rules (and hence leads to unbalanced trees). His algorithm was designed to achieve an amortized $\mathrm{O}(\log^* n)$ bound per operation rather than the $\mathrm{O}(\log n)$ bound achieved by standard compression. So in truth, we are borrowing just the skeleton of his algorithm and will be providing a fully new analysis of it in this different setting.

Segmented compression can be viewed as a two-step process, though its implementation will be more direct. First, each non-subroot is made to point to its nearest ancestral subroot. Next, each subroot is made to point to the root of the tree. By default, segmented compression returns the roots of both the coarse and fine set containing the node to be found. All coarse FIND operations will be done using segmented compression, including those within UNITE operations. See Algorithms 1 and 2 for an implementation of FIND with segmented compression and Figure 1 for a visual example.

While one could use segmented compression for fine FIND operations, this is over-kill in a certain sense. We use a more local implementation for fine FIND operations. We perform standard compression within the fine subtree containing the node to be found and return the root of this subtree, which is the nearest ancestral subroot of the input node. We call this level-sensitive approach *local compression.* Local compression is compatible with the analysis throughout the paper, but we will not specifically mention it elsewhere for notational convenience, referring instead to just segmented compression.

We now describe our adapted form of linking by rank. Each node has an initial rank of 0. When performing a UNITE at the coarse level, the algorithm works

Algorithm 1. Segmented Compression

```
procedure SegmentCompress($x$)
    if $x.p = x$ then
        return $(x, x)$
    $(u, v) \leftarrow$ SegmentCompress $(x.p)$
    if $x.\gamma =$ true then
        $x.p \leftarrow u$
        return $(u, x)$
    else
        $x.p \leftarrow v$
        return $(u, v)$          ▷ Segmented compression returns a (root, subroot) pair
```

Algorithm 2. Find with Segmented Compression

```
procedure Find($i, x$)
    $(u, v) \leftarrow$ SegmentCompress $(x)$
    if $i = 0$ then                                   ▷ Coarse partition
        return $u$
    else                                              ▷ Fine partition
        return $v$
```

exactly like that of normal linking by rank. The root of each set is found using segmented compression. That of lesser rank is made the child of the other, making it the *loser* of the link. In case of a rank tie, the new parent is chosen arbitrarily and its rank is incremented. When performing a Unite at the fine level, the process is slightly different. The two subroots are found using compression. If one is already the parent of the other, all that needs to be done is to mark the child as a non-subroot. Otherwise, the one of lesser rank is made the child of the other. In case of a rank tie, the new parent is chosen arbitrarily and its rank incremented. If the rank of the winner is now equal to that of its parent, we need to make an adjustment. If the parent is the root, then the rank of the root is increased. Otherwise we set the parent pointer to the grandparent. This preserves strictly increasing ranks along a Find path. In any such case, the loser of the link ceases to be a subroot. See Algorithm 3 for an implementation and

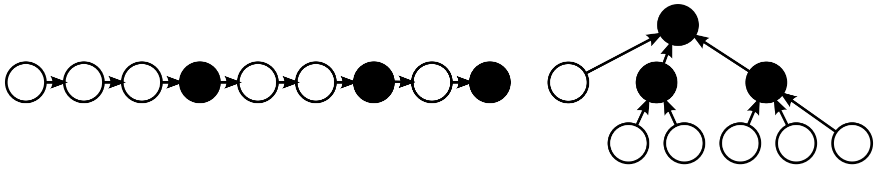


Fig. 1. The Find path on the left is segment compressed, resulting in the tree on the right. Subroots are shown in black, with non-subroots in white.

Figure 2 for demonstration of different cases of subroot linking. It is also possible to allow limited rank ties. Instead of always ensuring that the rank of the root is *greater* than those of its children, one can instead simply ensure that it is *no less* than those of its children. This leads to a slightly better constant factor guarantee on the maximum rank, but the statement of the associated bound (Lemma 1) is a bit uglier.

Algorithm 3. Nested UNITE with Linking by Rank

```
procedure UNITE($i, x, y$)
    $u \leftarrow$ FIND $(i, x)$
    $v \leftarrow$ FIND $(i, y)$
    if $u = v$ then
        return false
    if $u.r < v.r$ then
        $u \leftrightarrow v$
    $v.p \leftarrow u$
    if $u.r = v.r$ then
        $u.r \leftarrow u.r + 1$
    if $i = 1$ then                          ▷ Check for and fix rank monotonicity
        $v.\gamma =$ false
        if $u = u.p$ then
            return true
        else if $u.p = u.p.p$ and $u.p.r = u.r$ then
            $u.p.r \leftarrow u.r + 1$
        else if $u.p.r = u.r$ then
            $u.p \leftarrow u.p.p$
    return true
```

3 Properties of Node Ranks

We now establish a few key properties of node ranks with our modified algorithms. This lemma and the associated corollaries will allow us to adapt simple analyses of standard compression to work with segmented compression.

While it is almost trivial to prove that the sum of ranks is linear (each coarse UNITE can only contribute 1 to the sum, while a fine UNITE can contribute 2), we will need a somewhat stronger property. Let $R(k)$, $S(k)$, and $N(k)$ denote the maximum number of roots, subroots, and non-roots of rank k respectively (for the purpose of this analysis, "subroot" means a subroot which is not also a root).

Lemma 1. *Using linking by rank,* $R(k) \leq n \cdot (3/4)^k$ *and* $S(k) \leq 3/2 \cdot n \cdot (3/4)^k$.

Proof. We examine different types of transitions a node can undergo, and demonstrate that credits can be transferred between the nodes involved in a way which requires no new credit to be introduced to the system. The transitions and associated node count adjustments are the following:

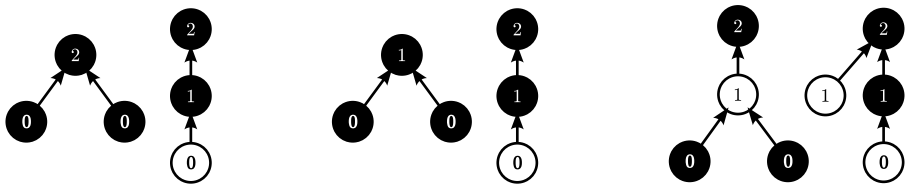


Fig. 2. Three different cases of subroot linking by rank. Subroots shown in black, with non-subroots shown in white. On the left, the two subroots are linked, and the rank of the winner increased. In the center, the rank increase causes the rank of the root to be equal to that of its child, so its rank is also increased. On the right, the rank increase causes the rank of the parent to be equal to that of its child, so the child's parent is set to the grandparent.

1. Two roots of ranks k and $j < k$ are linked. $R(j)$ decreases by 1 and $S(j)$ increases by 1.
2. Two subroots of ranks k and $j < k$ are linked. $S(j)$ decreases by 1 and $N(j)$ increases by 1.
3. Two roots of rank k are linked. $R(k)$ decreases by 2, $R(k+1)$ increases by 1, and $S(k)$ increases by 1.
4. Two subroots of rank k are linked underneath a root of rank at least $k+2$. $S(k)$ decreases by 2, $S(k+1)$ increases by 1, and $N(k)$ increases by 1.
5. Two subroots of rank k are linked underneath a root of rank $k+1$. $R(k+1)$ decreases by 1, $S(k)$ decreases by 2, $R(k+2)$ increases by 1, $S(k+1)$ increases by 1, and $N(k)$ increases by 1.

Each node must hold a minimum number of credits dependent on its type and rank. A root of rank k must hold at least $n_R(k)$ credits. Similarly a subroot and a nonroot of rank k must hold at least $n_S(k)$ and $n_N(k)$ credits respectively. We now examine the transitions and the requirements they impose on these these minimum values under the assumption that credits are preserved.

1. $n_R(k) \geq n_S(k)$
2. $n_S(k) \geq n_N(k)$
3. $2n_R(k) \geq n_R(k+1) + n_S(k)$
4. $2n_S(k) \geq n_S(k+1) + n_N(k)$
5. $n_R(k+1) + 2n_S(k) \geq n_R(k+2) + n_S(k+1) + n_N(k)$

These requirements leave some flexibility, but here is one simple and uniform way to satisfy them. We first set $n_N(k) = 0$ for all k, then we set $n_S(k+1) = {}^4\!/_3 n_S(k)$ and $n_R(k) = {}^3\!/_2 n_S(k)$. Finally we establish base values with $n_S(0) = 1$. This leads to a total credit of ${}^{3n}\!/_2$ in the system (n roots of initial rank 0).

One can easily verify that these transitions allow credits to be preserved while meeting the minimum requirements. We have that a root of rank k must have at least $n_R(0) \cdot ({}^4\!/_3)^k = {}^3\!/_2 \cdot ({}^4\!/_3)^k$ credits. Therefore with a total of ${}^{3n}\!/_2$ credit in the system, at most $n \cdot ({}^{3}\!/_{4})^{k}$ such nodes can exist. Adjusting for the value of $n_S(0)$, we also obtain a bound of at most ${}^{3n}\!/_{2} \cdot ({}^{3}\!/_{4})^{k}$ subroots of rank k.

Once credit leaves a root of rank k (either because it moves to a different node or because the node rank increases), it can never return to a root of rank k. This is also true of subroots and nonroots of a given rank. The manner in which credit is transferred imposes a partial order on node types. The lemma follows.

Corollary 1. *With linking by rank, the maximum node rank is $O(\log n)$.*

Corollary 2. *With linking by rank, the sum of ranks is $O(n)$.*

Corollary 3. *With linking by rank, the sum of maximum ranks is $O(n)$.*

4 Amortized Analysis

We will adapt the analysis of compression used in GKLT to count grandparent changes rather than parent changes. In particular we will use many of the same functional definitions and cite some building-block lemmas about them, but we will need to account for the non-fixed ranks and non-ancestral parent changes which may result from fine Unite operations. Among other implications, this means we will not be able to use the standard "union-tree" approach, which assumes that all Unite operations are performed first without compaction, and all compaction is performed on the final, full tree.

Ackermann's function [1,10] is defined recursively on two non-negative integer variables:

$$\begin{aligned} A(0,j) &= j+1 \\ A(k,0) &= A(k-1,1) \text{ if } k>0 \\ A(k,j) &= A(k-1,A(k,j-1)) \text{ if } k>0 \text{ and } j>0 \end{aligned}$$

A simple inductive argument shows that A is strictly increasing in both arguments, $A(k+1,j) \geq A(k,j+1)$, and $A(1,j) = j+2$. The function $A(k,j)$ increases very rapidly as k grows even a bit larger.

We will use the same definitions for the *inverse Ackermann function* $\alpha(r,d)$, the *index function* $b(k,r)$, and the *level function* $a(r,s)$ as found in GKLT. The variable $d = {}^{m}\!/_{n}$ is used for notational convenience. The inverse Ackermann function, defined for any non-negative integer r and non-negative real number d, is non-decreasing in r and non-increasing in d. The index function, defined for any non-negative integers k and r, is non-increasing in k and non-decreasing in r. Finally, the level function $a(r,s)$ is defined for any non-negative integers $r \leq s$.

$$\begin{aligned} \alpha(r,d) &= \min\{k>0 | A(k,\lfloor d \rfloor) > r\} \\ b(k,r) &= \min\{j \geq 0 | A(k,j) > r\} \\ a(r,s) &= \min(\{\alpha(r,d)+1\} \cup \{k \leq \alpha(r,d) \,|A(k,b(k,r)) > s\}) \end{aligned}$$

For each node x we define a *level* $x.a$ and an *index* $x.b$ follows:

$$x.a = a\,(x.r, x.p.p.r)$$
$$x.b = b\,(x.a - 1, x.p.p.r) \text{ if } x.a > 0, x.b = 0 \text{ otherwise}$$

The following three lemmas about these definitions correspond to Lemmas 3.1, 3.2, and 3.3 in GKLT respectively, and the proofs are omitted.

Lemma 2. *If $r \leq s$, $a\,(r, s) = 0$ if and only if $r = s$.*

Lemma 3. *If $x.a \leq \alpha\,(x.r, d)$, then $x.b \leq \max\{x.r, 1\} \leq x.r + 1$.*

Lemma 4. *If $x.a = \alpha\,(x.r, d) + 1 = \alpha\,(x.p.p.r, d) + 1$, then $x.b \leq d$.*

With these preliminaries, we are ready to start building a framework to count grandparent changes. We start by defining a *count* for each node x.

$$x.c = x.a \times (x.r + 2) + x.b$$

The following two lemmas are the key to counting grandparent changes. Since ranks remain fixed during a Find with the substitution of parents and grandparents the statements and proofs are equivalent to those of Lemmas 3.4 and 3.5 in GKLT (so, once more, the proofs are omitted).

Lemma 5. *During a* Find*, for every node x, $x.c$ never decreases, and $x.c$ increases whenever $x.a$ or $x.b$ changes. If $x.a$ increases by k, $x.c$ increases by at least k.*

Lemma 6. *Let x and y be nodes such that $x.p.p.r \leq y.r$ and $x.a = y.a$ just before a* Find *that sets $x.p.p$ to a node with rank at least $y.p.p.r$. Then the* Find *increases $x.c$.*

Now we are ready to count grandparent changes. Let the potential of a node x be

$$\max\{0, (\alpha\,(x.r_m, d) + 1) \times (x.r_m + 2) + d + 1 - x.c\}\,.$$

Let the potential of a collection of trees be the sum of the potentials of their nodes, and let the amortized cost of an operation be the number of grandparent changes it makes plus the change in potential it causes (all but the last three nodes on the Find path will change grandparents to the root).

Lemma 7. *The initial potential is $O\,(n\alpha\,(n, d) + m)$.*

Proof. By Corollary 3 the sum of maximum ranks is linear. By Corollary 1 the value of $\alpha\,(x.r_m, d)$ is $\mathrm{O}\,(\alpha\,(n, d))$. Thus summing over the first product term, we get at most $\mathrm{O}\,(n\alpha\,(n, d))$. An additive $\mathrm{O}\,(m + n)$ term comes from summing $d + 1$ over each node x.

Lemma 8. *The sequence of* Unite *operations increases the potential by at most $O\,(n\alpha\,(n, d) + m)$.*

Proof. A UNITE can change two parent pointers and increase up to two ranks. This means that three nodes may have their counts decreased, contributing an increase to the potential.

We consider the potential increase contributed by the parent changes first. Since the new parent of the loser need not be an ancestor, it is possible to completely reset the level and index. This can happen only once per node though, as the node ceases to be a subroot. Thus if we sum over the maximum node counts for each node, then with Corollary 3 and Lemmas 3 and 4, we get

$$\begin{aligned}\sum_x x.c &= \sum_x (x.a \times (x.r+2) + x.b)\\ &\le \sum_x ((\alpha(n,d)+1) \times (x.r_m+2) + x.b)\\ &= (\alpha(n,d)+1)\sum_x x.r_m + \sum_x x.b\\ &= \mathrm{O}(n\alpha(n,d)+m)\,.\end{aligned}$$

It is also posible that the UNITE changes the parent of the winner, though in this case the rank of the grandparent can only increase. Thus it cannot increase the potential.

Now we consider the effect of rank increases on node counts. Each UNITE may increase the rank of a root, but when a root changes rank, its level and index remain fixed at 0, meaning there is no change in potential. We need only examine rank changes in non-root subroots. Each fine UNITE may increment the rank of one such node x. A given node may have its rank updated many times this way over the course of multiple UNITE operations; however, we can use Lemma 1 to bound the total change. The decrease in count due to a rank increase of x is at most $x.a \times (x.r+2) + x.b$. By Lemma 1 there can be at most $3/2 \cdot n \cdot (3/4)^k$ subroots of rank k, and subsequently at most $3/2 \cdot n \cdot (3/4)^k$ UNITE operations increase a subroot's rank from $k-1$ to k. Since $x.r$ is $\mathrm{O}(\log n)$ by Corollary 1 for all nodes, we can sum the potential increased in this manner over all UNITE operations with relevant nodes indexed from 1 to $j < 3n$

$$\begin{aligned}\sum_{i=1}^{j} x_i.c &\le (\alpha(n,d)+1)\sum_{i=1}^{j}(x_i.r+2) + \sum_{i=1}^{j} x_i.b\\ &\le 3/2 \cdot n \cdot (\alpha(n,d)+1)\sum_{k=0}^{\infty}(k+2)\cdot(3/4)^k + \sum_{i=1}^{j} x_i.b\\ &\le 45n \cdot (\alpha(n,d)+1) + \sum_{i=1}^{j}(x_i.r+1) + \sum_{i=1}^{j} d\\ &\le 45n \cdot (\alpha(n,d)+1) + 3/2 \cdot n \cdot \sum_{k=0}^{\infty}(k+1)\cdot(3/4)^k + 3m\\ &= 45n \cdot (\alpha(n,d)+1) + 36n + 3m\end{aligned}$$

to get a bound of $\mathrm{O}(n\alpha(n,d)+m)$ with the help of Lemmas 3 and 4. Thus the total increase in potential due to UNITE operations is $\mathrm{O}(n\alpha(n,d)+m)$.

Lemma 9. *The amortized cost a* FIND *using segmented compression is* $O(\alpha(n,d))$.

Proof. Consider any FIND path. Segmented compression of the FIND path does not increase the potential of any node. Let v be the last node on the path, and let x be any node on the path whose grandparent is changed by the segmented compression. For all such nodes, $x.a > 0$ by Lemma 2. If $x.a > 0$, $\alpha(x.r,d) = \alpha(x.p.p.r,d)$, and there is a node y after x on the path such that $x.p.p.r \leq y.r$ and $y.a = x.a$, it must be the case that the FIND increases $x.p.p.r$ to at least $y.p.p.r$, since it sets $x.p.p = v$. Thus, segment compressing the path reduces the potential of x by at least two by Lemmas 3, 4, and 6. Thus the segmented compression decreases the potential of x unless $\alpha(x.r,d) < \alpha(x.p.r,d)$ or x is either the last or second–to–last on its level with rank at least $x.r$. Since $\alpha(x.r,d) \leq \alpha(x.p.r,d)$ for every x, at most $2\alpha(v.r,d)$ nodes x have $\alpha(x.r,d) < \alpha(x.p.p.r,d)$. Since every node on the path has level at most $\alpha(v.r,d)+1$, at most $2\alpha(v.r,d)+2$ nodes are last or second–to–last on their level. The amortized cost of the FIND is thus at most $4\alpha(v.r,d)+2$. By Corollary 1 $v.r$ is $\mathrm{O}(\log n)$. Subsequently the amortized cost of the FIND is $\mathrm{O}(\alpha(n,d))$.

Theorem 1. *The total time to execute* $m \geq n$ *operations using linking by rank and segmented compression is* $O(m\alpha(n, {}^{m}/_{n}))$.

Proof. Each UNITE takes constant real time, uses two internal FIND operations, and may increase the potential. Thus, the theorem follows from Lemmas 7, 8, and 9.

5 Remarks

We have shown that the two-level nested set union problem can be solved in optimal time to within a constant factor while only using one pointer and very little additional space per element. We have provided a relatively simple deterministic solution. We will now proceed to address some natural questions about extensions to the problem and the design space around it, some of which remain open.

Additional results in the design space. It is natural to wonder whether other algorithms for disjoint set union may be adapted to the nested case. In our full paper [9], we answer this question in the affirmative. It is possible to adapt linking by size to work instead of linking by rank, though with slightly worse constants. In addition to segmented compression, we have developed segmented versions of splitting and halving, thereby offering one-pass compaction methods. Any of the three compaction methods can be used with either deterministic linking rule, and their standard (local) versions can be used for fine FIND operations, including those within fine Unite operations. It has also been shown that randomized linking works in conjunction with segmented compression. It is unclear whether it will work with the other compaction methods, including local compression. The simple permutation argument to bound rank ties no longer holds. Currently the proof relies critically on the fact that fine Unite operations can only introduce a single new ancestor (i.e. both subroots are children of the root before linking).

Deeper Nesting. As we do not yet have a concrete application for nesting deeper than two levels, we did not study this extension in full depth. Both the algorithm itself and the presentation of the analysis are more complicated, but all the core tools for such work are present in this paper. Rather than a single subroot bit, a small integer field would be kept to maintain the finest level at which a node is still a root. The natural generalization of segmented compression would make each node point to the nearest coarser node on the Find path, and each recursive call would return a k-tuple rather than a pair. This solution increases the potential by a factor of k, adds an $\mathrm{O}(k)$ term to the amortized time for a Find operation, and requires $\mathrm{O}(\log k)$ extra bits per node. This makes it optimal up to a constant factor when k is $\mathrm{O}(\alpha(n, m/n))$ and kn is $\mathrm{O}(m)$. It remains open whether a better solution could be had for larger k.

References

1. Ackermann, W.: Zum hilbertschen aufbau der reellen zahlen. Mathematische Annalen 99(1), 118–133 (1928)
2. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall (1976)
3. Farrow, R.: Efficient on-line evaluation of functions defined on paths in trees. Technical Report 476-093-17, Rice University (1977)
4. Fraczak, L., Georgiadis, W., Miller, A., Tarjan, R.E.: Finding dominators via disjoint set union. J. Discrete Algorithms 23, 2–20 (2013)
5. Fredman, M.L., Saks, M.E.: The cell probe complexity of dynamic data structures. In: Proc. 21st Annual ACM Symposium on Theory of Computing, pp. 345–354 (1989)
6. Galler, B.A., Fisher, M.J.: An improved equivalence algorithm. Commun. ACM 7(5), 301–303 (1964)
7. Goel, A., Khanna, S., Larkin, D.H., Tarjan, R.E.: Disjoint set union with randomized linking. In: Proc. 25th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 1005–1017 (2014)
8. Knuth, D.E.: The Art of Computer Programming, 3rd edn. Fundamental Algorithms, vol. 1. Addison-Wesley (1997)
9. Larkin, D.H., Tarjan, R.E.: Nested set union. CoRR (2014)
10. Péter, R.: Rekursive funktionen. Académiai Kiadó (1951)
11. Tarjan, R.E.: Applications of path compression on balanced trees. J. ACM 26(4), 690–715 (1979)
12. Tarjan, R.E., van Leeuwen, J.: Worst-case analysis of set union algorithms. J. ACM 31(2), 245–281 (1984)
13. van der Wiede, T.P.: Datastructures: An Axiomatic Approach and the Use of Binomial Trees in Developing and Analyzing Algorithms. Mathematisch Centrum (1980)
14. van Leeuwen, J., van der Wiede, T.P.: Alternative path compression techniques. Technical Report RUU-CS-77-3, Rijksuniversiteit Utrecht (1977)

Improved Explicit Data Structures in the Bitprobe Model[⋆]

Moshe Lewenstein[1], J. Ian Munro[2], Patrick K. Nicholson[3], and Venkatesh Raman[4]

[1] Department of Computer Science, Bar Ilan University, Israel
[2] Cheriton School of Computer Science, University of Waterloo, Canada
[3] Max-Planck-Institut für Informatik, Saarbrücken, Germany
[4] Institute for Mathematical Sciences, Chennai, India
moshe@cs.biu.ac.il, imunro@uwaterloo.ca,
pnichols@mpi-inf.mpg.de, vraman@imsc.res.in

Abstract. Buhrman *et al.* [SICOMP 2002] studied the membership problem in the bitprobe model, presenting both randomized and deterministic schemes for storing a set of size n from a universe of size m such that membership queries on the set can be answered using t bit probes. Since then, there have been several papers focusing on deterministic schemes, especially for the first non-trivial case when $n = 2$. The most recent, due to Radhakrishnan, Shah, and Shannigrahi [ESA 2010], describes non-explicit schemes (existential results) for $t \geq 3$ using probabilistic arguments. We describe a fully explicit scheme for $n = 2$ that matches their space bound of $\Theta(m^{2/5})$ bits for $t = 3$ and, furthermore, improves upon it for $t > 3$, answering their open problem. Our structure (consisting of query and storage algorithms) manipulates blocks of bits of the query element in a novel way that may be of independent interest. We also describe recursive schemes for $n \geq 3$ that improve upon all previous fully explicit schemes for a wide range of parameters.

1 Introduction

This paper is about the fundamental *membership* problem, which asks us to store a (static) subset $\mathcal{E}$ of n elements from a universe $[1, m]$, such that we can efficiently answer membership queries on the set $\mathcal{E}$. The information theoretic lower bound indicates that any data structure solving this problem must occupy $\lg \binom{m}{n} = n \lg(m/n) + \Theta(n)$ bits of space.[1] Furthermore, existing word-RAM data structures—for instance those of Fredman, Komlós and Szemerédi [5], Brodnik and Munro [3], Pagh [8], and Pătraşcu [9]—can achieve this space bound to within constant factors, and can answer queries using a constant number of word operations: i.e., by reading $\Theta(\lg m)$ bits from the data structure.

⋆ This work was supported in part by NSERC, the Canada Research Chairs program, a David Cheriton Scholarship, and a Derick Wood Graduate Scholarship.

[1] We use $\lg x$ to denote $\log_2 x$. The stated bound assumes $n \leq m/2$.

Following Buhrman *et al.* [4] we address this problem in the *bitprobe model* (see [4, Section 2] for a formal definition). In the bitprobe model we are concerned with the tradeoff between the number of bits occupied by a data structure, and the exact number of bits of the data structure that must be read, or *probed*, in order to answer queries. Like Buhrman *et al.* [4], we argue that the problem of membership in the bitprobe model is both fundamental and natural, since to answer a query we need only one bit, indicating yes or no, rather than $\Theta(\lg m)$. Even though the bitprobe model does not accurately describe modern computing architectures, which can read and write many bits in parallel, there are scenarios in which it may find utility. For example, suppose we wish to communicate with a device that has limited storage, and for which the cost of sending a bit of information is high, yet can nonetheless recieve messages. Alon and Feige [1], who studied the problem and gave asymptotic results for larger values of n and small values of t, describe additional applications. Buhrman *et al.* also discuss different interpretations of this problem in terms of coding theory.

Buhrman *et al.*'s main result was that if randomization and false positives are allowed, then membership queries can be answered using one bit probe and (almost) optimal $\Theta(n \lg m)$ bits of space, with a fixed constant probability of error. For deterministic schemes, they showed a lower bound of $\Omega(tn^{1-1/t}m^{1/t})$ bits of space, for t probe schemes, where $n \ll m$ and $t \ll \lg m$.[2] For upper bounds, they showed that it is possible to generalize FKS-hashing [5] to the bitprobe model, achieving $\Theta(ntm^{1/t'})$ bits of space for t probes, where $t' = t - \Theta(\lg n + \lg\lg m)$, and $t' \geq 1$.

Our Results and High Level Description: As in previous work [10,11], we focus on the case where n and t are small relative to m. Our main result is a *fully explicit* deterministic scheme for the case where $n = 2$ and $t \geq 3$. A scheme is fully explicit if the locations of the probes to be performed by the query algorithm are computable in time polynomial in t and $\lg m$ given the query, *and* the bits stored in the data structure are computable in time polynomial of its size, given the subset $\mathcal{E}$ [4]. Radhakrishnan, Shah, and Shannigrahi [11] designed a non-explicit scheme occupying $\Theta(m^{2/5})$ bits for $t = 3$, and $\Theta(tm^{1/(t-(t-1)2^{1-t})})$ for $t > 3$, via probabilistic arguments. They left open whether a fully explicit scheme matching these bounds exists. Answering their question in the affirmative, we describe a fully explicit scheme that not only matches their bound for $t = 3$, but also improves upon it for $t > 3$. The only fully explicit bound previously known for the case where $n = 2$ and $t = 3$ occupies $\Theta(\sqrt{m})$ bits [10]. For $t > 3$, this had been generalized by Radhakrishnan, Shah, and Shannigrahi [11] to occupy $\Theta(t^2 m^{1/(t-1)})$ bits. In contrast, in Section 2.1 we describe a fully explicit scheme for $n = 2$ occupying $\Theta(2^t m^{1/(t-2^{2-t})})$ bits, for $t \geq 3$ probes.

Also as in previous work [10,11], our scheme divides the bits of the query element into *blocks*, and treats these blocks of bits as table indices, which are then probed sequentially. However, our scheme uses a novel interpretation of the bits read during the first $t-1$ probes: we describe the three-probe case

[2] It was, in fact, Alon and Feige [1] who wrote the lower bound in this form.

for clarity. In particular, the query algorithm treats the first two bits read as a one-to-many code that specifies two *subblocks* within the blocks: one subblock per block. During a successful search for a query element, at least one of the subblocks specified by the code is such that the bits of the two elements being stored differ within it. We use this information to select the hash function with which to make the final probe, returning the correct answer *without revealing in which block the bits differ.* Conceptually, the storage algorithm assigns two bit codes to each element, ensuring that the two stored elements x and y have different codes, if this is possible. There are additional constraints on how the codes are assigned, but our main technical contribution is that we explicitly describe a set of hash functions $h_i : m^{4/5} \mapsto m^{1/5}$, for $i = 1, ..., 4$ such that all elements assigned the same code as x (resp. y) do not collide with x (resp. y). We contrast our approach with the previous best fully explicit three probe scheme [10], which, with a simple modification, can be made to reveal in which block the two stored elements differ during a successful search. We believe that our approach of limiting the amount of information revealed about the set $\mathcal{E}$ during a search—by *overloading* the interpretation of the bits returned in the first two probes—leads to a more space efficient data structure.

For $n \geq 3$, the non-explicit scheme of Radhakrishnan, Shah, and Shannigrahi [11] occupies $\Theta(ntm^{1/(t-(n-1)(t-1)2^{1-t})})$ bits. They also described a fully explicit scheme that occupies $\Theta(t^n m^{1/(t-n+1)})$ bits. Both of these schemes require $t > n$. In Section 2.2 we present schemes for $n \geq 3$ that are fully explicit and, though they fall short of matching the upper bounds for the non-explicit scheme, improve all known fully explicit bounds. For any $t \geq 2\lfloor \lg n \rfloor + 1$, our scheme uses $\Theta(2^t m^{1/(t-\min\{2\lfloor \lg n \rfloor, n-3/2\})})$ bits of space. In particular, we note this scheme occupies less space than both the fully explicit FKS-based scheme of Buhrman *et al.* [4] (notice our lack of a dependence on m in the exponent), and the fully explicit scheme of Radhakrishnan, Shah, and Shannigrahi [11]. Furthermore, the subtracted term in the exponent-of-m in our scheme is not only exponentially smaller than that of Radhakrishnan, Shah, and Shannigrahi [11], but is also applicable in the range $t \in [2\lfloor \lg n \rfloor + 1, n]$.

The main technique we use for the $n \geq 3$ case is to define new kinds of (j, δ)-*decompositions* of the universe $[1, m]$. Simply put, a (j, δ)-decomposition divides the universe into buckets of size $m^{1-\delta}$, and assigns j bits to each bucket. Thus, a (j, δ)-decomposition occupies jm^{δ} bits of space. Using this terminology, the previous fully explicit result of Radhakrishnan, Shannigrahi, and Shah [11] uses a $(n, 1/(t - n + 1))$-decomposition, based on a unary encoding. Our approach is to use what we call a $(2, 1/(t - \Theta(\lg n))$-*balanced decomposition.* While the previous unary encoding approach is analogous to a linked list, our approach is analogous to a balanced tree. Furthermore, our decomposition strategy may also be of independent interest, as it can be used to solve a number of problems that generalize membership, such as rank [9], one-dimensional range counting (which can be solved using two rank queries) and emptiness [2], with similar space tradeoffs; see Chapter 4 of the third author's thesis for details [6].

Finally, in Section 3 we close by discussing the difficulties of matching the known non-explicit bounds [11] for the $n \geq 3$ case in a fully explicit scheme.

Notation and Definitions: As in previous work [4,10,11] we use the notation (n, m, s, t)-scheme to refer to a scheme structure that uses s bits of memory to store any n element subset of a universe of size m, such that queries can always be answered using t probes. For example, the trivial bit vector data structure with direct access is an $(n, m, m, 1)$-scheme for the membership problem. A scheme is *adaptive* if any probe after the first uses the results of prior probes in order to determine the location of the next probe; otherwise the scheme is *non-adaptive.* All the results we have discussed are for adaptive schemes, though we note that many interesting results have been proved for non-adaptive schemes as well [4,10,1,11]: for example, Buhrman et al. [4] showed that there is a non-explicit non-adaptive $(n, m, O(ntm^{4/t+1}), t)$-scheme.

Other Related Work: For a comprehensive discussion of related work and definitions we refer the reader to a recent survey on the topic [7]; we only mention the most closely related results here. For $n = 1$, there is a folklore explicit scheme that achieves $\Theta(tm^{1/t})$ bits of space. The scheme divides the bits of the element we wish to represent into t blocks, and stores the characteristic bit vector for each block. Given a query element, we can answer a membership query by probing each characteristic bit vector. For $n = t = 2$, Radhakrishnan, Raman, and Rao [10] designed a subtle fully explicit scheme that uses $\Theta(m^{2/3})$ bits of space. They also showed that this bound is tight for a restricted set of schemes. Radhakrishnan, Shah, and Shannigrahi [11] proved a general (unrestricted) lower bound of $\Omega(m^{4/7})$, which improves the lower bound of Buhrman *et al.* for the $n = 2$ case. Alon and Feige [1] presented several schemes for the case where $t \in \{2, 3, 4\}$. While they have strong asymptotic bounds for larger n, in the case when $n = 2$, these schemes yield weaker upper bounds: $\Theta(m^{2/3})$ for $t = 3$ probes. Finally, we mention that Viola [12] has proved a lower bound for the case when $n = \Theta(m)$, which is not covered by the lower bound of Buhrman *et al.*

2 Technical Discussion

2.1 Explicit Adaptive Schemes for $n = 2$

In this section we begin by proving a special case of our main theorem, where $t = 3$. For brevity, we omit floor and ceiling operators in cases where they do not affect asymptotic complexity.

Theorem 1. *There is a fully explicit adaptive* $(2, m, \Theta(m^{2/5}), 3)$*-scheme. In other words, there is a fully explicit adaptive scheme that occupies* $\Theta(m^{2/5})$ *bits for storing two elements from the universe* $[1, m]$*, such that membership queries can always be answered in* 3 *probes.*

Proof. We define a *subblock* to be $\lg m/5$ consecutive bits. A *block* is two consecutive subblocks. Let $z \in [0, m-1]$ be an integer[3], and z_i be the i-th bit in

[3] For notational convenience, we remap the universe $[1, m]$ to $[0, m-1]$ in our proofs.

the binary representation of z, for $1 \le i \le \lg m$, where z_1 is the most significant bit. We divide the binary representation of z into blocks, $B_1(z), B_2(z), B_3(z)$, where $B_j(z)$ are bits $z_{(j-1)\chi+1}, z_{(j-1)\chi+2}, \ldots, z_{\min(j\chi, \lg m)}$, where $j \in \{1,2,3\}$ and $\chi = 2\lg m/5$. The block $B_3(z)$ is special in that it is not a complete block: i.e., it will only consist of one subblock, rather than two. Finally, we use $B_{j,k}(z)$ to denote the k-th subblock of the j-th block of z, for $1 \le k \le 2$. An illustration of these definitions can be found in Figure 1. Note that in the following description when we refer to a particular block or subblock, we are referring to the binary number represented by the bits contained in the block, following the convention that the leftmost bit is the most significant bit.

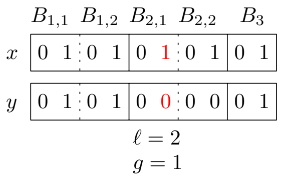


Fig. 1. Given integers $x = 341$, and $y = 321$, and a universe $[0, 1023]$, we divide the bits of x and y into blocks

Our scheme stores 7 tables. Each table is denoted $\mathcal{T}_{\mathcal{S}}$, where $\mathcal{S}$ is a bit string in $\{\epsilon, 0, 1, 00, 01, 10, 11\}$, and ϵ represents the empty string. Each table occupies $m^{2/5}$ bits. Thus, the total space is $7m^{2/5}$.

We begin by describing the algorithm for searching the data structure. Let q be the element we are searching for:

1. We probe table $\mathcal{T}_\epsilon$ at location $B_1(q)$, and are given bit r_1.
2. We probe table $\mathcal{T}_{r_1}$ at location $B_2(q)$, and are given bit r_2.
3. We read the bit r_3 by probing table $\mathcal{T}_{r_1 r_2}$ at location:

$$\left((B_{1,r_2+1}(q) + B_{2,r_1+1}(q)) \mod m^{1/5}\right) m^{1/5} + B_3(q)\ . \tag{1}$$

4. If $r_3 = 1$ then we return `YES`, otherwise we return `NO`.

Next we describe how to construct the data structure, i.e., set the bits. Consider the two elements $x, y \in [0, m-1]$ that we wish to store, and assume without loss of generality that $x < y$. Let ℓ be the smallest integer such that $B_\ell(x)$ differs from $B_\ell(y)$, and g be the smallest integer such that $B_{\ell,g}(x)$ differs from $B_{\ell,g}(y)$. See Figure 1 for an example. Let $g' = g - 1$: it is a bit representing the subblock in which x and y differ. We next argue that we can assume $\ell < 3$, since a trivial assignment exists in the alternate case.

If $\ell = 3$, then we are free to select any of the 4 tables $\{\mathcal{T}_{\mathcal{S}}\}$, where $|\mathcal{S}| = 2$, as a *destination* for the elements x and y. Without loss of generality, assume we choose $\mathcal{T}_{11}$. Thus, we assign the characteristic bit vector of $B_1(x)$ to table $\mathcal{T}_\epsilon$, and the characteristic bit vector of $B_2(x)$ to table $\mathcal{T}_1$. In the final table, $\mathcal{T}_{11}$, we store (at most) two ones in the locations that can be computed by plugging in x and y into Equation 1. All other entries are set to zero. It is not difficult to see that the search for q will function correctly for this assignment, because of the fact that x and y are identical in all blocks except possibly B_3.

If $\ell = 1$, then let $\mathcal{S}_x = 0g'$, and $\mathcal{S}_y = 1g'$. Otherwise, if $\ell = 2$, then let $\mathcal{S}_x = g'0$, and $\mathcal{S}_y = g'1$. Our aim is use $\mathcal{T}_{\mathcal{S}_x}$ as the destination for x and $\mathcal{T}_{\mathcal{S}_y}$ as the destination for y, while ensuring that the search algorithm always returns the correct result. We use the notation $\mathcal{S}_{z,u}$ to denote the u-bit prefix of $\mathcal{S}_z$. For example, if $\mathcal{S}_z = 01$, then $\mathcal{S}_{z,0} = \epsilon$, $\mathcal{S}_{z,1} = 0$, $\mathcal{S}_{z,2} = 01$.

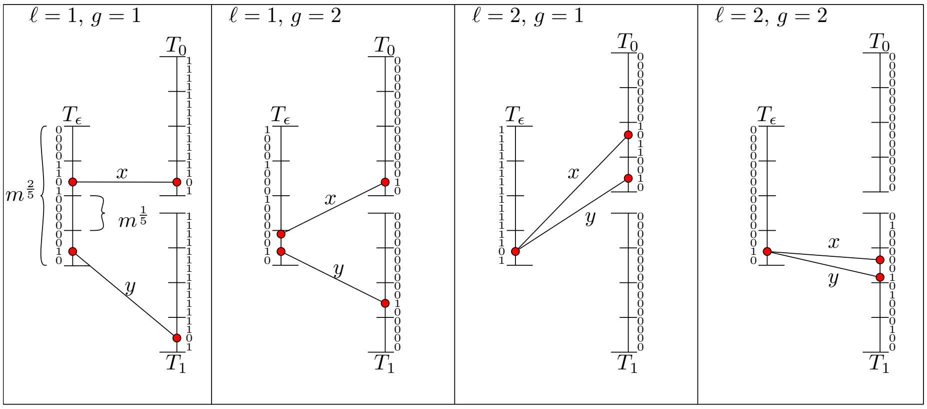


Fig. 2. Four cases illustrating how to set the bits in tables $\mathcal{T}_\epsilon$, $\mathcal{T}_0$, and $\mathcal{T}_1$. In this example the universe is the range $[0, 1023]$.

We now describe the assignment of bits to the tables:

1. For the value $u \in [0, 1]$, where $u \neq \ell - 1$, we describe how to set the values in $\mathcal{T}_{\mathcal{S}_{z,u}}$, for $z \in \{x, y\}$. Let v be the $(u+1)$-th bit of $\mathcal{S}_z$. If v is a 0, then we set the bit in location $B_{u+1}(z)$ to 0, and all other bits to 1 in $\mathcal{T}_{\mathcal{S}_{z,u}}$. Otherwise, if v is 1, then we set the bit in location $B_{u+1}(z)$ to 1, and all other bits to 0.
2. Next, we explain how to set the bits in $\hat{\mathcal{T}} = \mathcal{T}_{\mathcal{S}_{x,\ell-1}} = \mathcal{T}_{\mathcal{S}_{y,\ell-1}}$. We store a 0 in location $B_\ell(x)$, and a 1 in location $B_\ell(y)$ in $\hat{\mathcal{T}}$. All locations $\gamma_x \neq B_\ell(x)$, such that $\lfloor \gamma_x / m^{(1-g')/5} \rfloor \equiv B_{\ell,g}(x) \mod m^{1/5}$ are assigned a 1. All locations $\gamma_y \neq B_\ell(y)$, such that $\lfloor \gamma_y / m^{(1-g')/5} \rfloor \equiv B_{\ell,g}(y) \mod m^{1/5}$ are assigned a 0. After setting the tables in the way described above, we perform a search for x and y using the search algorithm, and set the two bits corresponding to x and y to 1 in tables $\mathcal{T}_{\mathcal{S}_x}$ and $\mathcal{T}_{\mathcal{S}_y}$, respectively. Finally, all table locations that remain unspecified are set to 0. We give an example of how to set tables $\mathcal{T}_\epsilon$, $\mathcal{T}_0$, and $\mathcal{T}_1$ in each of the four cases in Figure 2.

All that remains is to prove that the search algorithm will always return the correct result, provided we set the bits as described above. In the first case, if the query element q is equal to either x or y, then we will return YES, which is correct. In the second case, suppose $q \neq x$ *and* $q \neq y$, and $\mathcal{S}_q$ is the sequence of first two bits returned by the search algorithm for q. If $\mathcal{S}_q \neq \mathcal{S}_x$ *and* $\mathcal{S}_q \neq \mathcal{S}_y$, then we will return NO—since all tables are zeroed out except those containing x and y—which is correct. Otherwise, suppose $\mathcal{S}_q = \mathcal{S}_z$ where $z \in \{x, y\}$; note that we can assume $\mathcal{S}_x \neq \mathcal{S}_y$ by the assumption that $\ell < 3$. If q differs from z in block B_3, then the search algorithm will return NO, which is correct. Thus, we can assume q differs from z somewhere other than block B_3. Consider the table $\mathcal{T}_{\mathcal{S}_{q,u}} = \mathcal{T}_{\mathcal{S}_{z,u}}$, where $u \in [0, 1]$ *and* $u \neq \ell - 1$, and let v denote the value of the $(u+1)$-th bit in $\mathcal{S}_q$. Since the table $\mathcal{T}_{\mathcal{S}_{z,u}}$ contains only one location that stores bit v, we can infer that q and z can differ *only* in block ℓ. However, $B_{\ell,g}(q) \neq B_{\ell,g}(z)$, according to the way we set the bits in table $\hat{\mathcal{T}}$, since $q \neq z$. Based on the discussion above, we have that either: (1) $B_{1,r_2+1}(q) \neq B_{1,r_2+1}(z)$ *and* $B_{2,r_1+1}(q) = B_{2,r_1+1}(z)$; or (2) $B_{1,r_2+1}(q) = B_{1,r_2+1}(z)$ *and* $B_{2,r_1+1}(q) \neq B_{2,r_1+1}(z)$. This implies that: $(B_{1,r_2+1}(q) + B_{2,r_1+1}(q) \neq B_{1,r_2+1}(z) + B_{2,r_1+1}(z)) \mod m^{1/5}$. Thus, we will return the correct answer of NO. This completes the proof of correctness.

Next, we state our main theorem, and provide most of the details of the proof. The remaining details for this proof can be found in Chapter 4 of the third author's thesis [6]. Note that this result improves the non-explicit bound of Radhakrishnan, Shah, and Shannigrahi [11], provided $2^t = o(m^{\varepsilon})$ for any constant $\varepsilon > 0$.

Theorem 2. *There is a fully explicit adaptive* $(2, m, \Theta((2^t - 1)m^{1/(t-2^{2-t})}), t)$-*scheme for the membership problem, for* $t \geq 3$.

Proof. We define a *subblock* to be $(\lg m/(t2^{t-2} - 1))$ consecutive bits. A *block* is 2^{t-2} consecutive subblocks. Let $z \in [0, m-1]$ be an integer, and z_i be the i-th bit in the binary representation of z, for $1 \leq i \leq \lg m$, where z_1 is the most significant bit. We divide the binary representation of z into blocks, $B_1(z), ..., B_t(z)$, where $B_j(z)$ are bits $z_{(j-1)\chi+1}, z_{(j-1)\chi+2}, ..., z_{\min(j\chi, \lg m)}$, where $1 \leq j \leq t$ and $\chi = ((2^{t-2})\lg m)/(t2^{t-2} - 1)$. The block $B_t(z)$ is special in that it is not a complete block: i.e., it will only consist of $2^{t-2} - 1$ consecutive subblocks, rather than 2^{t-2}. Finally, we use $B_{j,k}(z)$ to denote the k-th subblock of the j-th block of z, for $1 \leq k \leq 2^{t-2}$. Note that in the following description when we refer to a particular block or subblock, we are referring to the binary number represented by the bits contained in the block, following the convention that the leftmost bit is the most significant bit.

Our scheme stores $2^t - 1$ tables. Each table is denoted $\mathcal{T}_{\mathcal{S}}$, where $\mathcal{S}$ is a binary string of length between 0 (an empty string), and $t-1$ bits. Each table $\mathcal{T}_{\mathcal{S}}$, where $|\mathcal{S}| \leq t$ will store $m^{(2^{t-2})/(t2^{t-2}-1)}$ bits. The sum of the sizes of these tables is no more than the space bound claimed in the statement of the theorem.

We begin by describing the algorithm for searching the data structure:

1. Let q be the query element, $\mathcal{S}$ be an empty binary string, and $i = 1$.
2. We probe table $\mathcal{T}_{\mathcal{S}}$ at location $B_i(q)$, and are given bit r_i. We append r_i to $\mathcal{S}$ (i.e., add r_i to the end of $\mathcal{S}$) and increment i. If $i \leq t - 1$, then we repeat this step.

3. At this point $\mathcal{S}$ consists of $t-1$ bits. Let $\mathcal{S}[j]$ be the binary number that results from deleting the j-th digit (counting left to right) from $\mathcal{S}$. We read the bit r_t by probing table $\mathcal{T}_{\mathcal{S}}$ at location:

$$\left(\left(\sum_{j=1}^{t-1} B_{j,\mathcal{S}[j]+1}(q)\right) \mod m^{1/(t2^{t-2}-1)}\right) m^{(2^{t-2}-1)/(t2^{t-2}-1)} + B_t(q) \ . \tag{2}$$

4. If $r_t = 1$ then we return `YES`, otherwise we return `NO`.

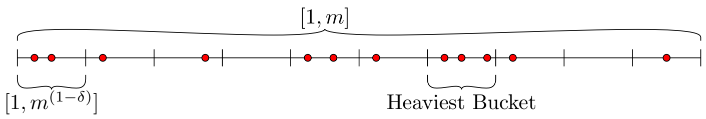
Next we describe how to construct the data structure, i.e., set the bits. Consider the two elements $x, y \in [0, m-1]$ that we wish to store, and assume without loss of generality that $x < y$. Let ℓ be the smallest integer such that $B_\ell(x)$ differs from $B_\ell(y)$, and g be the smallest integer such that $B_{\ell,g}(x)$ differs from $B_{\ell,g}(y)$. As in the $t = 3$ case, we can assume $\ell < t$, since a trivial assignment exists in the alternate case.

Let $g_1, ..., g_{t-2}$ be the digits of the binary representation of $g-1$. Let $\mathcal{S}_x = g_1, g_2, ..., g_{\ell-1}, 0, g_\ell, ..., g_{t-2}$, and $\mathcal{S}_y = g_1, g_2, ..., g_{\ell-1}, 1, g_\ell, ..., g_{t-2}$. We use the notation $\mathcal{S}_{z,u}$ to denote the u-bit prefix of $\mathcal{S}_z$. For each $u \in [0, t-2]$, where $u \neq \ell - 1$, we describe how to set the values in $\mathcal{T}_{\mathcal{S}_{z,u}}$, for $z \in \{x, y\}$. Let v be the $(u+1)$-th bit of $\mathcal{S}_z$. If v is a 0, then we set the bit in location $B_{u+1}(z)$ to 0, and all other bits to 1 in $\mathcal{T}_{\mathcal{S}_{z,u}}$. Otherwise, if v is 1, then we set the bit in location $B_{u+1}(z)$ to 1, and all other bits to 0. We now explain how to set the bits in $\hat{\mathcal{T}} = \mathcal{T}_{\mathcal{S}_{x,\ell-1}} = \mathcal{T}_{\mathcal{S}_{y,\ell-1}}$. We store a 0 in location $B_\ell(x)$, and a 1 in location $B_\ell(y)$ in $\hat{\mathcal{T}}$. All locations $\gamma_x \neq B_\ell(x)$, such that $\lfloor \gamma_x / m^{(2^{t-2}-1-g)/(t2^{t-2}-1)} \rfloor \equiv B_{\ell,g}(x) \mod m^{1/(t2^{t-2}-1)}$ are assigned a 1. All locations $\gamma_y \neq B_\ell(y)$, such that $\lfloor \gamma_y / m^{(2^{t-2}-1-g)/(t2^{t-2}-1)} \rfloor \equiv B_{\ell,g}(y) \mod m^{1/(t2^{t-2}-1)}$ are assigned a 0. After setting the tables in the way described above, we perform a search for x and y using the search algorithm, and set the two bits corresponding to x and y to 1 in tables $\mathcal{T}_{\mathcal{S}_x}$ and $\mathcal{T}_{\mathcal{S}_y}$, respectively. Finally, all table locations that remain unspecified are set to 0. We omit the discussion of correctness, which follows from arguments similar to the $t = 3$ case.

2.2 Explicit Adaptive Schemes for $n \geq 3$

In this section we describe our general scheme for the case when $n \geq 3$. We begin by providing some definitions for decomposition techniques that are used in the recursive schemes defined in the remainder of the paper. We then devise a scheme for small values of n that uses Theorem 2 recursively to outperform the previous fully explicit scheme of Radhakrishnan *et al.* Finally, using a slightly more sophisticated decomposition technique we improve all previous fully explicit results for the general case of $n \geq 3$. All proofs omitted due to space constraints can be found in Chapter 4 of the third author's thesis [6].

We begin by defining the *heaviest bucket* in a (j, δ)-decomposition as the bucket which contains the most elements of the set $\mathcal{E}$, breaking ties arbitrarily. Next, we discuss some special kinds of (j, δ)-decompositions. A $(1, \delta)$-*balanced*



a	0	0	0	0	0	0	1	0	0	0
b	0	0	0	0	0	0	1	0	0	0
	0	0	0	0	1	1	X	1	1	1

Fig. 3. Illustration of the various decomposition strategies on the universe represented by the horizontal line, containing elements marked with red dots. For each decomposition, each of the bits are drawn below the bucket with which it is associated. Bits labelled 'X' can either be set to 0 or 1. (a) is a $(1, \delta)$-balanced decomposition, and (b) is a $(2, \delta)$-balanced decomposition.

decomposition is a $(1, \delta)$-decomposition in which the heaviest bucket is assigned a 1, and all remaining buckets are assigned a 0 (see Figure 3 (a)). Similarly, a $(2, \delta)$-*balanced decomposition* is a $(2, \delta)$-decomposition in which the heaviest bucket is assigned a 10 or 11 (the second bit is irrelevant). We have the following:

Lemma 1. *Suppose there is a set of buckets, where each bucket contains between* 0 *and n elements, and the total number of elements in all the buckets is n. Suppose the heaviest bucket is removed. It is possible to partition the remaining buckets into two groups $\mathcal{G}_0$ and $\mathcal{G}_1$, such that there are no more than $\lfloor n/2 \rfloor$ elements in total contained in the buckets of either group.*

To the buckets in groups $\mathcal{G}_0$ and $\mathcal{G}_1$ in our $(2, \delta)$-balanced decomposition we assign the bits 00 and 01, respectively (see Figure 3 (b)).

Basic Scheme for $n \geq 3$. We describe a recursive scheme for the case when $n \geq 3$. We improve the bound for $n > 5$ in the next section. We begin by stating a useful preliminary result of Radhakrishnan, Raman, and Rao:

Lemma 2 (Theorem 1 from [10]). *For $n \geq 2$, there is a fully explicit adaptive $(n, m, 2n\sqrt{m}, \lceil \lg(n+1) \rceil + 1)$-scheme for the membership problem.*

We next describe a scheme that extends the result of Theorem 2 to obtain better space bounds for the case when $n \geq 3$ and the $t = n + 1$.

Theorem 3. *Suppose that for $n_0 \geq 2$, $f > 0$, and $c \in [1/3, 1]$, there is a fully explicit adaptive $(n_0, m, fm^c, n_0 + 1)$-scheme for the membership problem. Then, for any $n > n_0 \geq 2$, there is a fully explicit adaptive $(n, m, (f + \sum_{j=n_0+1}^{n}(2j + 1))m^c, n + 1)$-scheme for the membership problem.*

Proof. Proof by induction on n. The supposition in the statement of the theorem serves as the base case when $n = n_0$. For the induction hypothesis, we assume there is a $(n - 1, m, (f + \sum_{j=n_0+1}^{n-1}(2j + 1))m^c, n)$-scheme. We compute and store a $(1,c)$-balanced decomposition. The query proceeds by probing the bucket associated with the first c-th of the bits of the query element q. Since at least one element will be in the heaviest bucket, if the probe returns 0 it is sufficient to defer to an $(n-1, m, (f+\sum_{j=n_0+1}^{n-1}(2j+1))m^c, n)$-scheme, which exists by the induction hypothesis. In this case, we have reduced the number of elements by at least one, but have not reduced the size of the universe. In the alternative case, if the probe returns 1, then we know that at most n elements are stored in a smaller universe, of size m^{1-c}. Thus, we can defer this case to an $(n, m^{1-c}, 2nm^{(1-c)/2}, n)$-scheme, which exists by Lemma 2 (since $n \geq 3$ and $\lceil \lg(n+1) \rceil + 1$ probes are sufficient for this scheme). Overall the total number of bits used by the scheme is: $(f+1+\sum_{j=n_0+1}^{n-1}(2j+1))m^c + 2nm^{(1-c)/2}$, which is no more than $(f+1+\sum_{j=n_0+1}^{n}(2j+1))m^c$, since $c \geq 1/3$.

Combining Theorems 2 and 3 we get the following corollary, setting $n_0 = 2$ in Theorem 3, and by applying the scheme of Theorem 2 with $t = 3$.

Corollary 1. *For $n \geq 3$, there is a fully explicit adaptive $(n, m, (n^2 + 2n + 6)m^{2/5}, n+1)$-scheme for the membership problem.*

Next, using roughly the same decomposition technique as Theorem 3, we extend the result to the more general case when $t > n \geq 2$.

Theorem 4. *Suppose for the membership problem there is a fully explicit adaptive $(n_0, m, (2^{n_0+1} - 1)m^c, n_0 + 1)$-scheme for some $c \in (0,1]$, and a fully explicit adaptive $(n_0 - 1, m, (2^{t_1} - 1)m^{1/(t_1 - n_0 + 1/c - 1)}, t_1)$-scheme for $n_0 \geq 3$, and $t_1 > n_0$. For any $n \geq n_0$, $t > n$, there is a fully explicit adaptive $(n, m, (2^t - 1)m^{1/(t-n+1/c-1)}, t)$-scheme for the membership problem.*

We can combine Theorem 4 (with $c = 2/5$) with Corollary 1, and the observation that $n^2 + 2n + 6 < 2^t - 1$, for $n \geq 3$ and $t > n$. This improves the exponent-of-m in the result of Radhakrishnan, Shah, Shannigrahi [11].

Corollary 2. *For $n \geq 3$ and $t > n$, there is an fully explicit adaptive $(n, m, (2^t - 1)m^{1/(t-n+3/2)}, t)$-scheme for the membership problem.*

Improved Scheme for $n \geq 3$. Next, we present a fully explicit adaptive scheme that achieves significantly better bounds than the one from the previous section. We start by stating the bound of the trivial scheme for one element:

Lemma 3. *[4] There is a fully explicit non-adaptive $(1, m, tm^{1/t}, t)$-scheme for the membership problem.*

Using Lemma 3 we prove the following:

Theorem 5. *Let $\mathfrak{R}(n)$ be the recurrence defined by $\mathfrak{R}(0) = \mathfrak{R}(1) = 0$ and $\mathfrak{R}(n) = \mathfrak{R}(\lfloor n/2 \rfloor) + 1$. For $n \geq 2$ and $t \geq 2\mathfrak{R}(n) + 1$, there is a fully explicit adaptive $(n, m, (2^t - 1)m^{1/(t-2\mathfrak{R}(n))}, t)$-scheme for the membership problem.*

Proof. The proof is by strong induction on both t and n. In the base case we have $t = 2\mathfrak{R}(n)+1$, and we store the trivial $(n, m, m, 1)$-scheme. In the inductive case, we assume $t > 2\mathfrak{R}(n) + 1$. We compute and store a $(2, 1/(t - 2\mathfrak{R}(n)))$-balanced decomposition. The search algorithm proceeds as follows: if we probe the bucket associated with the query element and read a 1 bit, we immediately recurse to the $(n, m', (2^{t-1} - 1)m'^{1/(t-2\mathfrak{R}(n)-1)}, t-1)$-scheme that is guaranteed to exist by the induction hypothesis. In this case we have reduced the size of the universe to $m' = m^{(t-2\mathfrak{R}(n)-1)/(t-2\mathfrak{R}(n))}$, but not changed the number of elements. Otherwise, we read both bits associated with the bucket. After reading either 00 or 01, it is sufficient to recurse to a scheme that represents a set of $\lfloor n/2 \rfloor$ elements. Thus, in this case we have reduced the number of elements, but not the size of the universe.

If $\lfloor n/2 \rfloor = 1$, then we can recurse to two copies of the trivial $(1, m, (t - 2)m^{1/(t-2)}, t-2)$-scheme of Lemma 3 (one for 00 and one for 01). Otherwise, we assume the existence of a $(\lfloor n/2 \rfloor, m, (2^{t-2} - 1)m^{1/(t-2-2(\mathfrak{R}(n)-1))}, t-2)$-scheme, and recurse to two separate copies of this scheme.

We now analyze the space bound. If $n \in \{2, 3\}$ then $\lfloor n/2 \rfloor = 1$ and $\mathfrak{R}(n) = 1$. In this case, we have stored no more than: $(2 + 2^{t-1} - 1 + 2(t-2))m^{1/(t-2)}$ bits, which is no more than the claimed space bound since $t > 3$. If $n > 3$ then the overall space is no more than $(2 + 2^{t-1} - 1 + 2(2^{t-2} - 1))m^{1/(t-2\mathfrak{R}(n))}$ bits, which is exactly $(2^t - 1)m^{1/(t-2\mathfrak{R}(n))}$ bits, completing the proof.

Since $\mathfrak{R}(n) = \lfloor \lg n \rfloor$, we get the following corollary by combining Corollary 2 and Theorem 5. This result improves the both previously known fully explicit bounds [4,11], provided $2^t = o(m^\varepsilon)$ for any constant $\varepsilon > 0$.

Corollary 3. *For $n \geq 2$ and $t \geq 2\lfloor \lg n \rfloor + 1$, there is a fully explicit adaptive $(n, m, (2^t - 1)m^{1/(t-\min\{2\lfloor \lg n \rfloor, n-3/2\})}, t)$-scheme for the membership problem.*

3 Further Discussion

We have presented new fully explicit schemes for the membership problem in the bitprobe model. These schemes significantly outperform the previous fully explicit schemes of Radhakrishnan, Shah, and Shannigrahi [11], and Buhrman *et al.* [4], for a wide range of input parameters, answering an open problem [11].

Our main result for the $n = 2$ case is both fully explicit, and even outperforms the best previous non-explicit schemes. We conclude with a discussion regarding the difficulty of matching the non-explicit bounds for $n \geq 3$. First we introduce some further terminology. We observe that the functions used to determine the locations to probe on the first $t - 1$ probes for both the non-explicit scheme of Radhakrishnan *et al.* [11], and Theorem 2 have a particular format. In particular, if the desired space bound is $\Theta(m^c)$ for some constant $c > 0$—let us treat t as a constant to simplify the discussion—then the schemes divide the bits of the query element q into blocks $B_1(q), ..., B_t(q)$, where the first $t-1$ blocks consist of $c\lceil \lg m \rceil$ bits, and the t-th block is potentially smaller. We refer to such a scheme that divides the bits of the query element into blocks, such that the location of probe i, $1 \leq i \leq t-1$ is specified by (the number represented by) $B_i(q)$, as *blocking*. Note that this definition makes no claims about the function used to determine the final probe. Thus, the fully explicit scheme of Theorem 2 is blocking, whereas the fully explicit scheme of Corollary 3 is not. We also refine our notion of adaptivity, and define a ρ-*adaptive* scheme to be one in which only the final ρ probes are adaptive. We have the following conjecture:

Conjecture 1. There is no fully explicit $(2, m, \Theta(m^{1/3-\varepsilon}), 4)$-scheme that is both 2-adaptive and blocking for any constant $\varepsilon > 0$.

Assuming this conjecture is true, we have strong evidence that matching the non-explicit bound of Radhakrishnan *et al.* [11] using an approach similar to that of Theorem 2 would be difficult for $n \geq 3$. In particular, we show that a wide class of blocking schemes with rather natural seeming properties violate Conjecture 1. See Chapter 4 of the third author's thesis for details [6].

References

1. Alon, N., Feige, U.: On the power of two, three and four probes. In: Proc. of the 20th Annual Symposium on Discrete Algorithms (SODA), pp. 346–354. SIAM (2009)
2. Alstrup, S., Brodal, G., Rauhe, T.: Optimal static range reporting in one dimension. In: Proc. of the 33rd Annual Symposium on Theory of Computing (STOC), pp. 476–482. ACM (2001)
3. Brodnik, A., Munro, J.I.: Membership in constant time and almost-minimum space. SIAM Journal on Computing 28(5), 1627–1640 (1999)
4. Buhrman, H., Miltersen, P., Radhakrishnan, J., Venkatesh, S.: Are bitvectors optimal? SIAM Journal on Computing 31(6), 1723–1744 (2002)
5. Fredman, M.L., Komlós, J., Szemerédi, E.: Storing a sparse table with 0(1) worst case access time. Journal of the ACM (JACM) 31(3), 538–544 (1984)
6. Nicholson, P.K.: Space Efficient Data Structures in the Word-RAM and Bitprobe Models. Ph.D. thesis, University of Waterloo (2013)
7. Nicholson, P.K., Raman, V., Rao, S.: Data structures in the bitprobe model. In: Brodnik, A., López-Ortiz, A., Raman, V., Viola, A. (eds.) Ianfest-66. LNCS, vol. 8066, pp. 303–318. Springer, Heidelberg (2013)
8. Pagh, R.: Low redundancy in static dictionaries with constant query time. SIAM Journal on Computing 31(2), 353–363 (2001)
9. Pătraşcu, M.: Succincter. In: Proc. of the 49th Annual Symposium on Foundations of Computer Science, pp. 305–313. IEEE (2008)
10. Radhakrishnan, J., Raman, V., Rao, S.S.: Explicit deterministic constructions for membership in the bitprobe model. In: Meyer auf der Heide, F. (ed.) ESA 2001. LNCS, vol. 2161, pp. 290–299. Springer, Heidelberg (2001)
11. Radhakrishnan, J., Shah, S., Shannigrahi, S.: Data structures for storing small sets in the bitprobe model. In: de Berg, M., Meyer, U. (eds.) ESA 2010, Part II. LNCS, vol. 6347, pp. 159–170. Springer, Heidelberg (2010)
12. Viola, E.: Bit-probe lower bounds for succinct data structures. SIAM Journal on Computing 41(6), 1593–1604 (2012)

Deeper Local Search for Better Approximation on Maximum Internal Spanning Trees*

Wenjun Li[1], Jianer Chen[1,2], and Jianxin Wang[1]

[1] School of Information Science and Engineering
Central South University
Changsha, Hunan 410083, P.R. China
[2] Department of Computer Science and Engineering
Texas A&M University
College Station, Texas 77843-3112, USA

Abstract. Spanning tree has been fundamental in the research of graph algorithms. In this paper, we study the optimization problem MaxIST, which maximizes the number of internal nodes in a spanning tree of a given graph, and is a generalization of the famous Hamiltonian-Path problem. We present a polynomial-time approximation algorithm based on a deep local search strategy, identify combinatorial structures that support thorough analysis on the spanning trees resulted from such deep local search strategies, and prove that our algorithm has an approximation ratio 1.5 for the MaxIST problem, improving the previous best approximation algorithm of ratio 5/3 for the problem.

1 Introduction

Spanning trees have been fundamental in graph theory and algorithms [19]. A spanning tree uses the minimum number of edges that interconnect the vertices of a graph, which provides an economic structure for efficient processing in many applications, such as in network communication and network routing [13].

A graph can have many different spanning trees [5], but a specific application may particularly be interested in making use of a spanning tree with certain special properties. As a consequence, construction of spanning trees that optimize certain objective functions has been a popular topic in the research in combinatorial optimization [19]. In particular, spanning trees of the minimum weight have been extensively studied [1]. Other optimization problems on spanning trees include Maximum Bandwidth Spanning Tree, Maximum Internal Spanning Tree, Minimum Internal Spanning Tree, Minimum Diameter Spanning Tree, and Minimum Dilation Spanning Tree [10, 11, 17, 19].

In the current paper, we are focused on the Maximum Internal Spanning Tree problem (MaxIST), which is for a given graph to construct a spanning tree with the maximum number of internal nodes. The problem is an obvious

* This work is supported by the National Natural Science Foundation of China under Grant (61173051, 61103033, 71221061).

generalization of the famous NP-hard Hamiltonian-Path problem [7], thus, is NP-hard. The MaxIST problem has applications in the design of cost-efficient communication networks [18] and in minimizing turbulence in supply networks [2]. Recently, there have been considerable interests in the study of algorithms for MaxIST (see the survey by Salamon [17]), such as exact algorithms [2, 12], parameterized algorithms [2–4, 15], and approximation algorithms [9, 14, 16, 18].

In particular, approximation algorithms for MaxIST have gone through a number of rounds of improvements. Prieto and Sloper [14], based on local search, gave a 2-approximation algorithm for MaxIST. Salamon and Wiener [18] presented a linear-time 2-approximation algorithm for MaxIST based on depth first search. By refining the algorithm, they were also able to obtain a (3/2)-approximation algorithm for claw-free graphs and a (6/5)-approximation algorithm for cubic graphs [18]. For graphs without pendant nodes, Salamon [16] developed a (7/4)-approximation algorithm. Recently, Knauer and Spoerhase simplified and revised Salamon's algorithm in [16], and presented a (5/3)-approximation algorithm for general graphs [9].

All these algorithms are based on the techniques of local search/improvements, which have been a popular method used in practice and in theoretical study in dealing with computationally intractable problems [8]. Let G be a graph and let T be a spanning tree of G. If we pick an edge e in $G \setminus T$ and replace in T any edge in the unique cycle in $T + e$ by e, then we get another spanning tree for G. We will call such an operation an *edge swapping*, which has been the basic operation used by all approximation algorithms for the MaxIST problem [9, 14, 16, 18]. Therefore, in order to construct a spanning tree with many internal nodes for a graph, we can repeatedly apply the edge swapping operation, as long as the operation results in an improved spanning tree. In particular, both Prieto-Sloper algorithm [14] and Salamon-Wiener algorithm [18] use this simple strategy to achieve 2-approximation algorithms for the MaxIST problem.

Thus, edge swapping can be regarded as a basic step for local search processes of *depth* 1 on spanning trees. It is reasonable to consider local search processes of depth k, with $k > 1$, which examines a sequence of up to k consecutive edge swappings to seek a possible improvement on a spanning tree. Intuitively, a local search process of larger depth will result in a better spanning tree. This is indeed the idea used by Knauer and Spoerhase [9], who presented a local search process of depth 2 and proved that the process leads to a (5/3)-approximation algorithm for MaxIST. We also remark that the (7/4)-approximation algorithm for MaxIST on graphs without pendant nodes by Salamon [16] uses a large number of local improvement operations (14 improvement rules), in which one of the operations has depth 3 in terms of our characterization. However, as demonstrated by Knauer and Spoerhase [9], Salamon's algorithm [16] does not perform better than Knauer-Spoerhase algorithm [9] in the worst case.

Although local search process with further larger depth seems conceivably to lead to further improved spanning trees, it also presents a great challenge for the analysis to confirm the improvement. To see how the depth complicates the analysis, the readers may compare the analysis (of a few lines) for the depth-1 local search processes in [14, 18] with the analysis (of more than 5 pages) for the depth-2 local search process in [9]. Therefore, the questions raised in this direction are: (1) Is it feasible to analyze a local search process of depth larger than 2? and (2) If possible, how well can such a local search process do?

The current paper makes contributions to answering these questions. We propose a small set of simple local search rules, of which some has depth up to 5 in terms of the above characterization. We then present analysis to formally prove that these local search rules lead to a (3/2)-approximation algorithm for the MaxIST problem, thus improving all previous results. To achieve this, the concept of "quasi-branch" is introduced. Roughly speaking, quasi-branches are degree-2 nodes in a spanning tree that can be treated as nodes of degree larger than 2 in the tree in a local search of depth larger than 1. The concept of quasi-branches allows the edge swapping operation to become applicable on a much larger range of structures in the spanning tree. We also classify the other degree-2 internal nodes in a spanning tree into "live" and "dead" nodes, which enable us to apply edge swappings that, although not directly producing better spanning trees, lead to spanning trees of the same quality for further potential improvements. These new concepts and related structures reveal very rich and interesting combinatorial structures of spanning trees.

The paper is organized as follows. Section 2 presents our local search rules and verifies their validity. Section 3 discusses the structural properties of irreducible spanning trees, i.e., spanning trees on which none of the local search rules is applicable. Section 4 presents the proof to show that our approximation algorithm based on irreducible spanning trees has an approximation ratio bounded by 3/2. Section 5 concludes the paper with problems for further research.

2 The Local Search Rules

Our graphs are always simple, undirected, and connected. Let G be a graph and let v be a node in a subgraph H of G. Denote by $N_H(v)$ the set of the neighbors of v in H and denote by $d_H(v) = |N_H(v)|$ the degree of v in H.

Let T be a spanning tree of the graph G. For a node u in T, if $d_T(u) \leq 1$, then u is a *leaf* of T, otherwise, u is an *internal node* of T. For two nodes u and v in the tree T, denote by $P_T(u, v)$ the unique path in T from u to v, and denote by $u^{\overset{T}{\to} v}$ the unique neighbor of u in the path $P_T(u, v)$. An internal node u of T is an R_T*-branch* (i.e., a *real branch*) in T if $d_T(u) \geq 3$. An edge in G but not in the spanning tree T is called a *cotree edge* for T. A cotree edge $[l, x]$ with l a leaf and x an internal node of T is called an *L-cotree edge* for T. We will always write an L-cotree edge $[l, x]$ such that the first node l of the edge is the leaf. We say that the L-cotree edge $[l, x]$ *defines* the path $P_T(l, x)$.

Let $[v_1, v_2]$ be an cotree edge for T. An *edge swapping* with $[v_1, v_2]$ on T is to add $[v_1, v_2]$ to T and remove an edge $[w_1, w_2]$ on the path $P_T(v_1, v_2)$ (so the edge swapping is implemented *on* the edges $[v_1, v_2]$ and $[w_1, w_2]$). An edge swapping on a spanning tree of a graph always results in a spanning tree of the graph. An edge swapping is *improving*, *weakening*, and *holding*, respectively, if it increases, decreases, and unchanges the number of internal nodes in a spanning tree.

We now present our local search rules, which form the basis for our (3/2)-approximation algorithm for MaxIST. There are five rules and we assume that the rules are applied in order, i.e, Rule j will not be applied unless all Rules i for $i < j$ become unapplicable. Compared with the previous work [9, 14, 16, 18], our algorithm takes a much deeper local search strategy that in some cases considers a sequence of up to five consecutive edge swappings to seek an improvement on a spanning tree. Let G be a graph and let T be a given spanning tree of G. Without loss of generality, we assume that the graph G has more than two nodes and that the spanning tree T for G is not a path – otherwise the MaxIST problem for G can be trivially solved. Therefore, there is at least one R_T-branch in T.

Rule 1 (local search of depth 1). If there is an improving edge swapping on T, then apply the edge swapping.

Note that a cotree edge $[l_1, l_2]$ between two leaves of T can always induce an improving edge swapping: let v be any R_T-branch on the path $P_T(l_1, l_2)$. Then swapping the cotree edge $[l_1, l_2]$ and an edge on $P_T(l_1, l_2)$ that is incident to v will always give an improved spanning tree.

Suppose the improving edge swapping is on an L-cotree edge $[l, x]$ and an edge $[v_1, v_2]$ on the path $P_T(l, x)$, where $v_2 = v_1^{\xrightarrow{T} x}$. To obtain an improved spanning tree, both nodes v_1 and v_2 must have degree at least 3 in $T + [l, x]$. Therefore, v_1 must be an R_T-branch, while v_2 must be either an R_T-branch or $v_2 = x$.

For the convenience of our later reference, we distinguish the above two subcases in Rule 1 and split the rule into two detailed subrules:

Rule 1.1. If there is a cotree edge $[l_1, l_2]$ between two leaves l_1 and l_2 of T, then find an R_T-branch v on the path $P_T(l_1, l_2)$, swap $[l_1, l_2]$ and an edge on $P_T(l_1, l_2)$ that is incident to v.

Rule 1.2. If there is an L-cotree edge $[l, x]$ and an edge $[v_1, v_2]$ on the path $P_T(l, x)$, where $v_2 = v_1^{\xrightarrow{T} x}$, such that either both v_1 and v_2 are R_T-branches, or v_1 is an R_T-branch and $v_2 = x$, then swap $[l, x]$ and $[v_1, v_2]$.

Rule 1.1 has been used in all previous approximation algorithms for the MaxIST problem. A restricted version of Rule 1.2 has been used in [9, 16].

Note that a cotree edge connecting two internal nodes of T can never induce an improving edge swapping.

Rule 2 (local search of depth 2). If a holding edge swapping with an L-cotree edge creates a new cotree edge between two leaves, then apply the holding edge swapping, then apply Rule 1.1 on the created cotree edge between leaves.

Suppose that the holding edge swapping is on an L-cotree edge $[l, x]$ and an edge $[v_1, v_2]$ on the path $P_T(l, x)$. In order for the edge swapping to be holding, exact one of v_1 and v_2 has degree larger than 2 in $T + [l, x]$. Let v' be the one in $\{v_1, v_2\}$ that has degree 2 in $T + [l, x]$, then the cotree edge between leaves created by the edge swapping must have v' as one of its ends (the cotree edge cannot be between two leaves in T – otherwise, Rule 1.1 would be applied). Thus, in this case, there must be another leaf l' of T, $l' \neq l$, such that $[v', l']$ is an L-cotree edge for T. The following refined subrules of Rule 2 distinguish the subcases for $v' = v_2$ and $v' = v_1$. Thus, $[l, x]$ is an L-cotree edge, $[v_1, v_2]$ is an edge on the path $P_T(l, x)$, where $v_2 = v_1 \xrightarrow{T} x$.

Rule 2.1. If v_1 is an R_T-branch, $d_T(v_2) = 2$, $v_2 \neq x$, and there is an L-cotree edge $[l', v_2]$ with $l' \neq l$, then swap $[l, x]$ and $[v_1, v_2]$, and apply Rule 1.1 to $[l', v_2]$.

Rule 2.2. If $d_T(v_1) = 2$ with an L-cotree edge $[l', v_1]$, where $l' \neq l$, $v_2 = x$ or v_2 is an R_T-branch, then swap $[l, x]$ and $[v_1, v_2]$, and apply Rule 1.1 to $[l', v_1]$.

Rule 3 (local search of depth 3). If a weakening edge swapping with an L-cotree edge creates two cotree edges on two disjoint pairs of leaves, then apply the weakening edge swapping, and apply Rule 1.1 to each of the created cotree edges between leaves.

Rule 4 (local search of depth up to 4). If a holding edge swapping with an L-cotree edge results in a spanning tree on which Rule i becomes applicable, where $1 \leq i \leq 3$, then apply the holding edge swapping, and apply Rule i.

Rule 5 (local search of depth up to 5). If a weakening edge swapping with an L-cotree edge leads to a spanning tree on which applying Rule 1.1 produces a spanning tree T' such that Rule i is applicable on T', where $1 \leq i \leq 3$, then apply the edge swapping, Rule 1.1, and Rule i.

Lemma 1. *On any given spanning tree of the graph G, applying each of Rules 1-5 increases the number of internal nodes of a spanning tree by at least* 1.

A spanning tree T of the graph G is *irreducible* if none of Rules 1-5 is applicable on T. Our approximation algorithm for MaxIST starts with an arbitrary spanning tree of the graph G, and repeatedly applies Rules 1-5, until an irreducible spanning tree is obtained. It is easy to verify that in polynomial time we can check if any of Rules 1-5 is applicable and, in case there is an applicable rule, apply the rule. By Lemma 1, each application of a rule in Rules 1-5 increases the number of internal nodes of a spanning tree by at least 1. Since a spanning tree of the graph G of n nodes has at most $n - 2$ internal nodes, we have

Theorem 1. *An irreducible spanning tree of a graph can be constructed in polynomial time.*

3 On Irreducible Spanning Trees

In this section, we present a number of properties for irreducible spanning trees, which will be useful for us to show how well an irreducible spanning tree can approximate an optimal spanning tree for the MaxIST problem. The study also shows very rich and interesting combinatorial structures of irreducible spanning trees of a graph. Let T be a fixed irreducible spanning tree of the graph G. Assume that T is not a path (otherwise, T is an optimal spanning tree for G).

Definition 1. An internal node u in T with $d_T(u) = 2$ is a *Q_T-branch* (i.e., a *quasi-branch*) if there is an L-cotree edge $[l, u]$ for which the path $P_T(l, u)$

contains an internal node x such that either x is an R_T-branch in T or there is an L-cotree edge $[l', x]$ for T with $l' \neq l$ (x can be u).

The concept of Q_T-branches distinguishes our work from all previous approximation algorithms for MaxIST. In many cases, a Q_T-branch can be treated as an R_T-branch in a local search process of depth larger than 1.

An internal node of T is a *T-branch* if it is either an R_T-branch or a Q_T-branch in T. An internal node of T that is not a T-branch is called a *T-forward.* For a leaf l of T, denote by $b(l)$ the first T-branch we will encounter when we traverse the tree T, starting from the leaf l. Note that if T is not a simple path, then there must be at least one R_T-branch in T and all nodes we encounter between l and $b(l)$ during our traversing are T-forwards that are of degree 2 in T. Therefore, the node $b(l)$ is always well-defined.

Removing an edge $[u, v]$ in the tree T splits T into two subtrees. We denote by $T_{-[u,v]}(u)$ the subtree that contains u, and by $T_{-[u,v]}(v)$ the subtree that contains v. Similarly, removing edges of an edge subset E' in the tree T breaks T into a forest, which is denoted by $T_{-E'}$, and we denote by $T_{-E'}(x)$ the connected component of the forest $T_{-E'}$ that contains the node x.

Definition 2. An edge $[u, v]$ in the spanning tree T is a *T-bridge* if (1) there is no L-cotree edge $[l, x]$ such that the path $P_T(l, x)$ contains the edge $[u, v]$; and (2) each of $T_{-[u,v]}(u)$ and $T_{-[u,v]}(v)$ contains at least one T-branch.

Lemma 2. *Let $[x_1, x_2]$ be an edge in T that is not a T-bridge, where x_1 and x_2 are T-branches. Then there is an L-cotree edge $[l, x]$ with $[x_1, x_2]$ on the path $P_T(l, x)$ such that if $x_2 = x_1^{\xrightarrow{T} x}$, then x_1 is a Q_T-branch and $[l, x_1]$ is the only L-cotree edge incident to x_1 in G.*

The situation of the node x_1 in Lemma 2 turns out to be very special for counting the number of T-branches in T. For this, we give it a specific name.

Definition 3. A Q_T-branch x_1 is a *bad T-branch* if it is adjacent to a T-branch x_2 and there is an L-cotree edge $[l, x]$ such that the edge $[x_1, x_2]$ is on the path $P_T(l, x)$, $x_2 = x_1^{\xrightarrow{T} x}$, and $[l, x_1]$ is the only L-cotree edge incident to x_1 in G.

Remark 1. The node x_2 above can be assumed to be in one of the following cases: (a) x_2 is an R_T-branch; (b) $x_2 = x$; or (c) x_2 is a Q_T-branch with an L-cotree edge $[l', x_2]$, $l' \neq l$.

A T-branch is *good* if it is not a bad T-branch. By definition, an R_T-branch is always a good T-branch. A leaf of T is a *bad T-leaf* if it is adjacent in G to a bad T-branch. From Lemma 2, we get immediately:

Corollary 1. *Let $[x_1, x_2]$ be an edge in T, where x_1 and x_2 are T-branches. If $[x_1, x_2]$ is not a T-bridge, then at least one of x_1 and x_2 is a bad T-branch.*

Recall that for a leaf l, $b(l)$ is the T-branch closest to l in the tree T. Observe that if $[l, b(l)]$ is an edge in T, then $b(l)$ must be a good T-branch. In fact, if $b(l)$ were a bad T-branch, then $b(l)$ is a Q_T-branch adjacent to a T-branch x_2

such that the edge $[b(l), x_2]$ is on a path $P_T(l', x)$ defined by an L-cotree edge $[l', x]$ with $x_2 = b(l)^{\xrightarrow{T} x}$ and $[l', b(l)]$ is the only L-cotree edge incident to $b(l)$ in G. Since G is a simple graph, l' cannot be l. Thus, the path $P_T(l', x)$, which contains $[b(l), x_2]$ with $x_2 = b(l)^{\xrightarrow{T} x}$ but not the leaf l, would make $b(l)$ to have degree at least 3 in T, contradicting the assumption that $b(l)$ is a Q_T-branch.

As a consequence, a bad T-branch x in T is adjacent in the graph G to exact one leaf l of the spanning tree T, and $[l, x]$ must be an L-cotree edge for T.

Lemma 3. *Let v be a T-forward in T and let l be a leaf of T. If v is not on the path $P_T(l, b(l))$, then v and l are not adjacent in the graph G.*

A leaf l is a *long leaf* if $[l, b(l)]$ is not an edge in T. Otherwise, l is a *short leaf*. A long leaf l is *hung* if the edge $[b(l)^{\xrightarrow{T} l}, b(l)]$ is not on a path defined by an L-cotree edge for T, otherwise, l is *unhung*. Note that by Lemma 3, if a long leaf l is unhung, then the L-cotree edge that defines a path containing $[b(l)^{\xrightarrow{T} l}, b(l)]$ must have the leaf l as an end.

Lemma 4. *If a short leaf l is not a bad T-leaf, then all its neighbors in G are good T-branches.*

Now we consider the T-forwards, which can be classified into two classes. The nodes of each class can be further classified into "live" and "dead" nodes.

Definition 4. A T-forward is a *T_I-forward* if it is not on the path $P_T(l, b(l))$ for any leaf l of T. A T_I-forward v is *live* if there is an edge $[v, v']$ on the path $P_T(l, x)$ defined by an L-cotree edge $[l, x]$ for T such that either v' is an R_T-branch, or v' is a Q_T-branch and $v' = v^{\xrightarrow{T} x}$. The T_1-forward v is *dead* if it is not live.

Remark 2. The node v' in the above definition for the live T_I-forward v can be assumed to be in one of the following three cases: (a) v' is an R_T-branch; (b) $v' = x$; and (c) $v' = v^{\xrightarrow{T} x}$ and v' is a Q_T-branch with an L-cotree edge $[l', v']$, $l' \neq l$.

Definition 5. A T-forward is a *T_L-forward* if it is on the path $P_T(l, b(l))$ for a leaf l of T. A T_L-forward v is *live* if either $[l, v^{\xrightarrow{T} b(l)}]$ is an edge in G or $v^{\xrightarrow{T} b(l)} = b(l)$ and l is an unhung long leaf of T. The T_1-forward v is *dead* if it is not live.

Lemma 5. *For a live T_L-forward v on the path $P_T(l, b(l))$ for a leaf l of T, there is an L-cotree edge $[l, x]$ with the path $P_T(l, x)$ containing $[v, v^{\xrightarrow{T} b(l)}]$, such that $v^{\xrightarrow{T} x}$ is in one of the following three cases: (a) $v^{\xrightarrow{T} x}$ is an R_T-branch; (b) $v^{\xrightarrow{T} x} = x$; and (c) $v^{\xrightarrow{T} x}$ is a Q_T-branch with an L-cotree edge $[l', v^{\xrightarrow{T} x}]$, $l' \neq l$.*

A T-forward v is *live* if it is either a live T_I-forward or a live T_L-forward. A T-forward that is not live is a *dead T-forward*. Live T-forwards provide another important concept that distinguishes our work from others. Although a live T-forward does not directly induce improving edge swappings, it may introduce holding edge swappings that lead to spanning trees with more desired combinatorial structures.

Let B be the set of all T-bridges in the spanning tree T. Let $F_T = T_{-B} = \{\tau_1, \tau_2, \cdots, \tau_h\}$, where each τ_i is a connected component of F_T (thus, each τ_i is a tree). F_T will be called the *T-forest.* The following lemma follows directly from the definition of T-bridges.

Lemma 6. *No L-cotree edge for T can cross two connected components of F_T.*

Remark 3. Let $[z_1, z_2]$ be an edge in the tree T, such that z_1 is a good T-branch and z_2 is a T-branch. If there is an L-cotree edge $[l, z_2]$, then the leaf l cannot be in $T_{-[z_1,\, z_2]}(z_1)$.
Remark 4. Let $[z_1, z_2]$ be an edge in the tree T, where z_1 and z_2 are T-branches with L-cotree edges $[l_1, z_1]$ and $[l_2, z_2]$. If l_2 is in $T_{-[z_1,\, z_2]}(z_1)$, then $l_1 = l_2$.

Lemma 7. *For any two good T-branches x_1 and x_2 in the same connected component of F_T, there is at least one live T_I-forward on the path $P_T(x_1, x_2)$.*

Now we consider edges between internal nodes of the spanning tree T. For a leaf l of T, let $T_L(l)$ be the set of all live T_L-forwards on the path $P_T(l, b(l))$.

Lemma 8. *Let $v_1 \in T_L(l)$, where l is a hung long leaf of T. For any node v_2 that is either a bad T-branch or a live T-forward but not in $T_L(l)$, there is no edge between v_1 and v_2 in G.*

Our last lemma in this section considers the adjacency relation between two nodes that belong to different connected components in the T-forest F_T.

Lemma 9. *Suppose that each of the nodes v_1 and v_2 is either a bad T-branch or a live T-forward. If v_1 and v_2 are in different connected components in the T-forest F_T, then there is no edge between v_1 and v_2 in the graph G.*

4 An Approximation Algorithm of Ratio 1.5

In this section, we show that the number of internal nodes in a maximum internal spanning tree of a graph G is no more than 1.5 of that in an irreducible spanning tree T of G. This result combined with Theorem 1 leads directly to a polynomial-time approximation algorithm of ratio 1.5 for the MaxIST problem. Our analysis is involved in nontrivial comparisons on combinatorial structures of different spanning trees of G. To avoid confusion, for a subgraph H of G, we will say that a node v is *H-adjacent* to a node w if $[v, w]$ is an edge in the subgraph H.

Let T^* be an optimal spanning tree of the graph G for the MaxIST problem, that is, T^* has the maximum number of internal nodes over all spanning trees of G. Let T be a fixed irreducible spanning tree of G.

We divide the internal nodes of the irreducible spanning tree T of G into six disjoint sets: (1) The set of all good T-branches in T; (2) The set of all bad T-branches in T; (3) The set of all live T_L-forwards in T; (4) The set of all dead T_L-forwards in T; (5) The set of all live T_I-forwards in T; and (6) The set of all dead T_I-forwards in T.

Let I_{gb} be the set of all good T-branches in T, and let $F_T^* = T^* \setminus I_{gb} = \{\tau_1^*, \cdots, \tau_r^*\}$, which will be called the T^*-*forest*, where each τ_i^* is a connected component of F_T^*. Recall that the T-forest $F_T = T_{-B}$ is the forest obtained from the irreducible spanning tree T with the set B of all T-bridges removed.

Lemma 10. *Suppose that in the T^*-forest $F_T^* = T^* \setminus I_{gb} = \{\tau_1^*, \cdots, \tau_r^*\}$, for each i, $1 \le i \le r$, there are m_i edges in T^* between τ_i^* and I_{gb}. Then $\sum_{i=1}^{r}(m_i - 1) \le |I_{gb}| - 1$.*

Now we consider the neighborhood structures of a leaf l of T in T^*.

Lemma 11. *Let l be a leaf of T, and let $[l, v]$ be an edge in G. Then l and v are in the same connected component of F_T, and v is neither a leaf nor a T_I-forward in T. Moreover, if l is a short leaf of T but not a bad T-leaf, then l makes a single-node connected component of the T^*-forest F_T^*.*

Lemma 12. *If two leaves l_1 and l_2 of T are in the same connected component τ_i^* of the T^*-forest F_T^*, then no node in T^* can be T^*-adjacent to both l_1 and l_2.*

A leaf l of T is an *inside-leaf* (w.r.t. T^*) if $d_{T^*}(l) \ge 2$ and each component of $T^* \setminus l$ contains at least one leaf of T. Otherwise, l is an *outside-leaf* of T.

Let l be a leaf of T and let v be a node in $F_T^* = T^* \setminus I_{gb}$ such that $[l, v]$ is an edge in G. By Lemma 11, v is either a bad T-branch or a T_L-forward. Based on this, we can classify the nodes in F_T^* that are related to a leaf of T as follows.

Fix a connected component τ_i^* of the T^*-forest F_T^*, and let l be a leaf of T that is in τ_i^*. Recall that $T_L(l)$ is the set of live T_L-forwards in T on the path $P_T(l, b(l))$. We also denote by $T_B(l)$ the set of bad T-branches in T that are G-adjacent to l. Let L_i^x and L_i^o, respectively, be the set of inside-leaves and the set of outside-leaves of T in the tree τ_i^*, and let $L_i = L_i^x \cup L_i^o$. Naturally, let $T_L(L_i) = \bigcup_{l \in L_i} T_L(l)$ and $T_B(L_i) = \bigcup_{l \in L_i} T_B(l)$. Finally, let H_i be the set $T_L(L_i) \cup T_B(L_i)$ plus all nodes in τ_i^* that are dead T-forwards in T.

Lemma 13. *Let m_i be the number of edges in T^* between the tree τ_i^* and the set I_{gb}. Then $|H_i| + m_i \ge 2|L_i^x|$. Moreover, if τ_i^* is T^*-adjacent to at most one good T-branch in each component of the T-forest F_T, then $|H_i| + m_i \ge 2|L_i^x| + 1$.*

Lemma 13 gives, in terms of the number of inside-leaves in the component τ_i^* of the T^*-forest F_T^*, a lower bound on the size of the set H_i, which contains bad T-branches, dead T-forwards, and live T_L-forwards in T. The following lemma gives a lower bound on the number of live T_I-forwards in T.

Lemma 14. *If there are q connected components in the T^*-forest F_T^* such that each of them is T^*-adjacent to at least two good T-branches in some connected component of the T-forest F_T, then there are at least q live T_I-forwards in T.*

Now we are ready for our main result of this paper.

Theorem 2. *Let β^* be the number of internal nodes in the optimal spanning tree T^* and let β_T be the number of internal nodes in the irreducible spanning tree T. Then $\beta^*/\beta_T < 1.5$.*

Proof. Recall for a component τ_i^* of the T^*-forest $F_T^* = T^* \setminus I_{gb} = \{\tau_1^*, \ldots, \tau_r^*\}$, L_i^x is the set of inside-leaves of T in τ_i^*, L_i^o is the set of outside-leaves of T in τ_i^*, $L_i = L_i^x \cup L_i^o$, H_i is the set $T_L(L_i) \cup T_B(L_i)$ plus all dead T-forwards in τ_i^*, and m_i is the number of edges in T^* between τ_i^* and I_{gb}. First observe that for $i \neq j$, the sets H_i and H_j are disjoint, because: (1) a dead T-forward in H_i is in τ_i^* that is disjoint with τ_j^*; (2) a live T_L-forward in $T_L(l)$ for a leaf l in L_i is on the path $P_T(l, b(l))$ so it cannot be in $T_L(l')$ for a leaf l' in L_j; and (3) a bad T-branch in $T_B(l)$ for a leaf l in L_i cannot be in $T_B(l')$ for a leaf l' in L_j because by definition a bad T-branch is G-adjacent to only one leaf in T.

Without loss of generality, let q, $0 \leq q \leq r$, be such an integer that for $1 \leq i \leq q$, the connected component τ_i^* is T^*-adjacent to at most one good T-branch in each connected component of the T-forest F_T, and for $q+1 \leq j \leq r$, the connected component τ_j^* is T^*-adjacent to at least two good T-branches in some connected component of F_T. Finally, let $L^x = \bigcup_{i=1}^r L_i^x$ be the set of all inside-leaves of T, and let I_{li} be the set of all live T_I-forwards in T.

By Lemma 13, we have

$$\sum_{i=1}^{r}(|H_i| + m_i) = \sum_{i=1}^{q}(|H_i| + m_i) + \sum_{i=q+1}^{r}(|H_i| + m_i)$$
$$\geq \sum_{i=1}^{q}(2|L_i^x| + 1) + \sum_{i=q+1}^{r} 2|L_i^x| = 2\sum_{i=1}^{r}|L_i^x| + q = 2|L^x| + q. \tag{1}$$

By Lemma 10 and Lemma 14, we have

$$\sum_{i=1}^{r}(m_i - 1) \leq |I_{gb}| - 1, \qquad \text{and} \qquad |I_{li}| \geq r - q. \tag{2}$$

The set $\bigcup_{i=1}^r H_i$ consists of bad T-branches, T_L-forwards, and dead T-forwards in T, while I_{gb} is the set of good T-branches in T, and I_{li} is the set of live T_I-forwards in T. These sets are disjoint sets of internal nodes of T. Therefore,

$$\beta_T \geq |\bigcup_{i=1}^{r} H_i \cup I_{gb} \cup I_{li}| = |\bigcup_{i=1}^{r} H_i| + |I_{gb}| + |I_{li}| = \sum_{i=1}^{r}|H_i| + |I_{gb}| + |I_{li}|, \tag{3}$$

where the last equality comes from the pairwise disjointness of H_i, $1 \leq i \leq r$.

Combining (3) with (1) and (2), we get

$$\beta_T \geq (2|L^x| + q - \sum_{i=1}^{r} m_i) + (\sum_{i=1}^{r}(m_i - 1) + 1) + (r - q) = 2|L^x| + 1 > 2|L^x|. \tag{4}$$

This provides a lower bound on β_T in terms of the number of inside-leaves of T.

The outside-leaves of T are mapped to leaves of T^*, as follows. Let l be an outside-leaf of T. If l is also a leaf of T^*, then let l be mapped to itself. If l is an internal node of T^*, then by definition, at least one of the connected components of $T^* \setminus l$ contains no leaves of T. In this component, we construct a path P_l in

T^* from l to a(ny) leaf l^* of T^*, and map l to l^*. Note that for two different outside-leaves l_1 and l_2 of T, the two paths P_{l_1} and P_{l_2} in T^* are node-disjoint: otherwise, the component of $T^* \setminus l_1$ containing the path P_{l_1} would contain the leaf l_2. The disjointness of these paths shows that the mapping we constructed is an injective mapping from the set L^o of outside-leaves of T to the set L^* of leaves of T^*. Thus, $|L^o| \leq |L^*|$. From this and using (4), we get (where n is the total number of nodes in the graph G):

$$\beta^* = n - |L^*| \leq n - |L^o| = \beta_T + |L^x| < \beta_T + \beta_T/2 = 1.5 \cdot \beta_T. \qquad \square$$

Corollary 2. *There is a polynomial-time approximation algorithm of approximation ratio bounded by* 1.5 *for the* MaxIST *problem.*

5 Conclusion

We have presented a polynomial-time approximation algorithm for the MaxIST problem. Our algorithm is based on a deep local search strategy that allows local search with up to five consecutive edge swapping operations. We identified proper combinatorial structures to support the analysis of the approximability of such deep local search strategies. We were able to formally prove that our approximation algorithm has a ratio bounded by 1.5.

The combinatorial structures, such as Q_T-branches, dead T-branches, and live T-forwards, have shown rich and interesting properties of spanning trees of graphs, which may also be useful for other studies on graph spanning trees.

References

1. Bazlamacci, C., Hindi, K.: Minimum-weight spanning tree algorithms: a survey and empirical study. Computer & Operations Research 28, 767–785 (2001)
2. Binkele-Raible, D., Fernau, H., Gaspers, S., Raible, D.: Exact and parameterized algorithms for max internal spanning tree. Algorithmica 65(1), 95–128 (2013)
3. Cohen, N., Fomin, F., Gutin, G., Kim, E., Saurabh, S., Yeo, A.: Algorithm for finding k-vertex out-trees and its application to k-internal out-branching broblem. J. Comput. Syst. Sci. 76(7), 650–662 (2010)
4. Fomin, F., Gaspers, S., Saurabh, S., Thomassé, S.: A linear vertex kernel for maximum internal spanning tree. J. Comput. Syst. Sci. 79(1), 1–6 (2012)
5. Gabow, H., Myers, E.: Finding all spanning trees of directed and undirected graphs. SIAM J. Comput. 7(3), 280–287 (1978)
6. Galbiati, G., Maffioli, F., Morzenti, A.: A short note on the approximability of the maximum leaves spanning tree problem. Inform. Process. Lett. 52, 45–49 (1994)
7. Garey, M., Johnson, D.: Computers and Intractability: A Guide to the Thoery of NP-completeness. Freeman, W. H. and Company, New York (1979)
8. Johnson, D., Papadimitriou, C., Yanakakis, M.: How easy is local search? J. Comput. Syst. Sci. 37, 79–100 (1988)
9. Knauer, M., Spoerhase, J.: Better Approximation Algorithms for the Maximum Internal Spanning Tree Problem. In: Dehne, F., Gavrilova, M., Sack, J.-R., Tóth, C.D. (eds.) WADS 2009. LNCS, vol. 5664, pp. 459–470. Springer, Heidelberg (2009)

10. Lu, H., Ravi, R.: Approximating maximum leaf spanning trees in almost linear time. J. Algorithms 29, 132–141 (1998)
11. Malpani, N., Chen, J.: A note on practical construction of maximum bandwidth paths. Inf. Process. Lett. 83(3), 175–180 (2002)
12. Nederlof, J.: Fast polynomial-space algorithms using inclusion-exclusion: improving on Steiner tree and related problems. Algorithmica 65(4), 868–884 (2013)
13. Perlman, R.: Interconnections: Bridges, Routers, Switches, and Internetworking Protocols, ch. 3, 2nd edn. Addison-Wesley (2000)
14. Prieto, E., Sloper, C.: Either/or: Using vertex cover structure in designing FPT-algorithms – the case of k-internal spanning tree. In: Dehne, F., Sack, J.-R., Smid, M. (eds.) WADS 2003. LNCS, vol. 2748, pp. 474–483. Springer, Heidelberg (2003)
15. Prieto, E., Sloper, C.: Reducing to independent set structure – the case of k-internal spanning tree. Nord. J. Comput. 12(3), 308–318 (2005)
16. Salamon, G.: Approximating the maximum internal spanning tree problem. Theor. Comput. Sci. 410, 5273–5284 (2009)
17. Salamon, G.: A survey on algorithms for the maximum internal tree and related problem. Electron. Notes Discrete Math. 36, 1209–1216 (2010)
18. Salamon, G., Wiener, G.: On finding spanning trees with few leaves. Information Processing Letters 105, 164–169 (2008)
19. Wu, B., Chao, K.-M.: Spanning Trees and Optimization Problems. CRC Press, Boca Raton (2004)

FPTAS for Counting Weighted Edge Covers

Jingcheng Liu[1], Pinyan Lu[2], and Chihao Zhang[1,⋆]

[1] Shanghai Jiao Tong University
{liuexp,chihao.zhang}@gmail.com
[2] Microsoft Research
pinyanl@microsoft.com

Abstract. An edge cover of a graph is a set of edges in which each vertex has at least one of its incident edges. The problem of counting the number of edge covers is #**P**-complete and was shown to admit a fully polynomial-time approximation scheme (FPTAS) recently [10]. Counting weighted edge covers is the problem of computing the sum of the weights for all the edge covers, where the weight of each edge cover is defined to be the product of the edge weights of all the edges in the cover. The FPTAS in [10] cannot apply to general weighted counting for edge covers, which was stated as an open question there. Such weighted counting is generally interesting as for instance the weighted counting independent sets (vertex covers) problem has been exhaustively studied in both statistical physics and computer science. Weighted counting for edge cover is especially interesting as it is closely related to counting perfect matchings, which is a long-standing open question. In this paper, we obtain an FPTAS for counting general weighted edge covers, and thus solve an open question in [10]. Our algorithm also goes beyond that to certain generalization of edge cover.

1 Introduction

An edge cover for an undirected graph $G(V, E)$ is a set of edges $C \subseteq E$ such that for every $v \in V$, it holds that $N(v) \cap C \neq \varnothing$ where $N(v)$ is the set of edges incident to v. The problem of counting edge covers in an undirected graphs was known to be #**P**-hard and was recently shown to admit a fully polynomial-time approximation scheme (FPTAS)[10].

A natural generalization of the edge cover problem is to consider edge weights. That is, we assign a positive real number λ_e for every edge $e \in E$ and an edge cover C is of weight $w_C \triangleq \prod_{e \in C} \lambda_e$. Denote $EC(G)$ the set of edge covers of G, the problem of counting weighted edge covers is to compute

$$\sum_{C \in EC(G)} w_C = \sum_{C \in EC(G)} \prod_{e \in C} \lambda_e.$$

Such sum of product expression is usually called partition function in statistical physics, or graph polynomial in combinatorics, which are of great interests. For

⋆ The author is supported by NSF of China (61033002, ANR 61261130589).

example, if we replace the edge cover constraint with matching constraint, then we get the well-known matching polynomial. If we replace the constraint with vertex cover (or complementary independent set) constraint, and edge weights with vertex weights, we get the problem of counting weighted independent sets, which is also known as hard-core model in statistical physics. This problem is extensively studied and a complete understanding was not available until very recently [5,20,16,6,17]. There is a phase transition in terms of weights for the computational complexity of the problem. In [10], Lin et al. asked whether the problem of counting weighted edge covers also exhibits a phase transition in terms of edge weights. In particular, the method in [10] can be extended to that all edges are of uniform weight λ with $\lambda \geq \frac{4}{9}$, but not further. We answer the question by designing an FPTAS for counting weighted edge covers with arbitrary edge weights, even if they are not uniform, provided that they are constants.

In weighted edge covers, $\lambda_e > 1$ indicates that the edge e is preferred to be chosen and it is preferred not if $\lambda_e < 1$. If all the edge weights are the same and smaller than 1, then an edge cover with smaller cardinality contributes more in the sum. As the uniform edge weight approach zero (exponentially small in terms of the graph size), the weights from the minimum edge covers will dominate all the other terms. Provided that the graph has a perfect matching, the set of minimum edge covers is exactly the same as the set of perfect matchings. Therefore when the edge weights are exponentially small in terms of the graph size, the problem of counting weighted edge covers is essentially counting minimal edge covers, which is even stronger than counting perfect matchings, for which no polynomial-time approximation algorithm in general was known. It is widely open whether one can design an FPTAS for counting perfect matchings or not. But unfortunately, our FPTAS only works for constant edge weights, which is not exponentially small in terms of the input size.

It is worth noting that there is a similar situation for counting weighted matchings (not necessarily perfect). There is a fully polynomial-time randomized approximation scheme (FPRAS) for constant edge weights based on Markov chain Monte-Carlo method [7]. If one allows the weights go to infinity (exponentially large in terms of the graph size), counting matching is essentially the same as counting perfect matching provided that the graph has one since the contribution from those perfect matchings will dominate the others. But the known algorithm does not work for exponentially large weights either. In some sense, the constraint of perfect matching is upper and lower bounded by the constraints of edge cover and partial matching respectively. For both, we have approximate algorithms, while the perfect matching problem is widely open. With our new FPTAS for counting weighted edge covers, it is interesting to see that if we can play with these upper and lower bounds simultaneously to get an algorithm for counting perfect matchings. We remark that our algorithm for weighted edge cover is deterministic while the general algorithm for counting matchings is randomized and deterministic FPTAS is only known for graphs with bounded degree even in the unweighted setting [1].

We then consider another generalization of edge cover: We allow vertices stay uncovered in a "cover" and each of these (uncovered) vertices v contributes a weight (or penalty) $\mu_v \in [0,1]$ to the weight of the cover. Formally, we have

$$Z(G) \triangleq \sum_{\sigma \in \{0,1\}^E} \prod_{e \in E} \lambda_e^{\sigma(e)} \prod_{v \in V} \mu_v^{\delta(\sigma,v)},$$

where $\delta(\sigma, v)$ is defined to be

$$\delta(\sigma, v) = \begin{cases} 0, & \text{if } \sigma(e) = 1 \text{ for some } e \text{ incident to } v \\ 1, & \text{otherwise.} \end{cases}$$

This is similar to allowing omissible vertices for perfect matching [18]. The original edge cover is equivalent to the case that $\mu_v = 0$ for every $v \in V$. Our FPTAS can also be generalized to this generalization of weighted edge cover. Indeed, we shall state our theorem, algorithm and proof for this generalization directly and the ordinary weighted edge cover follows as a special case. Formally, we have the following main result.

Theorem 1. *For any constant $\lambda > 0$, there is an FPTAS to approximate $Z(G)$ for graphs $G(V, E)$ with edge weights $\lambda_e \geq \lambda$ for every edge $e \in E$ and vertex weights $\mu_v \in [0,1]$ for every $v \in V$.*

1.1 Related Works

Counting edge covers was previously studied in [2] where a Markov chain Monte-Carlo based algorithm was given for 3-regular graphs. Later in [10], an FPTAS for general graphs was proposed.

Our technique for designing FPTAS is the correlation decay method. The technique was proved to be very powerful in obtaining FPTAS for counting problems, some notable examples include [1,20,14,9,11,15,12]. An crucial ingredient of our analysis is the use of *potential function* (or called *message* in some literature) to amortize error propagated [13,8,14,9,15,11].

The problem of counting (perfect) matching, edge cover and our generalization with vertex weights can be uniformly treated in the framework of Holant problems [19,3,4].

2 Preliminaries

2.1 Dangling Edge

Following [10], we introduce dangling edges into our graph to simplify the description of our algorithm and proofs.

Definition 2. *A **dangling edge** $e = (u, \_)$ in a graph $G(V, E)$ is such an edge with exactly one end point $u \in V$.*

*A **free edge** $e = (\_, \_)$ is an edge with no end points.*

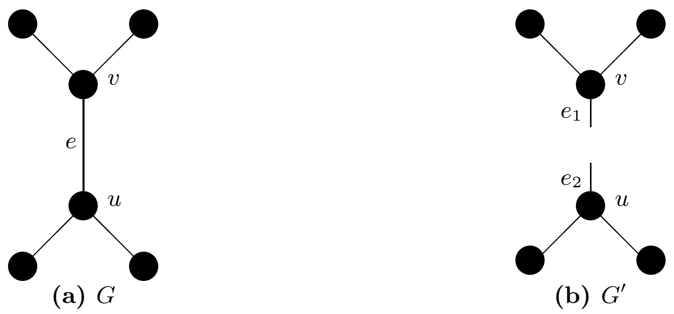


Fig. 1. Breaking up a normal edge into two dangling edges

A graph with two dangling edges e_1, e_2 is depicted in Figure 1b.

It is natural to generalize $Z(G)$ to graphs with dangling edges. For a graph $G = (V, E)$, an edge $e = (u, v) \in E$ and a vertex $u \in V$, define

$$\begin{aligned} G - e &\triangleq (V, E - e) \\ e - u &\triangleq (\_, v) \quad \text{(note that here } v \text{ could be } \_) \\ G - u &\triangleq (V - u, \\ &\qquad \{e \mid e \in E, e \text{ is not incident with } u\} \\ &\qquad \cup \{e - u \mid e \in E, e \text{ is incident with } u\}) \end{aligned}$$

The definition of $G - u$ indicates that all edges incident to u in G become dangling in $G - u$.

2.2 Approximate Counting from Estimation of Marginal Probabilities

The definition of the partition function naturally induces a Gibbs measure on all configurations over E. From this joint distribution, we can also define marginal probability for a (dangling) edge e. For $c \in \{0, 1\}$, we define

$$Z_{e=c}(G) \triangleq \sum_{\substack{\sigma \in \{0,1\}^E \\ \sigma(e)=c}} \prod_{v \in V} \mu_v^{\delta(\sigma,v)} \prod_{e \in E} \lambda_e^{\sigma(e)}$$

The marginal probability that e is chosen ($\sigma(e) = 1$) or not ($\sigma(e) = 0$) can be expressed as

$$\mathbb{P}_G(e = 0) \triangleq \frac{Z_{e=0}(G)}{Z(G)}, \quad \mathbb{P}_G(e = 1) \triangleq \frac{Z_{e=1}(G)}{Z(G)}.$$

It is a standard routine to approximate the partition function $Z(G)$ if the marginal probability can be well-estimated.

Proposition 3. *There is an FPTAS for approximating the partition function of weighted edge cover provided an oracle $\mathcal{O}$ to estimate $\mathbb{P}_G\left(e=1\right)$ where $G(V,E)$ is an arbitrary graph with (dangling) edge e. $\mathcal{O}$ takes input $G, e, \varepsilon > 0$ and is required to satisfy*

1. *$\mathcal{O}$ outputs an estimate $\hat{p}$ within time polynomial in $|G|$ and $1/\varepsilon$;*
2. *$\exp(-\varepsilon)\cdot\hat{p} \le \mathbb{P}_G\left(e=1\right) \le \exp(\varepsilon)\cdot\hat{p}$.*

Proof. Let $G(V,E)$ be a graph and we now give an algorithm to estimate $Z(G)$ with the help of the oracle. Let $\boldsymbol{\sigma} \in \{0,1\}^E$ be the configuration that $\boldsymbol{\sigma}(e)=1$ for every $e \in E$. Then

$$\mathbb{P}_G\left(\boldsymbol{\sigma}\right) = \frac{\omega_{\boldsymbol{\sigma}}}{Z(G)}$$

where $w_{\boldsymbol{\sigma}}$ is the weight of configuration $\boldsymbol{\sigma}$ and it is easily computable. Thus in order to compute $Z(G)$, it is sufficient to estimate $\mathbb{P}_G\left(\boldsymbol{\sigma}\right)$.

We fix an arbitrary order of edges in E, i.e., $E=\{e_1,\dots,e_m\}$ in which $e_i=(u_i,v_i)$ for every $1\le i\le m$. Then

$$\mathbb{P}_G\left(\boldsymbol{\sigma}\right) = \mathbb{P}_G\left(\bigwedge_{i=1}^{m}\sigma(e_i)=1\right) = \prod_{i=1}^{m}\mathbb{P}_G\left(e_i=1\,\middle|\,\bigwedge_{j=1}^{i-1}e_j=1\right).$$

Define $G_1 \triangleq G$, $G_i \triangleq G_{i-1}-e_{i-1}-u_{i-1}-v_{i-1}$, for $2\le i\le m$. We have $\mathbb{P}_G\left(e_i=1\,\middle|\,\bigwedge_{j=1}^{i-1}e_j=1\right) = \mathbb{P}_{G_i}\left(e_i=1\right)$. For every $1\le i\le m$, we call the oracle with input $\left(G_i, e_i, \frac{\varepsilon}{2|E|}\right)$. Let $\hat{p}_i$ be the result of our i-th call, $\hat{p}=\prod_i^m\hat{p}_i$ and $\hat{Z}=\frac{\omega_{\boldsymbol{\sigma}}}{\hat{p}}$, then it holds that

$$\exp(-\varepsilon)\cdot Z(G) \le \hat{Z} \le \exp(\varepsilon)\cdot Z(G).$$

□

3 Approximation for Marginal Probabilities

In this section, we prove Theorem 1. By Proposition 3, we only need to estimate the marginal probabilities as following:

Lemma 4. *Let $G(V,E)$ be an instance of weighted edge cover with an edge e, and vertex weight $\mu_v \le 1$, there is an algorithm $\mathcal{A}$ that efficiently approximates $\mathbb{P}_G\left(e=1\right)$. More precisely, $\mathcal{A}$ takes as input $G, e, \varepsilon > 0$ and the following holds:*

1. *$\mathcal{A}$ outputs an estimate $\hat{p}$ within time polynomial in $|G|$ and $1/\varepsilon$;*
2. *$\exp(-\varepsilon)\cdot\hat{p} \le \mathbb{P}_G\left(e=1\right) \le \exp(\varepsilon)\cdot\hat{p}$.*

The lemma together with Proposition 3 implies Theorem 1.

3.1 Computational Tree Recursion and the Algorithm

We use computational tree recursion to compute $\mathbb{P}_G(e=0)$, a good estimate of which is also a good estimate of $\mathbb{P}_G(e=1)$. We express $\mathbb{P}_G(e=0)$ as a function of marginal probabilities on smaller instances.

Free Edge. If e is a free edge, then $\mathbb{P}_G(e=0) = \frac{1}{1+\lambda_e}$.

Normal Edge. Assume $e=(u,v)$, we define a recursion to compute $R_G(e) \triangleq \frac{\mathbb{P}_G(e=1)}{\mathbb{P}_G(e=0)}$. Then $\mathbb{P}_G(e=0) = \frac{1}{1+R_G(e)}$.

To this end, we replace $e=(u,v)$ with dangling edges $e_1=(u,\_)$ and $e_2=(v,\_)$. Denote this new graph by $G'(V',E')$, as depicted in Figure 1a and 1b.

We further let $G_1 \triangleq G'-e_2$, $G_2 \triangleq G'-e_1-u$. It holds that

$$\begin{aligned}R_G(e) &= \frac{\mathbb{P}_{G'}(e_1=1,e_2=1)}{\mathbb{P}_{G'}(e_1=0,e_2=0)}\\ &= \frac{\mathbb{P}_{G'}(e_1=1,e_2=0)}{\mathbb{P}_{G'}(e_1=0,e_2=0)}\cdot\frac{\mathbb{P}_{G'}(e_1=1,e_2=1)}{\mathbb{P}_{G'}(e_1=1,e_2=0)}\\ &= \frac{\mathbb{P}_{G_1}(e_1=1)}{\mathbb{P}_{G_1}(e_1=0)}\cdot\frac{\mathbb{P}_{G_2}(e_2=1)}{\mathbb{P}_{G_2}(e_2=0)}\\ &= R_{G_1}(e_1)\cdot R_{G_2}(e_2).\end{aligned}$$

This directly gives the recursion for $\mathbb{P}_G(e=0)$:

$$\mathbb{P}_G(e=0) = \frac{\mathbb{P}_{G_1}(e_1=0)\,\mathbb{P}_{G_2}(e_2=0)}{1-\mathbb{P}_{G_1}(e_1=0)-\mathbb{P}_{G_2}(e_2=0)+2\mathbb{P}_{G_1}(e_1=0)\,\mathbb{P}_{G_2}(e_2=0)}.$$

We remark that in the RHS of the recursion, both e_1 and e_2 are dangling edges in G_1 and G_2 respectively.

Dangling Edge. Let $e=(u,\_)$ be the dangling edge. Denote $\hat{E}=\{e_i \mid 1\le i\le d\}$ the set of other edges incident to u. Let $G' \triangleq G-e-u$ as illustrated in Figure 2a and 2b.

Define a family of graphs $\{G_i\}_{1\le i\le d}$ obtained by removing edges in $\hat{E}$ consecutively: $G_1 \triangleq G'$, $G_i \triangleq G_{i-1}-e_{i-1}$, for $2\le i\le d$.

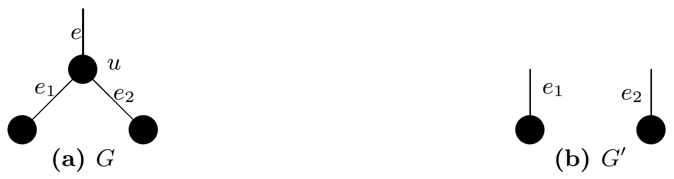


Fig. 2. Dangling edges examples

Let $\boldsymbol{\alpha} \in \{0,1\}^d$ be a configuration over $\hat{E}$. We use $Z_{\boldsymbol{\alpha}}(G)$ to denote the sum of weights over configurations of G whose restriction on $\hat{E}$ is consistent with $\boldsymbol{\alpha}$. Formally we let

$$Z_{\boldsymbol{\alpha}}(G) \triangleq \sum_{\substack{\sigma \in \{0,1\}^E \\ \sigma|_{\hat{E}} = \boldsymbol{\alpha}}} \prod_{v \in V} \mu_v^{\delta(\sigma,v)} \prod_{e \in E} \lambda_e^{\sigma(e)}.$$

Then by the definition of the marginal probability, we have

$$\begin{aligned}
\mathbb{P}_G\left(e=0\right) &= \frac{Z_{e=0}(G)}{Z_{e=0}(G) + Z_{e=1}(G)} \\
&= \frac{\mu_u Z_{\mathbf{0}}(G') + \sum_{\boldsymbol{\alpha} \in \{0,1\}^d, \boldsymbol{\alpha} \neq \mathbf{0}} Z_{\boldsymbol{\alpha}}(G')}{(\mu_u + \lambda_e) Z_{\mathbf{0}}(G') + (1+\lambda_e) \sum_{\boldsymbol{\alpha} \in \{0,1\}^d, \boldsymbol{\alpha} \neq \mathbf{0}} Z_{\boldsymbol{\alpha}}(G')} \\
&= \frac{Z(G') - (1-\mu_u) Z_{\mathbf{0}}(G')}{(1+\lambda_e) Z(G') - (1-\mu_u) Z_{\mathbf{0}}(G')} \\
&= \frac{1 - (1-\mu_u) \frac{Z_{\mathbf{0}}(G')}{Z(G')}}{1 + \lambda_e - (1-\mu_u) \frac{Z_{\mathbf{0}}(G')}{Z(G')}}.
\end{aligned} \tag{1}$$

The term $\frac{Z_{\mathbf{0}}(G')}{Z(G')}$ can be expressed as a product of probabilities:

$$\frac{Z_{\mathbf{0}}(G')}{Z(G')} = \mathbb{P}_{G'}\left(\hat{E} = \mathbf{0}\right) = \prod_{i=1}^{d} \mathbb{P}_{G'}\left(e_i = 0 \,\middle|\, \bigwedge_{j=1}^{i-1} e_j = 0\right) = \prod_{i=1}^{d} \mathbb{P}_{G_i}\left(e_i = 0\right).$$

Plugging this into (1), we obtain

$$\mathbb{P}_G\left(e=0\right) = \frac{1 - (1-\mu_u) \prod_{i=1}^{d} \mathbb{P}_{G_i}\left(e_i = 0\right)}{1 + \lambda_e - (1-\mu_u) \prod_{i=1}^{d} \mathbb{P}_{G_i}\left(e_i = 0\right)}.$$

We remark the if e is the only incident edge to u (i.e. $d = 0$), we have $\mathbb{P}_G\left(e=0\right) = \frac{\mu_u}{\lambda_e + \mu_u}$, which is consistent with the above recursion if we take the convention that an empty product is 1. We also note that every edge e_i in the RHS is a dangling or free edge of G_i.

The above recursion gives a computation tree to compute $\mathbb{P}_G\left(e=0\right)$. We truncate it get our Algorithm 1 to estimate $\mathbb{P}_G\left(e=0\right)$.

3.2 Analysis of Correlation Decay

We recall that $\lambda_e \geq \lambda$ for a constant $\lambda > 0$. We show that Algorithm 1 is a good estimator for $\mathbb{P}_G\left(e=0\right)$. Formally, denote $\mathbb{P}_G^{\ell}\left(e=0\right) \triangleq \texttt{compute}\left(\ell, \mathtt{G}, \mathtt{e}\right)$, we prove

Lemma 5. *For every $\ell \geq 0$,*

$$\left|\mathbb{P}_G^{\ell}\left(e=0\right) - \mathbb{P}_G\left(e=0\right)\right| \leq \alpha \cdot (1+\lambda)^{-\ell/2}.$$

where $\alpha \triangleq \frac{1}{4} \ln\left(1 + \frac{1}{\lambda}\right) \cdot \max\left\{1, \frac{2(1+\lambda)^3}{\lambda^4}\right\}$.

Algorithm 1. Estimating $\mathbb{P}_G(e=0)$

```
*function* compute($\ell$, G, e) :
input  : Recursion depth $\ell$; Graph $G(V,E)$ with edge $e$
output: An estimate of $\mathbb{P}_G(e=0)$
begin
    if $\ell \le 0$ then
        return $\frac{1}{1+\lambda_e}$;
    else if $e$ is a free edge then
        return $\frac{1}{1+\lambda_e}$;
    else if $e$ is a dangling edge then
        $\ell' \leftarrow \ell - \lceil \frac{d+1}{2} \rceil$;
        return $\frac{1-(1-\mu_u)\prod_{i=1}^{d} \mathtt{compute}(\ell', \mathtt{G_i}, \mathtt{e_i})}{1+\lambda_e-(1-\mu_u)\prod_{i=1}^{d} \mathtt{compute}(\ell', \mathtt{G_i}, \mathtt{e_i})}$;
    else // $e$ is a normal edge
        $X \leftarrow$ compute ($\ell$, $\mathtt{G_1}$, $\mathtt{e_1}$);
        $Y \leftarrow$ compute ($\ell$, $\mathtt{G_2}$, $\mathtt{e_2}$);
        return $\frac{XY}{1-X-Y+2XY}$;
```

In order to establish this lemma, we first prove two auxiliary lemmas. Lemma 7 deals with the recursion for dangling edges and Lemma 8 provides a universal bound for marginal probabilities we estimate.

A powerful technique to prove the correlation decay property for a recursion system is to use potential function to amortize the error propagated.

Let $f : D^d \to \mathbb{R}$ be a d-ary function where $D \subseteq \mathbb{R}$ and $\phi : \mathbb{R} \to \mathbb{R}$ is an increasing differentiable continuous function. Denote $\Phi(x) \triangleq \phi'(x)$ and $f^{\phi}(\boldsymbol{x}) \triangleq \phi(f(\phi^{-1}(x_1), \ldots, \phi^{-1}(x_d)))$. The following proposition is a consequence of mean value theorem:

Proposition 6. *For every $\boldsymbol{x} = (x_1, \ldots, x_d), \hat{\boldsymbol{x}} = (\hat{x}_1, \ldots, \hat{x}_d) \in D^d$, it holds that*

1. $|f(\boldsymbol{x}) - f(\hat{\boldsymbol{x}})| = \frac{1}{|\Phi(\tilde{x})|} \cdot |\phi(f(\boldsymbol{x})) - \phi(f(\hat{\boldsymbol{x}}))|$ *for some $\tilde{x} \in D$;*
2. *Assume $x_i = f(\boldsymbol{x_i})$ and $\hat{x}_i = f(\hat{\boldsymbol{x}}_{\boldsymbol{i}})$ for all $1 \le i \le d$, then*

$$|\phi(f(\boldsymbol{x})) - \phi(f(\hat{\boldsymbol{x}}))| \le \left\|\nabla f^{\phi}(\tilde{\boldsymbol{x}})\right\|_1 \cdot \max_{1 \le i \le d} |\phi(f(\boldsymbol{x_i})) - \phi(f(\hat{\boldsymbol{x}}_{\boldsymbol{i}}))|$$

for some $\tilde{\boldsymbol{x}} \in D^d$.

The proof is standard, one can find it in, e.g. [15].

Lemma 7. *Let* $\boldsymbol{x} = (x_1, \ldots, x_d), \hat{\boldsymbol{x}} = (\hat{x}_1, \ldots, \hat{x}_d) \in \left(0, \frac{1}{1+\lambda}\right]^d$ *for some* $\lambda > 0$. *For every* $\hat{\lambda} > 0$ *and* $0 \le \hat{\mu} \le 1$, *define*

$$\begin{aligned} f_{\hat{\lambda},\hat{\mu}}(\boldsymbol{x}) &\triangleq \frac{1-(1-\hat{\mu})\prod_{i=1}^d x_i}{1+\hat{\lambda}-(1-\hat{\mu})\prod_{i=1}^d x_i}; \\ \Phi(x) &\triangleq \frac{1}{x(1-x)}; \\ \phi(x) &\triangleq \int \Phi(x)\,\mathrm{d}x = \ln\left(\frac{x}{1-x}\right); \\ f^{\phi}_{\hat{\lambda},\hat{\mu}}(\boldsymbol{x}) &\triangleq \phi(f_{\hat{\lambda},\hat{\mu}}(\phi^{-1}(x_1), \ldots, \phi^{-1}(x_d))). \end{aligned}$$

Assume $x_i = f_{\lambda_i,\mu_i}(\boldsymbol{z}_i)$, $\hat{x}_i = f_{\lambda_i,\mu_i}(\hat{\boldsymbol{z}}_i)$ *for all* $1 \le i \le d$. *Then*

1. $f_{\hat{\lambda},\hat{\mu}}(\boldsymbol{x}) \le \frac{1}{1+\hat{\lambda}}$.
2. $\left|\phi(f_{\hat{\lambda},\hat{\mu}}(\boldsymbol{x})) - \phi(f_{\hat{\lambda},\hat{\mu}}(\hat{\boldsymbol{x}}))\right| \le (1+\lambda)^{-\frac{d+1}{2}} \max_{1\le i\le d} |\phi(f_{\lambda_i,\mu_i}(\boldsymbol{z}_i)) - \phi(f_{\lambda_i,\mu_i}(\hat{\boldsymbol{z}}_i))|$.

Proof. 1. $f_{\hat{\lambda},\hat{\mu}}(\boldsymbol{x})$ is monotonically decreasing with respect to each x_i, thus $f_{\hat{\lambda},\hat{\mu}}(\boldsymbol{x}) \le f_{\hat{\lambda},\hat{\mu}}(\boldsymbol{0}) \le \frac{1}{1+\hat{\lambda}}$.

2. For every $\boldsymbol{x} \in \left(0, \frac{1}{1+\lambda}\right]^d$, it holds that

$$\begin{aligned} \left\|\nabla f^{\phi}_{\hat{\lambda},\hat{\mu}}(\boldsymbol{x})\right\|_1 &= \Phi(f_{\hat{\lambda},\hat{\mu}}(\boldsymbol{x})) \cdot \sum_{i=1}^d \left| \frac{\frac{\partial f_{\hat{\lambda},\hat{\mu}}(x_1,\ldots,x_d)}{\partial x_i}}{\Phi(x_i)} \right| \\ &= \frac{\left(1+\hat{\lambda}-(1-\hat{\mu})\prod_{i=1}^d x_i\right)^2}{\hat{\lambda}\left(1-(1-\hat{\mu})\prod_{i=1}^d x_i\right)} \cdot \frac{\hat{\lambda}(1-\hat{\mu})\prod_{i=1}^d x_i}{\left(1+\hat{\lambda}-(1-\hat{\mu})\prod_{i=1}^d x_i\right)^2} \cdot \sum_{i=1}^d (1-x_i) \\ &= \frac{(1-\hat{\mu})\prod_{i=1}^d x_i}{1-(1-\hat{\mu})\prod_{i=1}^d x_i}\left(d - \sum_{i=1}^d x_i\right) \\ &\le \frac{\prod_{i=1}^d x_i}{1-\prod_{i=1}^d x_i}\left(d - \sum_{i=1}^d x_i\right). \end{aligned}$$

Let $y \triangleq \left(\prod_{i=1}^d x_i\right)^{1/d}$ and note that $y \le \frac{1}{1+\lambda}$, we have

$$\left\|\nabla f^{\phi}_{\hat{\lambda},\hat{\mu}}(\boldsymbol{x})\right\|_1 \le \frac{dy^d(1-y)}{1-y^d} = \frac{dy^d}{\sum_{i=0}^{d-1} y^i} \le \frac{d}{\sum_{i=1}^d (1+\lambda)^i} \le (1+\lambda)^{-\frac{d+1}{2}}.$$

Then the lemma follows from Proposition 6. □

Lemma 8. *For an arbitrary* $G(V,E)$ *with dangling edge* $e = (u, \_)$ *and* $\ell \ge 0$. *It holds that*

$$\mathbb{P}^{\ell}_G(e=0), \mathbb{P}_G(e=0) \le \frac{1}{1+\lambda_e} \le \frac{1}{1+\lambda}$$

Proof. If e is a free edge, then the lemma naturally holds. Otherwise, the bound follows from the first part of Lemma 7. □

We are now ready to prove Lemma 5.

Proof (of Lemma 5).

– If e is a free edge, then $\left|\mathbb{P}_G^{\ell}(e=0) - \mathbb{P}_G(e=0)\right| = 0$.
– If $e = (u, \_)$ is a dangling edge, recall that $\phi(x) = \ln\left(\frac{x}{1-x}\right)$, we first prove that for every ℓ (may be negative), it holds that

$$\left|\phi\left(\mathbb{P}_G^{\ell}(e=0)\right) - \phi\left(\mathbb{P}_G(e=0)\right)\right| \le \ln\left(1+\frac{1}{\lambda}\right)\cdot(1+\lambda)^{-L/2} \tag{2}$$

where $L \triangleq \max\{\ell, 0\}$.
Denote $\hat{E} \triangleq \{e_1, \ldots, e_d\}$ the set of edges incident to e. If $d = 0$, we have $\mathbb{P}_G^{\ell}(e=0) = \mathbb{P}_G(e=0)$. So we assume $d \ge 1$ and apply induction on L. The base case is that $L = 0$, which means $\ell \le 0$.
Then

$$\begin{aligned}\left|\phi\left(\mathbb{P}_G^{\ell}(e=0)\right) - \phi\left(\mathbb{P}_G(e=0)\right)\right| &= \left|\phi\left(\frac{1}{1+\lambda_e}\right) - \phi\left(\frac{1-(1-\mu_u)\prod_{i=1}^d x_i}{1+\lambda_e-(1-\mu_u)\prod_{i=1}^d x_i}\right)\right| \\ &= \ln\left(\frac{1}{1-(1-\mu_u)\prod_{i=1}^d x_i}\right)\end{aligned}$$

where $x_i \triangleq \mathbb{P}_{G_i}(e_i = 0)$. It follows from Lemma 8 that for every $1 \le i \le d$, $x_i \le \frac{1}{1+\lambda}$, thus

$$\left|\phi\left(\mathbb{P}_G^{\ell}(e=0)\right) - \phi\left(\mathbb{P}_G(e=0)\right)\right| \le -\ln\left(1-\frac{1}{(1+\lambda)^d}\right) \le \ln\left(1+\frac{1}{\lambda}\right).$$

Now assume $L = \ell > 0$ and (2) holds for smaller L. Then the induction hypothesis implies that

$$\varepsilon \triangleq \max_{1\le i\le d}\left|\phi\left(\mathbb{P}_{G_i}^{\ell'}(e_i=0)\right) - \phi\left(\mathbb{P}_{G_i}(e_i=0)\right)\right| \le \ln\left(1+\frac{1}{\lambda}\right)(1+\lambda)^{-L'/2}$$

where $L' = \max\left\{0, \ell - \lceil\frac{d+1}{2}\rceil\right\}$.
Applying Proposition 6, Lemma 7 and Lemma 8, we obtain

$$\begin{aligned}\left|\phi\left(\mathbb{P}_G^{\ell}(e=0)\right) - \phi\left(\mathbb{P}_G(e=0)\right)\right| &\le (1+\lambda)^{-\frac{d+1}{2}}\varepsilon \\ &\le (1+\lambda)^{-\frac{d+1}{2}}\cdot\ln\left(1+\frac{1}{\lambda}\right)\cdot(1+\lambda)^{-L'/2} \\ &\le \ln\left(1+\frac{1}{\lambda}\right)\cdot(1+\lambda)^{-\left(L-\lceil\frac{d+1}{2}\rceil+d+1\right)/2} \\ &\le \ln\left(1+\frac{1}{\lambda}\right)\cdot(1+\lambda)^{-L/2}.\end{aligned}$$

Recall that $\Phi(x) = \frac{1}{x(1-x)} \geq 4$ for $x \in (0,1)$. For all $\ell \geq 0$, Proposition 6 and Lemma 8 together imply that

$$\begin{aligned}\left|\mathbb{P}_G^{\ell}(e=0) - \mathbb{P}_G(e=0)\right| &\leq \frac{1}{4}\ln\left(1+\frac{1}{\lambda}\right)\cdot(1+\lambda)^{-L/2} \\ &= \frac{1}{4}\ln\left(1+\frac{1}{\lambda}\right)\cdot(1+\lambda)^{-\ell/2}.\end{aligned}$$

– If e is a normal edge, the recursion in this case is only applied once and we do not decrease ℓ. Then the algorithm only deals with dangling edges. Consider the recursion we defined in Section 3.1:

$$g(x,y) = \frac{xy}{1-x-y+2xy}.$$

It holds that

$$\|\nabla g\|_1 = \frac{y(1-y)+x(1-x)}{(1-x-y+2xy)^2} \leq \frac{x+y}{(1-x)^2(1-y)^2} \leq \frac{2(1+\lambda)^3}{\lambda^4}$$

whenever $x, y \in \left(0, \frac{1}{1+\lambda}\right]$. Thus we have

$$\begin{aligned}\left|\mathbb{P}_G^{\ell}(e=0) - \mathbb{P}_G(e=0)\right| &\leq \frac{2(1+\lambda)^3}{\lambda^4}\max_{i\in\{1,2\}}\left|\mathbb{P}_{G_i}^{\ell}(e_i=0) - \mathbb{P}_{G_i}(e_i=0)\right| \\ &\leq \frac{(1+\lambda)^3}{2\lambda^4}\ln\left(1+\frac{1}{\lambda}\right)\cdot(1+\lambda)^{-\ell/2}.\end{aligned}$$

□

3.3 Putting All Together

In this section, we prove Lemma 4. It follows from Lemma 5 and Lemma 8 that

(1) $\left|\mathbb{P}_G^{\ell}(e=0) - \mathbb{P}_G(e=0)\right| \leq \alpha\cdot(1+\lambda)^{-\ell/2}$ for some constant α; and
(2) $\mathbb{P}_G^{\ell}(e=0), \mathbb{P}_G(e=0) \leq \frac{1}{1+\lambda} < 1$.

Choosing $\ell = O(\log\frac{1}{\varepsilon})$ is sufficient to ensure

$$\exp(-\varepsilon)\cdot\hat{p} \leq \mathbb{P}_G(e=1) \leq \exp(\varepsilon)\cdot\hat{p}$$

where $\hat{p} = 1 - \mathbb{P}_G^{\ell}(e=0)$.

Now we bound the running time of Algorithm 1. Denote $T(\ell)$ the running time with recursion depth ℓ and denote n the size of the graph. Since we only branch into the case of normal edge once, the following recursion for the case of dangling edge dominates the running time of our algorithm:

$$T(\ell) = d\cdot T(\ell - \Theta(d)) + O(n)$$

where d is the degree of the dangling edge in consideration. Solving the recursion gives $T(\ell) = O(n\exp(\ell))$. Taking $\ell = O(\log\frac{1}{\varepsilon})$ concludes the proof.

References

1. Bayati, M., Gamarnik, D., Katz, D., Nair, C., Tetali, P.: Simple deterministic approximation algorithms for counting matchings. In: Proceedings of STOC, pp. 122–127. ACM (2007)
2. Bezáková, I., Rummler, W.A.: Sampling edge covers in 3-regular graphs. In: Královič, R., Niwiński, D. (eds.) MFCS 2009. LNCS, vol. 5734, pp. 137–148. Springer, Heidelberg (2009)
3. Cai, J.-Y., Lu, P., Xia, M.: Holant problems and counting CSP. In: Proceedings of STOC, pp. 715–724 (2009)
4. Cai, J.-Y., Lu, P., Xia, M.: Computational complexity of Holant problems. SIAM Journal on Computing 40(4), 1101–1132 (2011)
5. Dyer, M., Frieze, A., Jerrum, M.: On counting independent sets in sparse graphs. SIAM Journal on Computing 31(5), 1527–1541 (2002)
6. Galanis, A., Ge, Q., Štefankovič, D., Vigoda, E., Yang, L.: Improved inapproximability results for counting independent sets in the hard-core model. In: Goldberg, L.A., Jansen, K., Ravi, R., Rolim, J.D.P. (eds.) RANDOM 2011 and APPROX 2011. LNCS, vol. 6845, pp. 567–578. Springer, Heidelberg (2011)
7. Jerrum, M., Sinclair, A.: The Markov chain Monte Carlo method: an approach to approximate counting and integration. In: Approximation Algorithms for NP-hard Problems, pp. 482–520 (1996)
8. Li, L., Lu, P., Yin, Y.: Approximate counting via correlation decay in spin systems. In: Proceedings of SODA, pp. 922–940. SIAM (2012)
9. Li, L., Lu, P., Yin, Y.: Correlation decay up to uniqueness in spin systems. In: Proceedings of SODA, pp. 67–84 (2013)
10. Lin, C., Liu, J., Lu, P.: A simple FPTAS for counting edge covers. In: Proceedings of SODA, pp. 341–348 (2014)
11. Liu, J., Lu, P.: FPTAS for counting monotone CNF. arXiv preprint arXiv:1311.3728 (2013)
12. Lu, P., Wang, M., Zhang, C.: FPTAS for weighted Fibonacci gates and its applications. In: Esparza, J., Fraigniaud, P., Husfeldt, T., Koutsoupias, E. (eds.) ICALP 2014. LNCS, vol. 8572, pp. 787–799. Springer, Heidelberg (2014)
13. Restrepo, R., Shin, J., Tetali, P., Vigoda, E., Yang, L.: Improved mixing condition on the grid for counting and sampling independent sets. Probability Theory and Related Fields 156(1-2), 75–99 (2013)
14. Sinclair, A., Srivastava, P., Thurley, M.: Approximation algorithms for two-state anti-ferromagnetic spin systems on bounded degree graphs. In: Proceedings of SODA, pp. 941–953. SIAM (2012)
15. Sinclair, A., Srivastava, P., Yin, Y.: Spatial mixing and approximation algorithms for graphs with bounded connective constant. In: Proceedings of FOCS, pp. 300–309. IEEE (2013)
16. Sly, A.: Computational transition at the uniqueness threshold. In: Proceedings of FOCS, pp. 287–296. IEEE (2010)
17. Sly, A., Sun, N.: The computational hardness of counting in two-spin models on d-regular graphs. In: Proceedings of FOCS, pp. 361–369. IEEE (2012)
18. Valiant, L.G.: Quantum circuits that can be simulated classically in polynomial time. SIAM Journal on Computing 31(4), 1229–1254 (2002)
19. Valiant, L.G.: Holographic algorithms. SIAM Journal on Computing 37(5), 1565–1594 (2008)
20. Weitz, D.: Counting independent sets up to the tree threshold. In: Proceedings of STOC, pp. 140–149. ACM (2006)

Solving Multicut Faster Than 2^n

Daniel Lokshtanov[1,⋆], Saket Saurabh[1,2,⋆⋆], and Ondřej Suchý[3,⋆⋆⋆]

[1] University of Bergen, Norway
daniello@ii.uib.no
[2] Institute of Mathematical Sciences, Chennai, India
saket@imsc.res.in
[3] Faculty of Information Technology, Czech Technical University in Prague
Czech Republic
ondrej.suchy@fit.cvut.cz

Abstract. In the Multicut problem, we are given an undirected graph $G = (V, E)$ and a family $\mathcal{T} = \{(s_i, t_i) \mid s_i, t_i \in V\}$ of pairs of requests and the objective is to find a minimum sized set $S \subseteq V$ such that every connected component of $G \setminus S$ contains at most one of s_i and t_i for any pair $(s_i, t_i) \in \mathcal{T}$. In this paper we give the first non-trivial algorithm for Multicut running in time $\mathcal{O}(1.987^n)$.

1 Introduction

Cuts and flows represent one of the most fundamental fields of studies in network design. Given two distinguished vertices in a graph one can determine the minimum size of a vertex or edge cut separating them in polynomial time using the well known min-cut max-flow duality. However, when one wants to separate more than two terminals from each other, the duality no longer works and the problem of determining the smallest size cut becomes NP-hard for every fixed number of at least three terminals [8].

In this paper we consider a generalization of the above problem, where one is given several pairs of vertices (*requests*) and the task is to determine the minimum size of a set of vertices that separates each pair. More formally, we consider the following problem:

Multicut
Input: An undirected graph $G = (V, E)$ and a family $\mathcal{T} = \{(s_i, t_i) \mid s_i, t_i \in V\}$.
Task: Find a minimum size set $S \subseteq V$ such that every connected component of $G \setminus S$ contains at most one of s_i and t_i for any pair $(s_i, t_i) \in \mathcal{T}$.

Note specifically, that we allow the terminals itself to be deleted, i.e., we consider the unrestricted variant of the problem as named in [2]. However, the version

⋆ Supported by Bergen Research Foundation grant BeHard.
⋆⋆ Supported by ERC Starting Grant PARAPPROX, 306992.
⋆⋆⋆ Work partially supported by the grant 14-13017P of the Czech Science Foundation.

where the terminals are forbidden to delete reduces to the version we investigate (see Observation 1 for more details). If a set $S \subseteq V$ has the requested properties, than we call it a *cut-set* for $(G, \mathcal{T})$. As the roles of s_i and t_i in the pairs of $\mathcal{T}$ are symmetric, we consider the pairs unordered, i.e., if $(s, t) \in \mathcal{T}$, then we also say $(t, s) \in \mathcal{T}$. Moreover, for technical reasons we allow $s = t = v$ for a pair $(s, t) \in \mathcal{T}$ and $v \in V$. In this case, obviously, v must be in any cut-set, as otherwise the component containing v contains both s and t.

MULTICUT generalizes MULTIWAY CUT, where one is given a set of terminals and the task is to separate each two of them. Therefore, MULTICUT is NP-hard already for three requests [8]. Moreover, there is no constant factor approximation for the problem, unless the Unique Games Conjecture fails [3]. Furthermore, the edge variant of MULTICUT is MaxSNP-hard already on stars [12].

Hence, we turn our attention to exact algorithms working in exponential time. Since one can try all subset of vertices, the problem admits a trivial $\mathcal{O}(2^n n^3)$-time algorithm, where n is the number of vertices of the input graph. In this work we break the 2^n barrier. Namely we prove the following theorem.

Theorem 1. MULTICUT *on an n vertex graph can be solved in $\mathcal{O}(1.987^n)$-time.*

Related work. Deleting vertices of the input graph such that the resulting graph satisfies some interesting properties is one of the most well studied directions in exact exponential algorithms. This includes the classical $\mathcal{O}(1.2109^n)$-time algorithm of Robson [18] for MAXIMUM INDEPENDENT SET, an $\mathcal{O}(1.7356^n)$-time algorithm for FEEDBACK VERTEX SET [19], or an $\mathcal{O}(1.4689^n)$-time algorithm for DOMINATING SET [13], to name at least a few of them. For MULTIWAY CUT, such an algorithm was presented by Fomin et al. [11] achieving $\mathcal{O}(1.8638^n)$-time. This was recently improved to $\mathcal{O}(1.4766^n)$-time by Chitnis et al. [5]. For MULTICUT, however, no algorithm faster than the trivial $\mathcal{O}(2^n \cdot n^{\mathcal{O}(1)})$-time is known.

Concerning approximation algorithms, Garg et al. [12] show that MULTICUT can be approximated within $\mathcal{O}(\log k)$ factor, where $k = |\mathcal{T}|$. Cut problems are also well studied from the perspective of parameterized algorithms. Marx [14] was the first to consider cut problems in the context of parameterized complexity. He gave an algorithm for parameterized MULTIWAY CUT with running time $\mathcal{O}(4^{k^3} n^{\mathcal{O}(1)})$ with the current fastest algorithm running in time $\mathcal{O}(2^k n^{\mathcal{O}(1)})$ [7]. The notions used in this paper has been useful in settling parameterized complexity of variety of problems including DIRECTED FEEDBACK VERTEX SET [4], ALMOST 2 SAT [17] and ABOVE GUARANTEE VERTEX COVER [16,17]. Recently, Marx and Razgon [15] and Bousquet, Daligault, and Thomassé [1] independently showed that MULTICUT, finding k vertices to disconnect given pairs of terminals is FPT. Continuing this line of study, Chitnis, Hajiaghayi and Marx studied MULTIWAY CUT on directed graphs and showed it to be FPT [6].

2 Preliminaries

Our notation for graph theoretic notions is standard. We summarize some of the frequently used concepts here. For a finite set V, a pair $G = (V, E)$ such that

$E \subseteq V^2$ is a graph on V. The elements of V are called *vertices*, while pairs of vertices $\{u,v\}$ such that $\{u,v\} \in E$ are called *edges*. A graph $G' = (V', E')$ is a subgraph of $G = (V,E)$ if $V' \subseteq V$ and $E' \subseteq E \cap V'^2$. If E' contains all the edges $\{u,v\} \in E$ with $u,v \in V'$, then G' is an *induced subgraph* of G, *induced by* V', denoted by $G[V']$. For any $U \subseteq V$, $G \setminus U = G[V \setminus U]$. For $v \in V$, $N_G(v) = \{u \mid \{u,v\} \in E\}$. A set of vertices C of V is said to be *clique* if there is an edge for every pair of vertices in C.

3 Basic Observations

Our algorithm applies the following two operations on vertices of the graph:

Definition 1. *1. By* deleting *a vertex v from the instance $(G, \mathcal{T})$ we mean removing the vertex v from the vertex set of G together with all its incident edges as well as removing all pairs containing v from $\mathcal{T}$. I.e., we continue with the instance $(G', \mathcal{T}')$, where $G' = G \setminus \{v\}$ and $\mathcal{T}' = \mathcal{T} \setminus \{(u,v) \mid u \in V\}$.*

2. By contracting *a vertex v in the instance $(G, \mathcal{T})$ we mean first turning the neighborhood of v into a clique and adding into $\mathcal{T}$ the pair (u,w), whenever (v,w) was in $\mathcal{T}$ and u is a neighbor of v in G. Finally, we remove the vertex v from V together with all its incident edges from E and all pairs containing v from $\mathcal{T}$. I.e., we continue with the instance $(G', \mathcal{T}')$, where $G' = (V \setminus \{v\}, E')$, $E' = E \cup \{\{u,w\} \mid u,w \in N_G(v)\} \setminus \{\{v,u\} \mid u \in V\}$, and $\mathcal{T}' = \mathcal{T} \cup \{(u,w) \mid u \in N_G(v) \wedge (v,w) \in \mathcal{T}\} \setminus \{(v,w) \mid w \in V\}$.*

The following two lemmata show, that the two operations correspond to taking and not taking the vertex into the constructed solution, respectively.

Lemma 1. *If $(G', \mathcal{T}')$ is obtained from $(G, \mathcal{T})$ by deleting a vertex v, then $S \subseteq V$ containing v is a cut-set for $(G, \mathcal{T})$ if and only if $S \setminus \{v\}$ is a cut-set for $(G', \mathcal{T}')$.*

Proof. First, if S is a cut-set for $(G, \mathcal{T})$, then $S \setminus \{v\}$ is a cut-set for $(G', \mathcal{T}')$, since we have $G' \setminus (S \setminus \{v\}) = G \setminus S$ and $\mathcal{T}' \subseteq \mathcal{T}$.

Let us now assume, that $S \setminus \{v\}$ is a cut-set for $(G', \mathcal{T}')$ and $v \in S$. Then again $G \setminus S = G' \setminus (S \setminus \{v\})$, each connected component of $G \setminus S$ contains at most one vertex of each pair in $\mathcal{T}'$ and, since all pairs in $\mathcal{T} \setminus \mathcal{T}'$ contain v, also of each pair in $\mathcal{T}$. □

Lemma 2. *If $(G', \mathcal{T}')$ is obtained from $(G, \mathcal{T})$ by contracting a vertex v and $(v,v) \notin \mathcal{T}$, then $S \subseteq V \setminus \{v\}$ is a cut-set for $(G, \mathcal{T})$ if and only if S is a cut-set for $(G', \mathcal{T}')$.*

Proof. Suppose first that $S \subseteq V \setminus \{v\}$ is not a cut-set for $(G, \mathcal{T})$. Hence there is a pair $(x,y) \in \mathcal{T}$ such that there is an x-y-path in $G \setminus S$. If this path does not contain v then it is also present in $G' \setminus S$, $(x,y) \in \mathcal{T}'$, and S is not a cut-set for $(G', \mathcal{T}')$. If $v \notin \{x,y\}$ but the x-y-path contains v, then we may omit it from the path to obtain a path in $G' \setminus S$, as all the neighbors of v are connected by edges in G'. Thus again S is not a cut-set for $(G', \mathcal{T}')$. Finally, if $v \in \{x,y\}$, we may assume without loss of generality, that $v = x$ and $v \neq y$ as $(v,v) \notin \mathcal{T}$. Let u be the neighbor of v on the path. The graph $G' \setminus S$ contains the u-y-path and $(u,y) \in \mathcal{T}'$ by the construction of $\mathcal{T}'$. Hence also in this case S is not a cut-set for $(G', \mathcal{T}')$.

Now suppose that S is not a cut-set for $(G', \mathcal{T}')$. Therefore there is a pair $(x,y) \in \mathcal{T}'$ such that there is an x-y-path in $G' \setminus S$. If $(x,y) \in \mathcal{T}$ and the path contains at most one neighbor of v, then this path is also contained in $G \setminus S$. If the path contains at least two neighbors of v, then we may go from the first neighbor of v on the path to v and from v to the last neighbor of v on the path, obtaining an x-y-path in $G \setminus S$. If $(x,y) \notin \mathcal{T}$, then assume without loss of generality that $(v,y) \in \mathcal{T}$ and x is a neighbor of v. Furthermore, as all neighbors of v have a request to y in $\mathcal{T}'$ in this case, we may assume that the path does not contain any further neighbor of v, except for x. Now we can prolong the path by adding v to the beginning to obtain a v-y-path in $G \setminus S$. Summing up, S is not a cut-set for $(G, \mathcal{T})$, finishing the proof. □

As we have said, if $(v,v) \in \mathcal{T}$ for some $v \in V$, then v is in any cut-set. Hence we apply the following reduction rule as often as possible.

Reduction Rule 1. If $(v,v) \in \mathcal{T}$, then delete v from $(G, \mathcal{T})$.

Observe now, that one can obtain an $\mathcal{O}(2^n \cdot n^{\mathcal{O}(1)})$-time algorithm for MULTICUT by branching for each vertex into two branches — either the vertex is deleted or contracted. Lemmata 1 and 2 give recipe how the minimum cut-sets returned by the recursive calls need to be modified to obtain the minimum cut-set for the instance (see Function Cut).

Function Cut$(G, \mathcal{T})$

```
begin
    if $\mathcal{T} = \emptyset$ then
        return $\emptyset$;
    Let $v$ be an arbitrary vertex of $G$;
    $(G', \mathcal{T}') \leftarrow Delete(v, (G, \mathcal{T}))$;
    $S_1 \leftarrow \{v\} \cup Cut(G', \mathcal{T}')$;
    if $(v,v) \in \mathcal{T}$ then
        return $S_1$;
    $(G', \mathcal{T}') \leftarrow Contract(v, (G, \mathcal{T}))$;
    $S_2 \leftarrow Cut(G', \mathcal{T}')$;
    if $|S_1| \leq |S_2|$ then
        return $S_1$;
    else
        return $S_2$;
end
```

As our algorithm is based on the same operations, we will no longer describe how the minimum cut-set is actually obtained, for brevity. The speed up of our algorithm is achieved by carefully choosing the vertices to branch on and omitting the branches that cannot lead to a (minimum) cut-set.

To conclude this section, we show that the variant of Multicut, where the terminals are forbidden to delete can be reduced to Multicut.

Observation 1. Multicut with Undeletable Terminals *can be reduced in polynomial time to* Multicut *with at most the same number of vertices.*

Proof. It is enough to contract all terminals. By Lemma 2, a subset of vertices is a cut-set in the original instance not containing the terminals, if and only if it is a cut set in the resulting instance. The contraction can be clearly carried out in polynomial time and does not increase the number of vertices. □

4 Our Algorithm

To guide the branching, our algorithm maintains a clique C, which we call the *active clique*. Hence, in the recursive calls, we give $G, \mathcal{T}$, and C as arguments. We should explain how our two operations affect the active clique. If $v \notin C$, then C stays untouched after the operations. If $v \in C$ and we delete v, then we let $C := C \setminus \{v\}$ whereas if we contract v we let $C := C \setminus \{v\} \cup N(v)$. Note that in the last case the new C is indeed a clique, as originally C must have been a subset of $N[v]$.

The bigger C is, the closer together are the vertices of the graph, which is beneficial for the algorithm. Hence, if the graph has a big active clique, we consider it little smaller. More precisely, we use the Measure and Conquer approach [9] and our measure μ for the size of the instance $(G, \mathcal{T}, C)$ is given by the following formula:

$$\mu = |V| - \alpha|C|,$$

where $0 < \alpha < 0.1$.

Along with the Reduction Rule 1 we apply also the following two reduction rules:

Reduction Rule 2. If vertex v is isolated in G and Reduction Rule 1 does not apply, then contract v in $(G, \mathcal{T})$.

Note that vertex v does not influence whether a set is a cut-set, since it always forms a component for itself and $(v, v) \notin \mathcal{T}$, justifying the correctness of the rule.

Reduction Rule 3. If $C = \emptyset$, then pick any vertex $v \in V$ and let $C = \{v\}$.

Note that each of the reduction rules decreases the value of μ.

We now describe the branching rules. We apply them in the given order, that is, a latter branching rule is only applied if none of the earlier ones applies. Moreover, we apply the reduction rules exhaustively before applying any of the branching rules. We argue the correctness of the rules, but postpone the discussion of the running time of the whole algorithm.

Rule 1. If there is $(x, y) \in \mathcal{T}$ such that the shortest path P between x and y in G is of length at most 3, then denote $V(P) = \{v_1, \dots, v_t\}$ for some $t \in \{2, \dots 4\}$ in such a way that $V(P) \cap C = \{v_{s+1}, \dots, v_t\}$ for some $s \in \{t-2, t-1, t\}$. Branch into following ways:

- v_1 is deleted;
- v_1 is contracted, v_2 is deleted;
- v_1, v_2 are contracted, v_3 is deleted;

  $\vdots$
- $v_1, \dots, v_{t-1}$ are contracted, v_t is deleted.

In the case described in Rule 1 at least one of the vertices $v_1, \dots, v_t$ must be deleted and the rule explores all such options. It follows that the rule is correct.

Rule 2. Let $N(C) = \bigcup_{c \in C} N(c) \setminus C$. If $|N(C)| \leq \frac{2}{3}|C|$ then for every $S_0 \subseteq (C \cup N(C))$ with $|S_0| \leq |N(C)|$ branch in the following way: delete all vertices in S_0 and contract all vertices in $(C \cup N(C)) \setminus S_0$.

The correctness of the rule follows from the following lemma:

Lemma 3. *If the situation is as in Rule 2, then there is a minimal cut-set S such that $|S \cap (C \cup N(C))| \leq |N(C)|$.*

Proof. Let S be a minimal cut-set such that $|S \cap (C \cup N(C))| > |N(C)|$. We claim that $S' = (S \setminus C) \cup N(C)$ is also a cut-set, contradicting the minimality of S as $|S'| < |S|$. Suppose S' is not a cut-set. Then there is a pair $(x, y) \in \mathcal{T}$ such that there is an x-y-path P in $G \setminus S'$. If $V(P) \cap (C \cup N(C)) = \emptyset$, then P is also present in $G \setminus S$ contradicting S being a cut-set. If $V(P) \cap (C \cup N(C)) \neq \emptyset$, then either $V(P) \cap N(C) \neq \emptyset$ or $V(P) \subseteq C$. The former case cannot appear as $V(P) \subseteq (V(G) \setminus S')$ and S' contains $N(C)$ and the later case implies that P is of length at most 1. However, in this case Rule 1 would apply, a contradiction. □

The correctness of Rule 2 now follows from Lemmata 1 and 2 as we exhaustively try all possible intersections of the minimal cut-set with $C \cup N(C)$.

Rule 3. If there is a vertex $v \in C$ such that $|N(v) \setminus C| \geq 3$, then branch into the following canonical ways:

- delete v;
- contract v.

As the branching explores the two canonical options, the correctness is clear. The following explains the significance of the coming rules.

Lemma 4. *If Rules 2 and 3 do not apply, then there is a vertex v in $N(C)$ with $|N(v) \cap C| \leq 2$.*

Proof. Suppose there is no such vertex and let us count the number z of edges between C and $N(C)$ in G. Since Rule 3 does not apply, we know that $z \leq 2|C|$. On the other hand, we know that $z \geq 3|N(C)|$ as otherwise there is a vertex in $N(C)$ incident to at most two edges with the other endpoint in C. As Rule 2 does not apply, we have $3|N(C)| > 3 \cdot \frac{2}{3}|C| = 2|C|$. Hence $2|C| < 3|N(C)| \leq z \leq 2|C|$— a contradiction. □

Rule 4. If v is a vertex in $N(C)$ with $|N(v) \cap C| \leq 2$ and $|N(v) \setminus C| \leq 3$, then denote $N(v) = \{u_1, \ldots, u_t\}$ such that $N(v) \cap C = \{u_{s+1}, \ldots, u_t\}$ for some $t \in \{1, \ldots, 5\}$ and $s \in \{t-2, t-1\}$. Branch into the following ways:

- u_1 is contracted;
- u_1 is deleted, u_2 is contracted;

  $\vdots$
- $u_1, \ldots, u_{t-1}$ are deleted, u_t is contracted;
- $u_1, \ldots, u_t$ are deleted, v is contracted.

To see the correctness of this rule, observe that the first t branches correspond to one of the neighbors of v not being part of the cut-set constructed, while the last one corresponds to the whole neighborhood of v being part of the cut-set constructed. In this sense the branching is exhaustive. In the last branch the vertex v is contracted by Reduction Rule 2.

Rule 5. If v is a vertex in $N(C)$ with $|N(v) \cap C| \leq 2$ and $|N(v) \setminus C| \geq 4$, then let $u \in C \cap N(v)$ and branch into the following ways:

- v is deleted;
- v is contracted, u is deleted;
- v and u are contracted.

Since the rule only applies the canonical branching to v and then to u in one of the branches, the correctness is clear.

5 Time Complexity

In this section we analyze the time complexity of our algorithm. As the algorithm is recursive, we first bound the number of recursive calls and then the time spent per each call.

Let us first bound the number of terminal calls $T(\mu)$ produced, when the algorithm is executed on an instance with measure at most μ. Recall that $\mu = |V| - \alpha|C|$ and, since $\alpha < 0.1$, we have $0.9|V| < \mu \leq |V|$. We claim that $T(\mu) \leq \lambda^{\mu}$ for $\lambda = 1.9865$, $\alpha = 0.032$, and all values of $\mu \geq 0$. We prove the claim by induction on μ.

If $\mu \leq 0$, then the graph is empty and the instance can be resolved by outputting $\emptyset$, giving one terminal call. For $0 < \mu \leq 1$, the graph contains at most one vertex and Reduction Rule 1 or 2 applies, reducing to previous case and giving one terminal call. This gives the base of the induction.

Now suppose we are facing an instance of measure μ and the claim holds for instances with measure μ' where $\mu' < \mu$. Note, that the measure is decreased by one for each vertex not in C deleted or contracted, by at least $1 - \alpha$ for each vertex in C contracted or deleted, and by α for each vertex newly put in C (e.g., due to contraction of its neighbor).

If any of the reduction rules applies to the instance, then the measure gets decreased without increasing the number of terminal calls and the claim follows. Now let us distinguish, which of the branching rules applies.

If Rule 1 applies, then in the branch i (the one where v_i is deleted) the measure is reduced by at least i if $i \leq s$ and by at least $s+(1-\alpha)(i-s) = i-\alpha(i-s)$ if $i > s$. It follows that $T(\mu) \leq \sum_{i=1}^{s} T(\mu-i)+\sum_{i=s+1}^{t} T(\mu-i+\alpha(i-s))$. Hence to prove the claim it is enough to prove that $\lambda^{\mu} \geq \sum_{i=1}^{s} \lambda^{\mu-i}+\sum_{i=s+1}^{t} \lambda^{\mu-i+\alpha(i-s)}$ which is equivalent to $1 \geq \sum_{i=1}^{s} \lambda^{-i} + \sum_{i=s+1}^{t} \lambda^{-i+\alpha(i-s)}$. Observe that decreasing s increases the right hand side, as $\lambda > 1$ and $\alpha > 0$. Hence, it suffices to prove the inequality for $s = t-2$.

Distinguishing the value of t the claim follows from that

- $1 \geq \lambda^{-1+\alpha} + \lambda^{-2+2\alpha} \doteq 0.778$ ($t = 2$),
- $1 \geq \lambda^{-1} + \lambda^{-2+\alpha} + \lambda^{-3+2\alpha} \doteq 0.896$ ($t = 3$), and
- $1 \geq \lambda^{-1} + \lambda^{-2} + \lambda^{-3+\alpha} + \lambda^{-4+2\alpha} \doteq 0.954$ ($t = 4$), for $\lambda = 1.9865$ and $\alpha = 0.032$.

If Rule 2 applies, let us denote $a = |C|$, $b = |N(C)|$, $m = a+b$, and $\beta = \frac{b}{m}$. The measure drops in each case by at least $b + a(1-\alpha) = m(1-\alpha) + b\alpha$. Hence $T(\mu) \leq \sum_{c=0}^{b} \binom{m}{c} T(\mu - m(1-\alpha) - b\alpha)$. To prove the claim we need to show that $1 \geq \sum_{c=0}^{b} \binom{m}{c} \lambda^{-m(1-\alpha)-b\alpha}$ for any $b \leq \frac{2}{3}a$. By [10, Lemma 3.13] we have that $\sum_{c=0}^{b} \binom{m}{c} \lambda^{-m(1-\alpha)-b\alpha} \leq \lambda^{-m(1-\alpha)-b\alpha} \cdot (\frac{1}{\beta})^{\beta m} (\frac{1}{1-\beta})^{(1-\beta)m} = \left(\frac{1}{\lambda^{1-\alpha}} (\frac{1}{\beta\lambda^{\alpha}})^{\beta} (\frac{1}{1-\beta})^{1-\beta}\right)^m$ and it remains to prove that $f(\beta) = \frac{1}{\lambda^{1-\alpha}} \cdot (\frac{1}{\beta\lambda^{\alpha}})^{\beta} \cdot (\frac{1}{1-\beta})^{1-\beta} \leq 1$ for every $0 \leq \beta \leq \frac{\frac{2}{3}}{1+\frac{2}{3}} = \frac{2}{5}$. We first show that this function is nondecreasing on the interval $(0, \frac{2}{5}]$. To this end, consider the function $g(\beta) = \ln f(\beta) = -(1-\alpha)\ln\lambda - \beta(\ln\beta + \alpha\ln\lambda) - (1-\beta)\ln(1-\beta)$. Function g is well defined on the interval and if g is nondecreasing, then so is f. For the derivative we have $g'(\beta) = -(\ln\beta + \alpha\ln\lambda) - \frac{\beta}{\beta} + \ln(1-\beta) + \frac{1-\beta}{1-\beta} = \ln\frac{1-\beta}{\beta\lambda^{\alpha}}$. Thus, $g(\beta)$ is nondecreasing as long as $\frac{1-\beta}{\beta} \geq \lambda^{\alpha}$. But since $\frac{3}{2} > \lambda^{\alpha} \doteq 1.02$, function g and, hence, also f is non-decreasing for all $\beta \in (0, \frac{2}{5}]$. Therefore, it is enough to notice that $1 \geq f(\frac{2}{5}) = \frac{1}{\lambda^{1-\alpha}}(\frac{5}{2})^{\frac{2}{5}}(\frac{5}{3})^{\frac{3}{5}} \doteq 0.99982$ and $1 \geq f(0) = \frac{1}{\lambda^{1-\alpha}} \doteq 0.51$ for $\lambda = 1.9865$ and $\alpha = 0.032$.

If Rule 3 applies, then the measure gets reduced by 1 and at least $1+2\alpha$, respectively, as in the latter case v is removed from C, while its at least 3 neighbors become part of C. Hence, we have $T(\mu) \leq T(\mu-1) + T(\mu-1-2\alpha)$. To prove the claim it is enough to observe that $1 \geq \lambda^{-1} + \lambda^{-1-2\alpha} \doteq 0.985$.

If Rule 4 applies, then in the branch i (the one where u_i is contracted) the measure is reduced by at least i if $i \leq s$ and by at least $s + (1-\alpha)(i-s) + \alpha = i - \alpha(i-s-1)$ if $t \geq i > s$, as in this case v becomes part of C, whereas in the last branch it is reduced by $s+(1-\alpha)(t-s)+1 = t-\alpha(t-s)+1$. It follows that $T(\mu) \leq \sum_{i=1}^{s} T(\mu-i) + \sum_{i=s+1}^{t} T(\mu-i+\alpha(i-s-1)) + T(\mu-t+\alpha(t-s)+1)$. Hence to prove the claim it is enough to prove that $1 \geq \sum_{i=1}^{s} \lambda^{-i} + \sum_{i=s+1}^{t} \lambda^{-i+\alpha(i-s-1)} + \lambda^{-t+\alpha(t-s)-1}$. Observe that decreasing s increases the right hand side, as $\lambda > 1$ and $\alpha > 0$. Hence, it suffices to prove the inequality for $s = t-2$.

Distinguishing the value of t we have that

- $1 \geq \lambda^{-1} + \lambda^{-2+\alpha} \doteq 0.762$ ($t = 1$),
- $1 \geq \lambda^{-1} + \lambda^{-2+\alpha} + \lambda^{-3+2\alpha} \doteq 0.896$ ($t = 2$),
- $1 \geq \lambda^{-1} + \lambda^{-2} + \lambda^{-3+\alpha} + \lambda^{-4+2\alpha} \doteq 0.954$ ($t = 3$),
- $1 \geq \lambda^{-1} + \lambda^{-2} + \lambda^{-3} + \lambda^{-4+\alpha} + \lambda^{-5+2\alpha} \doteq 0.984$ ($t = 4$),
- $1 \geq \lambda^{-1}+\lambda^{-2}+\lambda^{-3}+\lambda^{-4}+\lambda^{-5+\alpha}+\lambda^{-6+2\alpha} \doteq 0.9986$ ($t = 5$), for $\lambda = 1.9865$ and $\alpha = 0.032$.

Finally, if Rule 5 applies, then the measure is decreased by 1, by $2-\alpha$, and by at least $2 + 3\alpha$, respectively, as in the last branch the neighbors of v outside C become part of C, but u is removed from C. Therefore, we have $T(\mu) \leq T(\mu - 1) + T(\mu - 2 + \alpha) + T(\mu - 2 - 4\alpha)$. To prove the claim it is enough to observe that $1 \geq \lambda^{-1} + \lambda^{-2+\alpha} + \lambda^{-2-3\alpha} \doteq 0.99968$.

Now to compute the total number of recursive calls, observe that each branching rule reduces the number of vertices in the graph in each branch, and, hence, the total number of recursive calls is at most $n \cdot \lambda^{\mu}$. In each recursive call we first apply the reduction rules exhaustively and then check which of the branching rules applies. Reduction Rule 1 can be applied to each relevant vertex in $\mathcal{O}(n)$, without creating new opportunities to apply it. Therefore, it can be applied exhaustively in $\mathcal{O}(n^2)$ time. Similarly, Reduction Rule 2 can be applied to each vertex in $\mathcal{O}(n)$ time and this does not create any new opportunities to apply this rule or the previous one. Reduction Rule 3 can be always applied in constant time.

To apply Rule 1 we compute the distance between every pair of vertices in $\mathcal{O}(n^3)$ time and then check for each pair $(u, v) \in \mathcal{T}$ their distance. To delete a vertex from a graph takes $\mathcal{O}(n)$ time whereas contracting a vertex takes $\mathcal{O}(n^2)$. Thus, the whole preparation of the graph for each branch takes $\mathcal{O}(n^2)$ time in this case.

The size of the neighborhood $N(C)$ can be computed in $\mathcal{O}(n^2)$ time. It follows from the above ideas that a graph can be prepared for each branch in $\mathcal{O}(n^3)$ time. Therefore, Rule 2 can be applied in $\mathcal{O}(n^3)$ time, accounting the time for the preparation of the graph to the call executed.

Rules 3–5 can be applied in $\mathcal{O}(n^2)$ time, since we only have to compute for each vertex the number of its neighbors in C and outside C and then prepare graphs for constant number of branches, each deleting or contracting only constant number of vertices.

Altogether we spend only $\mathcal{O}(n^3)$ time per a recursive call. Since μ is always at most n we obtain $\mathcal{O}(1.9865^n \cdot n^4) = \mathcal{O}(1.987^n)$ running time for the whole algorithm. We only need a polynomial space. This completes the proof of our theorem.

6 Conclusions

In this paper we gave an algorithm for Multicut running in time $\mathcal{O}(1.987^n)$, the first algorithm breaking the barrier of 2^n. One can obtain an algorithm for edge variant of MULTICUT running in time $2^n n^{\mathcal{O}(1)}$. It is an interesting problem to obtain an algorithm for EDGE MULTICUT running in time $(2-\epsilon)^n n^{\mathcal{O}(1)}$ for some fixed $\epsilon > 0$. Finally, it would also be interesting to obtain an algorithm for MULTICUT in directed graphs running in time $(2-\epsilon)^n n^{\mathcal{O}(1)}$ for some fixed $\epsilon > 0$.

References

1. Bousquet, N., Daligault, J., Thomassé, S.: Multicut is FPT. In: Proceedings of the Forty-third Annual ACM Symposium on Theory of Computing, STOC 2011, pp. 459–468. ACM, New York (2011), `http://doi.acm.org/10.1145/1993636.1993698`
2. Călinescu, G., Fernandes, C.G., Reed, B.: Multicuts in unweighted graphs and digraphs with bounded degree and bounded tree-width. Journal of Algorithms 48(2), 333–359 (2003), `http://www.sciencedirect.com/science/article/pii/S0196677403000737`
3. Chawla, S., Krauthgamer, R., Kumar, R., Rabani, Y., Sivakumar, D.: On the hardness of approximating multicut and sparsest-cut. Computational Complexity 15(2), 94–114 (2006), `http://dx.doi.org/10.1007/s00037-006-0210-9`
4. Chen, J., Liu, Y., Lu, S., O'Sullivan, B., Razgon, I.: A fixed-parameter algorithm for the directed feedback vertex set problem. J. ACM 55(5) (2008)
5. Chitnis, R., Fomin, F., Lokshtanov, D., Misra, P., Ramanujan, M., Saurabh, S.: Faster exact algorithms for some terminal set problems. In: Gutin, G., Szeider, S. (eds.) IPEC 2013. LNCS, vol. 8246, pp. 150–162. Springer, Heidelberg (2013), `http://dx.doi.org/10.1007/978-3-319-03898-8_14`
6. Chitnis, R.H., Hajiaghayi, M., Marx, D.: Fixed-parameter tractability of directed multiway cut parameterized by the size of the cutset. In: Rabani, Y. (ed.) SODA, pp. 1713–1725. SIAM (2012)
7. Cygan, M., Pilipczuk, M., Pilipczuk, M., Wojtaszczyk, J.O.: On multiway cut parameterized above lower bounds. In: Marx, D., Rossmanith, P. (eds.) IPEC 2011. LNCS, vol. 7112, pp. 1–12. Springer, Heidelberg (2012)
8. Dahlhaus, E., Johnson, D.S., Papadimitriou, C.H., Seymour, P.D., Yannakakis, M.: The complexity of multiterminal cuts. SIAM J. Comput. 23(4), 864–894 (1994)
9. Fomin, F.V., Grandoni, F., Kratsch, D.: A measure & conquer approach for the analysis of exact algorithms. J. ACM 56(5), 25:1–25:32 (2009), `http://doi.acm.org/10.1145/1552285.1552286`
10. Fomin, F.V., Kratsch, D.: Exact Exponential Algorithms. Texts in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg (2010)
11. Fomin, F., Heggernes, P., Kratsch, D., Papadopoulos, C., Villanger, Y.: Enumerating minimal subset feedback vertex sets. Algorithmica 69(1), 216–231 (2014), `http://dx.doi.org/10.1007/s00453-012-9731-6`
12. Garg, N., Vazirani, V., Yannakakis, M.: Primal-dual approximation algorithms for integral flow and multicut in trees. Algorithmica 18(1), 3–20 (1997), `http://dx.doi.org/10.1007/BF02523685`
13. Iwata, Y.: A faster algorithm for dominating set analyzed by the potential method. In: Marx, D., Rossmanith, P. (eds.) IPEC 2011. LNCS, vol. 7112, pp. 41–54. Springer, Heidelberg (2012), `http://dx.doi.org/10.1007/978-3-642-28050-4_4`
14. Marx, D.: Parameterized graph separation problems. Theoret. Comput. Sci. 351(3), 394–406 (2006)

15. Marx, D., Razgon, I.: Fixed-parameter tractability of multicut parameterized by the size of the cutset. In: STOC, pp. 469–478 (2011)
16. Raman, V., Ramanujan, M.S., Saurabh, S.: Paths, flowers and vertex cover. In: Demetrescu, C., Halldórsson, M.M. (eds.) ESA 2011. LNCS, vol. 6942, pp. 382–393. Springer, Heidelberg (2011)
17. Razgon, I., O'Sullivan, B.: Almost 2-sat is fixed-parameter tractable. J. Comput. Syst. Sci. 75(8), 435–450 (2009)
18. Robson, J.: Algorithms for maximum independent sets. Journal of Algorithms 7(3), 425–440 (1986), `http://www.sciencedirect.com/science/article/pii/0196677486900325`
19. Xiao, M., Nagamochi, H.: An improved exact algorithm for undirected feedback vertex set. In: Widmayer, P., Xu, Y., Zhu, B. (eds.) COCOA 2013. LNCS, vol. 8287, pp. 153–164. Springer, Heidelberg (2013), `http://dx.doi.org/10.1007/978-3-319-03780-6_14`

Tight Bounds for Active Self-assembly Using an Insertion Primitive*

Caleb Malchik and Andrew Winslow

Tufts University, Medford, MA 02155, USA
`caleb.malchik@tufts.edu`, `awinslow@cs.tufts.edu`

Abstract. We prove two tight bounds on the behavior of a model of self-assembling particles introduced by Dabby and Chen (SODA 2012), called *insertion systems*, where monomers insert themselves into the middle of a growing linear polymer. First, we prove that the expressive power of these systems is equal to context-free grammars, answering a question posed by Dabby and Chen. Second, we prove that polymers of length $2^{\Theta(k^{3/2})}$ can be deterministically constructed by insertion systems of k monomer types in $O((\log n)^{5/3})$ expected time, and that this is the best possible in both the number of types and expected time.

Keywords: DNA computing, formal languages, polymers, context-free grammars.

1 Introduction

In this work we study a theoretical model of *algorithmic self-assembly*, in which simple particles aggregate in a distributed manner to carry out complex functionality. Perhaps the the most well-studied theoretical model of algorithmic self-assembly is the *abstract Tile Assembly Model (aTAM)* of Winfree [15] consisting of square *tiles* irreversibly attach to a growing polyomino-shaped assembly according to matching edge colors. This model is capable of Turing-universal computation [15], self-simulation [5], and efficient assembly of general (scaled) shapes [14] and squares [1,13]. Despite this power, the model is incapable of assembling some shapes efficiently; a single row of n tiles requires n distinct tile types and $\Omega(n \log n)$ expected assembly time [2] and any shape with n tiles requires $\Omega(\sqrt{n})$ expected time to assemble [7].

Such a limitation may not seem so significant, except that a wide range of biological systems form complex assemblies in time polylogarithmic in the assembly site, as Dabby and Chen [4] and Woods et al. [16] observe. These biological systems are capable of such growth because their particles (e.g. living cells) *actively* carry out geometric reconfiguration. In the interest of both understanding naturally occurring biological systems and creating synthetic systems with additional capabilities, several models of *active self-assembly* have been proposed recently. These include the graph grammars of Klavins et al. [9,10], the *nubots* model

* A full version of this paper can be found at http://arxiv.org/abs/1401.0359

of Woods et al. [3,16], and the insertion systems of Dabby and Chen [4]. Both graph grammars and nubots are capable of a topologically rich set of assemblies and reconfigurations, but rely on stateful particles forming complex bond arrangments. In contrast, insertion systems consist of stateless particles forming a single chain of bonds. Indeed, all insertion systems are captured as a special case of nubots in which a linear polymer is assembled via parallel insertion-like reconfigurations, as in Theorem 5.1 of [17]. The simplicity of insertion systems makes their implementation in matter a more immediately attainable goal; Dabby and Chen [4] describe a direct implementation of these systems in DNA.

We are careful to make a distinction between *active self-assembly*, where assemblies undergo reconfiguration, and *active tile self-assembly* [6,8,11,12], where tile-based assemblies change their bond structure. Active self-assembly enables exponential assembly rates by enabling insertion of new particles throughout the assembly, while active tile self-assembly does not: assemblies formed consist of rigid tiles and the $\Omega(\sqrt{n})$ expected-time lower bound of Keenan, Schweller, Sherman, and Zhong [7] still applies.

1.1 Our Results

We prove two tight bounds on the behavior of insertion systems. First, we consider what languages can be *expressed* by insertion systems, i.e. correspond to a set of polymers constructed by some insertion system. Dabby and Chen prove that only context-free languages are expressible by insertion systems, and ask whether every context-free language is indeed expressed by some insertion system. We answer this question in the affirmative, and as a consequence prove that the languages expressible by insertion systems are exactly the context-free languages.

Second, we consider constructing the largest finite polymers as fast as possible. Dabby and Chen prove that insertion systems with k monomer types can deterministically construct polymers of length $n = 2^{\Theta(\sqrt{k})}$ in $O(\log^3 n)$ expected time. We improve on both the polymer length *and* expected time by describing systems that deterministically constructing polymers of length $2^{\Theta(k^{3/2})}$ and $O((\log n)^{5/3})$ expected time by utilizing novel aspects of insertion systems. We also prove these systems are asymptotically optimal in both the length of the polymers they construct and the construction time.

2 Definitions

2.1 Grammars

A *context-free grammar* $\mathcal{G}$ is a 4-tuple $\mathcal{G} = (\Sigma, \Gamma, \Delta, S)$. The sets Σ and Γ are the *terminal* and *non-terminal symbols* of the grammar. The set Δ consists of *production rules* or simply *rules*, each of the form $L \to R_1 R_2 \cdots R_j$ with $L \in \Gamma$ and $R_i \in \Sigma \cup \Gamma$. Finally, the symbol $S \in \Gamma$ is a special *start symbol*. The *language of* $\mathcal{G}$, denoted $L(\mathcal{G})$, is the set of strings that can be *derived* by starting with S, and repeatedly replacing a non-terminal symbol found on the left-hand side of some rule in Δ with the sequence of symbols on the right-hand side of the rule. The *size* of $\mathcal{G}$ is $|\Delta|$, the number of rules in $\mathcal{G}$. If every rule in Δ is of the form $L \rightarrow R_1R_2$ or $L \rightarrow t$, with $R_1R_2 \in \Gamma$ and $t \in \Sigma$, then the grammar is said to be in *Chomsky normal form*. Every context-free grammar can be converted to a grammar in Chomsky normal form while increasing the size of grammar by at most a factor of 2.

An *integer-pair grammar*, used in Section 3, is a context-free grammar in Chomsky normal form such that each non-terminal symbol is an integer pair (a, d), and each production rule has the form $(a, d) \rightarrow (a, b)(c, d)$ or $(a, d) \rightarrow t$.

2.2 Insertion Systems

An *insertion system* in the active self-assembly model of Dabby and Chen [4] carries out the construction of a linear *polymer* consisting of constant length *monomers*. A polymer grows incrementally by the insertion of a monomer at an *insertion site* between two existing monomers in the polymer, according to complementary bonding sites between the monomer and the insertion site.

An insertion system $\mathcal{S}$ is defined as a 4-tuple $\mathcal{S} = (\Sigma, \Delta, Q, R)$. The first element, Σ, is a set of symbols. Each symbol $s \in \Sigma$ has a *complement* s^*. We denote the complement of a symbol s as $\overline{s}$, i.e. $\overline{s} = s^*$ and $\overline{s^*} = s$. The set Δ is a set of *monomer types*, each assigned a *concentration*. Each monomer is specified by a quadruple $(a, b, c, d)^+$ or $(a, b, c, d)^-$, where $a, b, c, d \in \Sigma \cup \{s^* : s \in \Sigma\}$, and each concentration is a real number between 0 and 1. The sum of all concentrations in Δ must be at most 1. The two symbols $Q = (a, b)$ and $R = (c, d)$ are special two-symbol monomers that together form the *initiator* of $\mathcal{S}$. It is required that either $\overline{a} = d$ or $\overline{b} = c$. The *size* of $\mathcal{S}$ is $|\Delta|$, the number of monomer types in $\mathcal{S}$.

A *polymer* is a sequence of monomers $Qm_1m_2 \ldots m_nR$ where $m_i \in \Delta$ such that for each pair of adjacent monomers $(w, x, a, b)(c, d, y, z)$, either $\overline{a} = d$ or $\overline{b} = c$. The *length* of a polymer is the number of monomers, including Q and R, it contains. Each pair of adjacent monomer ends $(a, b)(c, d)$ form an *insertion site*. Monomers can be inserted into an insertion site $(a, b)(c, d)$ (and the sequence of monomers) according to the following rules (see Figure 1):

1. If $\overline{a} = d$, then any monomer $(\overline{b}, e, f, \overline{c})^+$ can be inserted.
2. If $\overline{b} = c$, then any monomer $(e, \overline{a}, \overline{d}, f)^-$ can be inserted.[1]

A monomer is inserted after time t, where t is an exponential random variable with rate equal to the concentration of the monomer type. The set of all polymers *constructed* by an insertion system is recursively defined as any polymer constructed by inserting a monomer into a polymer constructed by the system, beginning with the initiator. Note that the insertion rules guarantee by induction that for every insertion site $(a, b)(c, d)$, either $\overline{a} = d$ or $\overline{b} = c$.

[1] In [4], this rule is described as a monomer $(\overline{d}, f, e, \overline{a})^-$ that is inserted into the polymer as $(e, \overline{a}, \overline{d}, f)$.

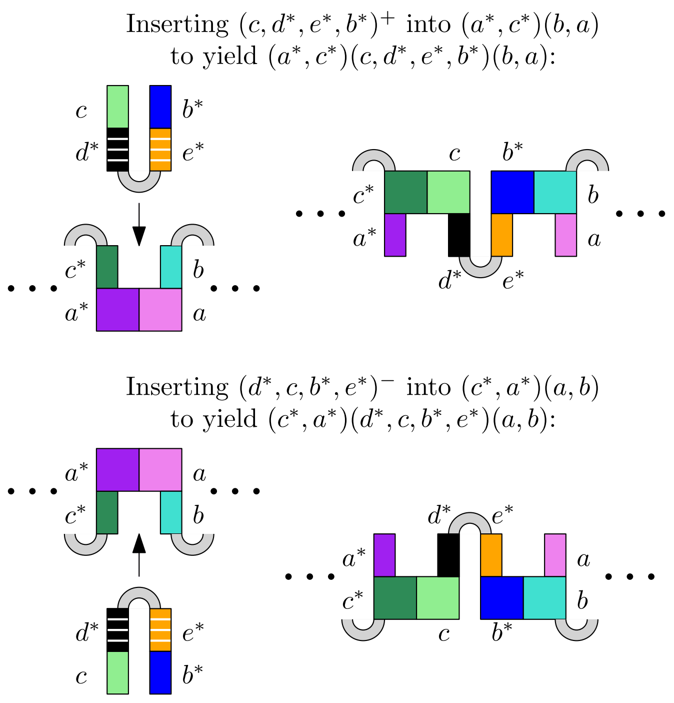

Inserting $(c, d^*, e^*, b^*)^+$ into $(a^*, c^*)(b, a)$ to yield $(a^*, c^*)(c, d^*, e^*, b^*)(b, a)$:
c d^* b^* e^*
c^* a^* b a
c b^* c^* a^* b a d^* e^*
Inserting $(d^*, c, b^*, e^*)^-$ into $(c^*, a^*)(a, b)$ to yield $(c^*, a^*)(d^*, c, b^*, e^*)(a, b)$:
a^* c^* a b
d^* c e^* b^*
d^* e^* a^* c^* a b c b^*

Fig. 1. A pictorial interpretation of the two insertion rules for monomers. Loosely based on Figure 2 and corresponding DNA-based implementation of [4].

We say that a polymer is *terminal* if no monomer can be inserted into any insertion site in the polymer, and that an insertion system *deterministically constructs* a polymer P if every polymer constructed by the system is either P or is non-terminal and has length less than that of P (i.e. can become P). The *stringification* of a polymer is the sequence of symbols in found on the polymer from left to right, e.g. $(a, b)(b^*, a, d, c)(c^*, a)$ has stringification abb^*adcc^*a. We call the set of stringifications of all terminal polymers of an insertion system $\mathcal{S}$ the *language* of $\mathcal{S}$, denoted $L(\mathcal{S})$.

2.3 Expressive Power

Intuitively, a system *expresses* another if the terminal polymers or strings created by the system "look" like the terminal polymers or strings created by the other system. In the simplest instance, an integer-pair grammar $\mathcal{G}'$ is said to *express* a context-free grammar $\mathcal{G}$ if $L(\mathcal{G}') = L(\mathcal{G})$. Similarly, a grammar $\mathcal{G}$ is said to *express* an insertion system $\mathcal{S}$ if $L(\mathcal{S}) = L(\mathcal{G})$, i.e. if the set of stringifications of the terminal polymers of $\mathcal{S}$ equals the language of $\mathcal{G}$.

An insertion system $\mathcal{S} = (\Sigma', \Delta', Q', R')$ is said to express a grammar $\mathcal{G} = (\Sigma, \Gamma, \Delta, S)$ if there exists a function $g : \Sigma' \cup \{s^* : s \in \Sigma'\} \to \Sigma \cup \{\varepsilon\}$ such that

$\{g(s'_1)g(s'_2)\ldots g(s'_n) : s'_1s'_2\ldots s'_n \in L(\mathcal{S})\} = L(\mathcal{G})$. More precisely, we require that there exists a fixed integer κ such that for any substring $s'_{i+1}s'_{i+2}\ldots s'_{i+\kappa}$ in a string in $L(\mathcal{S})$, $\{g(s'_{i+1}), g(s'_{i+2}), \ldots, g(s'_{i+\kappa})\} \neq \{\varepsilon\}$. That is, the insertion system symbols mapping to grammar terminal symbols are evenly distributed throughout the polymer. The requirement of a fixed integer κ prevents the possibility of a polymer containing arbitrarily long and irregular regions of "garbage" monomers.

3 The Expressive Power of Insertion Systems

Dabby and Chen proved that any insertion system has a context-free grammar expressing it. They construct such a grammar by creating a non-terminal for every possible pair of adjacent monomer types, and a production rule with this left-hand side non-terminal for each monomer that can be inserted into the insertion site formed by this pair. Here we give a reduction in the other direction, resolving, in the affirmative, the question posed by Dabby and Chen of whether context-free grammars and insertion systems have the same expressive power:

Theorem 1. *For every context-free grammar G, there exists an insertion system that expresses G.*

The primary difficulty in proving Theorem 1 lies in developing a way to simulate the "complete" replacement that occurs during derivation with the "incomplete" replacement that occurs when an insertion site is inserted into. For instance, $bcAbc$ becomes $bcDDbc$ via a production rule $A \to DD$ and A is completely replaced by DD. On the other hand, inserting a monomer $(b^*, d, d, c)^+$ into a site $(a, b)(c^*, a^*)$ yields the consecutive sites $(a, b)(b^*, d)$ and $(d, c)(c^*, a^*)$, with $(a, b)(c^*, a^*)$ only partially replaced – the left side of the first site and the right side of second site together form the initial site. This behavior constrains how replacement can be captured by insertion sites, and the κ parameter of the definition of expression (Section 2.3) prevents eliminating the issue via additional insertions.

We overcome this difficulty by proving Theorem 1 in two steps. First, we prove that integer-pair grammars, a constrained type of grammar with incomplete replacements, are able to express context-free grammars (Lemma 1). Second, we prove integer-pair grammars can be expressed by insertion systems (Lemma 2).

Lemma 1. *For every context-free grammar $\mathcal{G}$, there exists an integer-pair grammar that expresses $\mathcal{G}$.*

Lemma 2. *For every integer-pair grammar $\mathcal{G}$, there exists an insertion system that expresses $\mathcal{G}$.*

Proof. Let $\mathcal{G} = (\Sigma, \Gamma, \Delta, S)$. The integer-pair grammar $\mathcal{G}$ is expressed by an insertion system $\mathcal{S} = (\Sigma', \Delta', Q', R')$ that we now define. Let $\Sigma' = \{s_a, s_b : (a, b) \in \Gamma\} \cup \{u, x\} \cup \Sigma$. Let $\Delta' = \Delta'_1 \cup \Delta'_2 \cup \Delta'_3 \cup \Delta'_4$, where

$$\Delta'_1 = \{(s_b, u, s_b^*, x)^- : (a, d) \to (a, b)(c, d) \in \Delta\}$$

$$\Delta'_2 = \{(s_a, s_b, s_c^*, s_d^*)^+ : (a,d) \to (a,b)(c,d) \in \Delta\}$$

$$\Delta'_3 = \{(x, s_c, u^*, s_c^*)^- : (a,d) \to (a,b)(c,d) \in \Delta\}$$

$$\Delta'_4 = \{(s_a, t, x, s_d^*)^+ : (a,d) \to t \in \Delta\}$$

We give each monomer type equal concentration, although the precise concentrations are not important for expressive power. Let $Q' = (u^*, a^*)$ and $R' = (b, u)$, where $S = (a, b)$.

Insertion Types. We start by proving that for any polymer constructed by $\mathcal{S}$, only the following types of insertions of a monomer m_2 between two adjacent monomers m_1m_3 are possible:

1. $m_1 \in \Delta'_2$, $m_2 \in \Delta'_3$, $m_3 \in \Delta'_1$
2. $m_1 \in \Delta'_3$, $m_2 \in \Delta'_2 \cup \Delta'_4$, $m_3 \in \Delta'_1$
3. $m_1 \in \Delta'_3$, $m_2 \in \Delta'_1$, $m_3 \in \Delta'_2$

Moreover, we claim that for every adjacent m_1m_3 pair satisfying one of these conditions, an insertion *is* possible. That is, there is a monomer m_2 that can be inserted, necessarily from the monomer subset specified.

Consider each possible combination of $m_1 \in \Delta'_i$ and $m_3 \in \Delta'_j$, respectively, with $i, j \in \{1, 2, 3, 4\}$. Observe that for an insertion to occur at insertion site $(a, b)(c, d)$, the symbols $\overline{a}$, $\overline{b}$, $\overline{c}$, and $\overline{d}$ must each occur on some monomer. Then since x^* and t^* do not appear on any monomers, any i, j with $i \in \{1, 4\}$ or $j \in \{3, 4\}$ cannot occur. This leaves monomer pairs (Δ'_i, Δ'_j) with $(i, j) \in \{(2, 1), (2, 2), (3, 1), (3, 2)\}$.

Insertion sites between (Δ'_2, Δ'_1) pairs have the form $(s_c^*, s_d^*)(s_b, u)$, so an inserted monomer must have the form $(s_e, s_c, s_u^*, s_f)^-$ and is in Δ'_3. An insertion site $(s_c^*, s_d^*)(s_b, u)$ implies a rule of the form $(e, d) \to (e, f)(c, d)$ in Δ, so there exists a monomer $(x, s_c, u^*, s_c^*)^- \in \Delta'_3$ that can be inserted.

Insertion sites between (Δ'_3, Δ'_2) pairs have the form $(u^*, s_c^*)(s_a, s_b)$, so an inserted monomer must have the form $(\_, u, s_b^*, \_)^-$ and thus is in Δ'_1. An insertion site $(u^*, s_c^*)(s_a, s_b)$ implies a rule of the form $(a, d) \to (a, b)(e, d)$ in Γ, so there exists a monomer $(s_b, u, s_b^*, x)^- \in \Delta'_1$ that can be inserted.

Insertion sites between (Δ'_2, Δ'_2) pairs can only occur once a monomer $m_2 \in \Delta'_2$ has been inserted between a pair of adjacent monomers m_1m_3 with either $m_1 \in \Delta'_2$ or $m_3 \in \Delta'_2$, but not both. But we just proved that all such such possible insertions only permit $m_2 \in \Delta'_3 \cup \Delta'_1$. Moreover, the initial insertion site between Q' and R' has the form $(u^*, s_a^*)(s_b, u)$ of an insertion site with $m_1 \in \Delta'_3$ and $m_3 \in \Delta'_1$. So no pair of adjacent monomers m_1m_3 are ever both from Δ'_2 and no insertion site between (Δ'_2, Δ'_2) pairs can ever exist.

Insertion sites between (Δ'_3, Δ'_1) pairs have the form $(u^*, s_c^*)(s_b, u)$, so an inserted monomer must have the form $(s_c, \_, \_, b^*)^+$ or $(\_, u, u^*, \_)^-$ and is in Δ'_2 or Δ'_4. We show by induction that for each such insertion site $(u^*, s_c^*)(s_b, u)$ that $(c, b) \in \Gamma$. First, observe that this is true for the insertion site $(u^*, s_a^*)(s_b, u)$ between Q' and R', since $(a, b) = S \in \Gamma$. Next, suppose this is true for all insertion sites of some polymer and a monomer $m_2 \in \Delta'_2 \cup \Delta'_4$ is about to be inserted into the polymer between monomers from Δ'_3 and Δ'_1. Inserting a monomer $m_2 \in \Delta'_4$ only reduces the set of insertion sites between monomers in Δ'_3 and Δ'_1, and the inductive hypothesis holds. Inserting a monomer $m_2 \in \Delta'_2$ induces new (Δ'_3, Δ'_2) and (Δ'_2, Δ'_1) insertion site pairs between $m_1 m_2$ and $m_2 m_3$. These pairs must accept two monomers $m_4 \in \Delta_1$ and $m_5 \in \Delta_3$, inducing a sequence of monomers $m_1 m_4 m_2 m_5 m_3$ with adjacent pairs (Δ'_3, Δ'_1), (Δ'_1, Δ'_2), (Δ'_2, Δ'_3), (Δ'_3, Δ'_1). Only the first and last pairs permit insertion and both are (Δ'_3, Δ'_1) pairs.

Now consider the details of the three insertions yielding $m_1 m_4 m_2 m_5 m_3$, starting with $m_1 m_3$. The initial insertion site $m_1 m_3$ must have the form $(u^*, s_a^*)(s_d, u)$. So the sequence of insertions has the following form, with the last two insertions interchangeable. The symbol $\diamond$ is used to indicate the site being modified and the inserted monomer shown in bold:

$$(u^*, s_a^*) \diamond (s_d, u)$$

$$(u^*, s_a^*) \diamond \boldsymbol{(s_a, s_b, s_c^*, s_d^*)}(s_d, u)$$

$$(u^*, s_a^*)\boldsymbol{(s_b, u, s_b^*, x)}(s_a, s_b, s_c^*, s_d^*) \diamond (s_d, u)$$

$$(u^*, s_a^*)(s_b, u, s_b^*, x)(s_a, s_b, s_c^*, s_d^*)\boldsymbol{(x, s_c, u^*, s_c^*)}(s_d, u)$$

The two resulting (Δ'_3, Δ'_1) pair insertion sites are (u^*, s_a^*) (s_b, u) and $(u^*, s_c^*)(s_d, u)$. Assume, by induction, that the monomer m_2 must exist. So there is a rule $(a, d) \to (a, b)(c, d) \in \Delta$ and $(a, b), (c, d) \in \Gamma$, fulfilling the inductive hypothesis. So for every insertion site $(u^*, s_c^*)(s_b, u)$ between a (Δ'_3, Δ'_1) pair there exists a non-terminal $(c, b) \in \Gamma$. So for every adjacent monomer pair $m_1 m_3$ with $m_1 \in \Delta'_3$ and $m_3 \in \Delta'_1$, there exists a monomer $m_2 \in \Delta'_2 \cup \Delta'_4$ that can be inserted between m_1 and m_2.

Partial Derivations and Terminal Polymers. Next, consider the sequence of insertion sites between (Δ'_3, Δ'_1) pairs in a polymer constructed by a modified version of $\mathcal{S}$ lacking the monomers of Δ'_4. We claim that there is a constructed polymer with a sequence $(u^*, s_{a_1}^*)(s_{b_1}, u), (u^*, s_{a_2}^*)(s_{b_2}, u), \ldots, (u^*, s_{a_i}^*)(s_{b_i}, u)$ of these insertion sites if and only if there is a partial derivation $(a_1, b_1)(a_2, b_2) \ldots (a_i, b_i)$ of a string in $L(\mathcal{G})$. This follows directly from the previous proof by observing that two new adjacent (Δ'_3, Δ'_1) pair insertion sites $(u^*, s_a^*)(s_b, u)$ and $(u^*, s_c^*)(s_d, u)$ can replace a (Δ'_3, Δ'_1) pair insertion site if and only if there exists a rule $(a, d) \to (a, b)(c, d) \in \Delta$.

Observe that any string in $L(\mathcal{G})$ can be derived by first deriving a partial derivation containing only non-terminals, then applying only rules of the form $(a, d) \to t$. Similarly, since the monomers of Δ'_4 never form half of a valid insertion site, any terminal polymer of $\mathcal{S}$ can be constructed by first generating a polymer containing only monomers in $\Delta'_1 \cup \Delta'_2 \cup \Delta'_3$, then only inserting monomers from Δ'_4. Also note that the types of insertions possible in $\mathcal{S}$ imply that in any terminal polymer, any triple of adjacent monomers $m_1 m_2 m_3$ with $m_1 \in \Delta'_i$, $m_2 \in \Delta'_j$, and $m_3 \in \Delta'_k$, that $(i, j, k) \in \{(4, 1, 2), (1, 2, 3), (2, 3, 4), (3, 4, 1)\}$, with the first and last monomers of the polymer in Δ'_4.

Expression. Define the following piecewise function $g : \Sigma' \cup \{s^* : s \in \Sigma'\} \rightarrow \Sigma \cup \{\varepsilon\}$ that maps to ε except for the second symbols of monomers in Δ'_4.

$$g(s) = \begin{cases} t, \text{ if } t \in \Sigma \\ \varepsilon, \text{ otherwise} \end{cases}$$

Observe that every string in $L(\mathcal{S})$ has length $2+4\cdot(4n-3)+2 = 16n-8$ for some $n \geq 0$. Also, for each string $s'_1 s'_2 \dots s'_{16n-8} \in L(\mathcal{S})$, $g(s'_1)g(s'_2)\dots g(s'_{16n-8}) = \varepsilon^3 t_1 \varepsilon^{16} t_2 \varepsilon^{16} \dots t_n \varepsilon^5$. There is a terminal polymer with stringification in $L(\mathcal{S})$ yielding the sequence $s_1 s_2 \dots s_n$ if and only if the polymer can be constructed by first generating a terminal polymer excluding Δ'_4 monomers with a sequence of (Δ'_3, Δ'_1) insertion pairs $(a_1, b_1)(a_2, b_2)\dots(a_n, b_n)$ followed by a sequence of insertions of monomers from Δ'_4 with second symbols $t_1 t_2 \dots t_n$. Such a generation is possible if and only if $(a_1, b_1)(a_2, b_2)\dots(a_n, b_n)$ is a partial derivation of a string in $L(\mathcal{G})$ and $(a_1, b_1) \rightarrow t_1, (a_2, b_2) \rightarrow t_2, \dots, (a_n, b_n) \rightarrow t_n \in \Delta$. So applying the function g to the stringifications of the terminal polymers of $\mathcal{S}$ gives $L(\mathcal{G})$, i.e. $L(\mathcal{S}) = L(\mathcal{G})$. Moreover, the second symbol in every fourth monomer in a terminal polymer of $\mathcal{S}$ maps to a symbol of Σ using g. So $\mathcal{S}$ expresses $\mathcal{G}$ with the function g and $\kappa = 16$. □

4 Positive Results for Polymer Growth

Dabby and Chen also consider the size and speed of constructing finite polymers. They give a construction achieving the following result:

Theorem 2 ([4]). *For any positive integer r, there exists an insertion system with $O(r^2)$ monomer types that deterministically constructs a polymer of length $n = 2^{\Theta(r)}$ in $O(\log^3 n)$ expected time.*

We improve on this construction significantly in both polymer length and expected running time. In Section 5 we prove that our construction is the best possible with respect to both the polymer length and construction time.

Theorem 3. *For any positive integer r, there exists an insertion system with $O(r^2)$ monomer types that deterministically constructs a polymer of length $n = 2^{\Theta(r^3)}$ in $O((\log n)^{5/3})$ expected time.*

Proof. We give a constructive proof. The approach is to implement a three variable counter where each variable ranges over the values 0 to r, effectively carrying out the execution of a triple for-loop. Insertion sites of the form $(s_a, s_b)(s_c, s^*_a)$ are used to encode the state of the counter, where a, b, and c are the variables of the outer, inner, and middle loops, respectively.

1. (Inner): If $0 \leq b < r$, then $(s_a, s_b)(s_c, s^*_a)$ becomes $(s_a, s_{b+1})(s_c, s^*_a)$.
2. (Middle): If $b = r$ and $0 \leq c < r$, then $(s_a, s_b)(s_c, s^*_a)$ becomes $(s_a, s_0)(s_{c+1}, s^*_a)$.
3. (Outer): If $b = c = r$ and $0 \leq a < r$, then $(s_a, s_b)(s_c, s^*_a)$ becomes (s_{a+1}, s_0) (s_0, s^*_{a+1}).

A site is *modified* by a sequence of monomer insertions that yields a new usable site where all other sites created by the insertion sequence are unusable. For instance, we modify a site $(s_a, \boldsymbol{s_b})(s_c, s_a^*)$ to become $(s_a, \boldsymbol{s_d})(s_c, s_a^*)$, written $(s_a, s_b)(s_c, s_a^*) \rightarrow (s_a, s_d)(s_c, s_a^*)$, by adding the monomer types $(s_b^*, x, u, s_c^*)^+$ and $(x, u^*, s_a, s_b)^-$ to the system, where x is a special symbol whose complement is not found on any monomer. These two monomer types cause the following sequence of insertions, using $\diamond$ to indicate the site being modified and the inserted monomer shown in bold:

$$(s_a, s_b) \diamond (s_c, s_a^*)$$
$$(s_a, s_b)\boldsymbol{(s_b^*, x, u, s_c^*)} \diamond (s_c, s_a^*)$$
$$(s_a, s_b)(s_b^*, x, u, s_c^*)\boldsymbol{(x, u^*, s_a, s_d)} \diamond (s_c, s_a^*)$$

We call this simple modification, where a single symbol in the insertion site is replaced with another symbol, a *replacement*. Four types of replacements, seen in Table 1, can each be implemented by a pair of corresponding monomers.

Table 1. The four types of replacement steps and monomer pairs that implement them. The symbol u can be any symbol, and x is a special symbol whose complement does not appear on any monomer.

Replacement	Monomers
$(s_a, \boldsymbol{s_b})(s_c, s_a^*) \rightarrow (s_a, \boldsymbol{s_d})(s_c, s_a^*)$	$(s_b^*, x, u, s_c^*)^+$, $(x, u^*, s_a, s_d)^-$
$(s_a, s_b)(\boldsymbol{s_c}, s_a^*) \rightarrow (s_a, s_b)(\boldsymbol{s_d}, s_a^*)$	$(s_b^*, u, x, s_c^*)^+$, $(s_d, s_a^*, u^*, x)^-$
$(\boldsymbol{s_b}, s_a)(s_a^*, s_c) \rightarrow (\boldsymbol{s_d}, s_a)(s_a^*, s_c)$	$(x, s_b^*, s_c^*, u)^-$, $(u^*, x, s_d, s_a)^+$
$(s_b, s_a)(s_a^*, \boldsymbol{s_c}) \rightarrow (s_b, s_a)(s_a^*, \boldsymbol{s_d})$	$(u, s_b^*, s_c^*, x)^-$, $(s_a^*, s_d, x, u^*)^+$

Each of the three increment types are implemented using a sequence of site modifications. The resulting triple for-loop carries out a sequence of $\Theta(r^3)$ insertions, constructing a $\Theta(r^3)$-length polymer. A $2^{\Theta(r^3)}$-length polymer is achieved by simultaneously duplicating each site during each inner increment. Because the for-loop runs for $\Theta(r^3)$ steps and duplicates at a constant fraction of these steps (those with $0 \leq b < r$), the number of counters reaching the final $a = b = c = r$ state is $2^{\Theta(r^3)}$. In the remainder of the proof, we detail the implementation of each increment type, starting with the simplest: middle increments.

Middle Increment. A middle increment of a site $(s_a, s_b)(s_c, s_a^*)$ occurs when the site has the form $(s_a, s_r)(s_c, s_a^*)$ with $0 \leq c < r$, performing the modification $(s_a, s_r)(s_c, s_a^*) \rightarrow (s_a, s_0)(s_{c+1}, s_a^*)$. We implement middle increments using a sequence of three replacements:

$$(s_a, s_r)(s_c, s_a^*) \xrightarrow{1} (s_a, s_r)(s_{f_1(c)}, s_a^*) \xrightarrow{2} (s_a, s_0)(s_{f_1(c)}, s_a^*) \xrightarrow{3} (s_a, s_0)(s_{c+1}, s_a^*)$$

where $f_i(n) = n + 2ir^2$. The use of f is to avoid unintended interactions between monomers, since for any n_1, n_2 with $0 \leq n_1, n_2 \leq r$, $f_i(n_1) \neq f_j(n_2)$ for all $i \neq j$.

Compiling this sequence of replacements into monomer types gives the following set:

1. (Step 1): $(s_r^*, s_{f_2(c)}, x, s_c^*)^+$ and $(s_{f_1(c)}, s_a^*, s_{f_2(c)}^*, x)^-$.
2. (Step 2): $(s_r^*, x, s_{f_3(c)}, s_{f_1(c)}^*)^+$ and $(x, s_{f_3(c)}^*, s_a, s_0)^-$.
3. (Step 3): $(s_0^*, s_{f_4(c+1)}, x, s_{f_1(c)}^*)^+$ and $(s_{c+1}, s_a^*, s_{f_4(c+1)}^*, x)^-$.

Since each inserted monomer has an instance of x, all other insertion sites created are unusable. This is true of the insertions used for outer increments and duplications as well.

Outer Increment. An outer increment of the site $(s_a, s_b)(s_c, s_a^*)$ occurs when the site has the form $(s_a, s_r)(s_r, s_a^*)$ with $0 \le a < r$. We implement this step using a two-phase sequence of three (regular) replacements and a special quadruple replacement (Step 3):

$$(s_a, s_r)(s_r, s_a^*) \xrightarrow{1} (s_a, s_{f_5(a)})(s_r, s_a^*) \xrightarrow{2} (s_a, s_{f_5(a)})(s_{f_5(a)}^*, s_a^*)$$

$$(s_a, s_{f_5(a)})(s_{f_5(a)}^*, s_a^*) \xrightarrow{3} (s_{a+1}, s_{f_5(0)})(s_0, s_{a+1}^*) \xrightarrow{4} (s_{a+1}, s_0)(s_0, s_{a+1}^*)$$

At each step, a (necessary) complementary pair of symbols is maintained, which results in a sequence of more than 4 replacements. As with inner and middle increments, we compile replacement steps 1, 2, and 4 into monomers using Table 1. Step 3 is a special pair of monomers.

1. (Step 1): $(s_r^*, x, s_{f_6(r)}, s_r^*)^+$ and $(x, s_{f_6(r)}^*, s_a, s_{f_5(a)})^-$.
2. (Step 2): $(s_{f_5(a)}^*, s_{f_7(r)}^*, x, s_r^*)^+$ and $(s_{f_5(a)}^*, s_a^*, s_{f_7(r)}, x)^-$.
3. (Step 3): $(s_{f_5(a)}^*, x, s_{a+1}, s_{f_5(a)})^+$ and $(s_0, s_{a+1}^*, s_a, x)^-$.
4. (Step 4): $(s_{f_5(a)}^*, x, s_{f_7(r)}, s_0^*)^+$ and $(x, s_{f_7(r)}^*, s_{a+1}, s_0)^-$.

Inner Increment. The inner increment has two phases. The first phase performs the modification $(s_a, s_b)(s_c, s_a^*) \to (s_a, s_b)(s_{f_8(c)}, s_a^*) \ldots (s_a, s_{b+1})(s_c, s_a^*)$, yielding an incremented version of the original site and one other site. The second phase is $(s_a, s_b)(s_{f_8(c)}, s_a^*) \to (s_a, s_{b+1})(s_c, a^*)$, transforming the second site into an incremented version of the original site.

For the first phase, we use the three monomers $(s_b^*, s_{f_8(c)}, s_{f_8(b+1)}, s_c^*)^+$, $(s_{f_8(c)}, s_a^*, s_{f_8(c)}^*, x)^-$, and $(x, s_{f_8(b+1)}^*, s_a, s_{b+1})^-$ and call the entire phase Step 1. The site $(s_a, s_b)(s_{f_8(c)}, s_a^*)$ is transformed into $(s_a, s_{b+1})(s_c, s_a^*)$ by a sequence of replacement steps:

$$(s_a, s_b)(s_{f_8(c)}, s_a^*) \xrightarrow{2} (s_a, s_{f_9(b)})(s_{f_8(c)}, s_a^*) \xrightarrow{3} (s_a, s_{f_9(b)})(s_c, s_a^*) \xrightarrow{4} (s_a, s_{b+1})(s_c, s_a^*)$$

As with previous sequences of replacement steps, we compile this sequence into a set of monomers:

1. (Step 2): $(s_b^*, x, s_{f_{10}(b)}, s_{f_8(c)}^*)^+$ and $(x, s_{f_{10}(b)}^*, s_a, s_{f_9(b)})^-$.

2. (Step 3): $(s^*_{f_9(b)}, s_{f_{11}(c)}, x, s^*_{f_8(c)})^+$ and $(s_c, s^*_a, s^*_{f_{11}(c)}, x)^-$.
3. (Step 4): $(s^*_{f_9(b)}, x, s_{f_{12}(b+1)}, s^*_c)^+$ and $(x, s^*_{f_{12}(b+1)}, s_a, s_{b+1})^-$.

When combined, the two phases of duplication modify $(s_a, s_b)(s_c, s^*_a)$ to become $(s_a, s_{b+1})(s_c, s^*_a) \ldots (s_a, s_{b+1})(s_c, s^*_a)$, where all sites between the duplicated sites are unusable.

Putting It Together. The system starts with the intiator $(s_0, s_0)(s_0, s^*_0)$. Each increment of the counter occurs either through a middle increment, outer increment, or a duplication. There are at most $(r+1)^2$ monomer types in each family and $O(r^2)$ monomer types total. The size P_i of a subpolymer with an initiator encoding some value i between 0 and $(r+1)^3 - 1$ can be bounded by $2P_{i+2} + 9 \leq P_i 2P_{i+1} + 9$ with $P_{(r+1)^3-2} > 0$. So P_0, the size of the terminal polymer, is $2^{\Theta(r^3)}$.

Running Time. Define the concentration of each monomer type to be equal. There are less than $39r^2$ monomer types, so each monomer type has concentration at least $1/(39r^2)$. The polymer is complete as soon as every insertion site has been modified to be $(r, r)(r, r^*)$ and the monomer $(s^*_r, x, s_{f_7(r)}, s^*_r)^+$ has been inserted. There are fewer than 2^{8r^3} such insertions, and each insertion can occur once at most $9 \cdot 8r^3 = 72r^3$ previous insertions have occurred. So an upper bound on the expected time T_r for each such insertion is described as a sum of $72r^3$ random variables, each with expected time $39r^2$. The Chernoff bound for exponential random variables implies $\text{Prob}[T_r > 39r^2 \cdot 72r^3(1+\delta)] \leq e^{-r^5\delta/2}$ for all $\delta \geq 2$ and $T_{\mathcal{S}_r}$, the total running time of the system, has $\text{Prob}[T_{\mathcal{S}_r} > 39r^2 \cdot 72r^3(1+\delta)] \leq 2^{-r^5\delta/4}$ for all $\delta \geq 32$. So the expected value of $T_{\mathcal{S}_r}$, the construction time, is $O(r^5) = O((\log n)^{5/3})$ with an exponentially decaying tail probability. □

5 Negative Results for Polymer Growth

Here we prove that our system constructs polymers using an optimal number of monomer types and in optimal expected time.

Theorem 4. *Any polymer deterministically constructed by an insertion system with k monomer types has length $2^{O(k^{3/2})}$.*

Theorem 5. *Deterministically constructing a polymer of length n takes $\Omega((\log n)^{5/3})$ expected time.*

Acknowledgments. We wish to thank Benjamin Hescott for many helpful discussions and anonymous reviewers for feedback that helped to improve the presentation of the paper.

References

1. Adleman, L., Cheng, Q., Goel, A., Huang, M.-D.: Running time and program size for self-assembled squares. In: Proceedings of 33rd ACM Symposium on Theory of Computing (STOC) (2001)
2. Adleman, L.M., Cheng, Q., Goel, A., Huang, M.-D., Wasserman, H.: Linear self-assemblies: equilibria, entropy and convergence rates. In: Proceedings of 6th International Conference on Difference Equations and Applications (2001)
3. Chen, M., Xin, D., Woods, D.: Parallel computation using active self-assembly. In: Soloveichik, D., Yurke, B. (eds.) DNA 2013. LNCS, vol. 8141, pp. 16–30. Springer, Heidelberg (2013)
4. Dabby, N., Chen, H.-L.: Active self-assembly of simple units using an insertion primitive. In: Proceedings of 23rd ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 1526–1536 (2012)
5. Doty, D., Lutz, J.H., Patitz, M.J., Schweller, R.T., Summers, S.M., Woods, D.: The tile assembly model is intrinsically universal. In: Proceedings of 53rd IEEE Symposium on Foundations of Computer Sciences (FOCS), pp. 302–310 (2012)
6. Hendricks, J., Padilla, J.E., Patitz, M.J., Rogers, T.A.: Signal transmission across tile assemblies: 3D static tiles simulate active self-assembly by 2D signal-passing tiles. In: Soloveichik, D., Yurke, B. (eds.) DNA 2013. LNCS, vol. 8141, pp. 90–104. Springer, Heidelberg (2013)
7. Keenan, A., Schweller, R., Sherman, M., Zhong, X.: Fast arithmetic in algorithmic self-assembly. Technical report, arXiv (2013)
8. Keenan, A., Schweller, R., Zhong, X.: Exponential replication of patterns in the signal tile assembly model. In: Soloveichik, D., Yurke, B. (eds.) DNA 2013. LNCS, vol. 8141, pp. 118–132. Springer, Heidelberg (2013)
9. Klavins, E.: Universal self-replication using graph grammars. In: Proceedings of International Conference on MEMS, NANO, and Smart Systems, pp. 198–204 (2004)
10. Klavins, E., Ghrist, R., Lipsky, D.: Graph grammars for self assembling robotic systems. In: Proceedings of the International Conference on Robotics and Automation (ICRA), vol. 5, pp. 5293–5300 (2004)
11. Majumder, U., LaBean, T.H., Reif, J.H.: Activatable tiles: Compact, robust programmable assembly and other applications. In: Garzon, M.H., Yan, H. (eds.) DNA 2007. LNCS, vol. 4848, pp. 15–25. Springer, Heidelberg (2008)
12. Padilla, J.E., Patitz, M.J., Pena, R., Schweller, R.T., Seeman, N.C., Sheline, R., Summers, S.M., Zhong, X.: Asynchronous signal passing for tile self-assembly: fuel efficient computation and efficient assembly of shapes. In: Mauri, G., Dennunzio, A., Manzoni, L., Porreca, A.E. (eds.) UCNC 2013. LNCS, vol. 7956, pp. 174–185. Springer, Heidelberg (2013)
13. Rothemund, P.W.K., Winfree, E.: The program-size complexity of self-assembled squares (extended abstract). In: Proceedings of 32nd ACM Symposium on Theory of Computing (STOC), pp. 459–468 (2000)
14. Soloveichik, D., Winfree, E.: Complexity of self-assembled shapes. SIAM Journal on Computing 36(6), 1544–1569 (2007)
15. Winfree, E.: Algorithmic Self-Assembly of DNA. PhD thesis, Caltech (1998)
16. Woods, D., Chen, H.-L., Goodfriend, S., Dabby, N., Winfree, E., Yin, P.: Active self-assembly of algorithmic shapes and patterns in polylogarithmic time. In: Proceedings of 4th Conference on Innovations in Theoretical Compuer Science (ITCS), pp. 353–354 (2013)
17. Woods, D., Chen, H.-L., Goodfriend, S., Dabby, N., Winfree, E., Yin, P.: Active self-assembly of algorithmic shapes and patterns in polylogarithmic time. Technical report, arXiv (2013)

Trace Reconstruction Revisited

Andrew McGregor[1,⋆], Eric Price[2], and Sofya Vorotnikova[1]

[1] University of Massachusetts Amherst
{mcgregor,svorotni}@cs.umass.edu
[2] IBM Almaden Research Center
ecprice@mit.edu

Abstract. The trace reconstruction problem is to reconstruct a string x of length n given m random subsequences where each subsequence is generated by deleting each character of x independently with probability p. Two natural questions are a) how large must m be as a function of n and p such that reconstruction is possible with high probability and b) how can this reconstruction be performed efficiently. Existing work considers the case when x is chosen uniformly at random and when x is arbitrary. In this paper, we relate the complexity of both cases; improve bounds by Holenstein et al. (SODA 2008) on the sufficient value of m in both cases; and present a significantly simpler analysis for some of the results proved by Viswanathan and Swaminathan (SODA 2008), Kannan and McGregor (ISIT 2005), and Batu et al. (SODA 2004). In particular, our work implies the first sub-polynomial upper bound (when the alphabet is polylog n) and super-logarithmic lower bound on the number of traces required when x is random and p is constant.

1 Introduction

The basic trace reconstruction problem is to infer a string x of length n from m random subsequences $y^1, \ldots, y^m$ where each subsequence is generated by deleting each character of x independently with probability p. The random subsequences are referred to as *traces*. Two natural questions are a) how many traces (as a function of n and p) are required such that reconstruction is possible with high probability and b) how can this reconstruction be performed efficiently. Note that both questions are trivial if the entries of x were being substituted, rather than deleted. In that case, if $p < 1/2$ is constant and $m = O(\log n)$ then $x_i = \text{mode}(y_i^1, \ldots, y_i^m)$ with high probability. However, when there are deletions, there is no longer any clear way to align the subsequence and thereby decompose the problem into inferring each entry of x independently.

The original motivation for the problem was from computational biology where an active area of research is to reconstruct ancestral DNA sequences given the DNA sequences of the descendants. The above abstraction is a simplification of this problem in which we essentially restrict the possible mutations and assume the descendants are independent. The abstraction serves to both demonstrate

⋆ Supported by NSF CAREER Award CCF-0953754.

why the original problem is hard and exposures our lack of good algorithmic techniques for even basic inference problems. For example, for $p = 1/3$, it is not at all obvious whether the minimal sufficient value m has a polylogarithmic, polynomial, or exponential dependence on n.

Previous Work. The problem was introduced by Batu et al. [1] where they considered both an "average" case when x is chosen uniformly at random (in this case the probability of successful reconstruction is over both the choice of x and the deletions) and the case when x is chosen arbitrarily. In the average case, it was shown that $m = O(\log n)$ is sufficient if $p = O(1/\log n)$. This result was then extended to also handle insertions and substitutions by Kannan and McGregor [4] and Viswanathan and Swaminathan [8]. For small constant deletion probability, Holenstein et al. [3] showed that poly(n) traces was sufficient. While this last result represented a major step forward, it leaves open the question whether a polynomial number of traces is actually necessary or whether a logarithmic number would suffice, as in the case when there was only substitutions. For reconstructing an arbitrary x, Batu et al. [1] showed that $O(n \operatorname{polylog} n)$ traces suffices if $p = O(1/\sqrt{n})$ and Holenstein et al. [3] showed that $\exp(\sqrt{n} \cdot \operatorname{polylog} n)$ traces suffices if p is any constant.

A separate line of work considers the related problem of determining the value k such that the *k-deck* of any x uniquely determines x. The k-deck of x is the number of times each string of length k appears as a subsequence of x. Given a sufficient number of traces of length greater than k, we can compute the k-deck and thereby determine x if k is large enough. Scott [7] proved that $k = O(\sqrt{n \log n})$ and Dudik and Schulman [2] showed that $k = \exp(\Omega(\log^{1/2} n))$. We will make use of the first of these results in the last section.

Our Results. Our main results in the average case are that a) a sub-polynomial number of traces is sufficient if we consider a slightly larger alphabet and b) a super-logarithmic number of traces is necessary. In particular, if x is chosen uniformly from $[\sigma]^n$ where $\sigma = \Theta(\log n)$ and p is a small constant then

$$m = \exp(\sqrt{\log n} \cdot \operatorname{poly}(\log \log n))$$

traces are sufficient which contrasts with the bound $m = \exp(O(\log n))$ that was shown by Holenstein et al. for the binary case. We prove this result by establishing an almost tight relationship between the complexity in the average case to the complexity in the worst case. To do this, we first present a significantly simpler proof of the results of Batu et al. [1] and Viswanathan and Swaminathan [8]. It is then possible to extend the alternative approach to be robust to deletions that occur with constant probability.

In the case of arbitrary strings (binary or otherwise), we show that $m = \exp(\sqrt{n} \cdot \operatorname{polylog} n)$ traces are sufficient for all $p \leq 1 - c/\sqrt{n/\log n}$ for some constant $c > 0$. This result improves upon the other result by Holenstein et al. The previous result showed that the same number of traces were sufficient when the traces are random subsequences of length $\Theta(n)$. The new result shows that reconstruction is still possible even if the traces are only of length $\Theta(\sqrt{n \log n})$.

2 Preliminaries and Terminology

Given a string $x_1 x_2 \ldots x_n \in [\sigma]^n$, a *trace* generated with deletion probability p is a random subsequence of x, $y = x_{i_1} x_{i_2} x_{i_3} \ldots$ where $i_1 < i_2 < i_3 < \ldots$ and each $i \in [n]$ is present in the set $\{i_1, i_2, \ldots\}$ independently with probability $1-p$. It will sometimes be helpful to refer to the trace y as being *received* when x is *transmitted.* We are interested in whether it is possible to infer x with high probability from multiple independently traces $y^1, y^2, \ldots, y^m$.

We define $f(n,p,\sigma)$ to be the smallest value of m such that for any string $x \in [\sigma]^n$, m traces are sufficient to reconstruct x with high probability[1]. Define $g(n,p,\sigma)$ to be the smallest value of m such that for a *random* string $x \in_R [\sigma]^n$, m traces are sufficient to reconstruct x with high probability where the probability is taken over both the randomness of x and the generation of the traces. For example, existing results show that for small constant $c > 0$:

$$g(n,p,2) = \begin{cases} O(\log n) & \text{if } p \le c/\log n \\ \operatorname{poly} n & \text{if } p \le c \end{cases} .$$

We present a simple proof of the first part of this result and then prove that $g(n,p,\sigma)$ is sub-polynomial for small constant values of p if $\sigma = \Omega(\log n)$. To prove this result we show that for sufficiently large σ, $f(\log n, p, \sigma) \approx g(n,p,\sigma)$. Lastly, we prove $f(n,p,2) = \exp(\sqrt{n} \operatorname{polylog} n)$ for all $p \le 1 - O(1/\sqrt{n/\log n})$ whereas it was previously only known for constant p.

Note that any reconstruction algorithm for binary strings can be extended to a larger alphabet of size σ while increasing the number of traces by a factor of $O(\log \sigma)$. The following simple lemma includes the necessary details.

Lemma 1. *$f(n,p,\sigma) = O(\log \sigma) f(n,p,2)$. If m traces suffice to reconstruct a random string in $\{0,1\}^n$ with probability $1-\delta$, then m traces also suffice to reconstruct a random string in $[\sigma]^n$ with probability $1 - O(\delta \log \sigma)$.*

Proof. Suppose there exists an algorithm for arbitrary binary sequence reconstruction that uses m traces and has failure probability at most δ. By repeating the algorithm $O(\log \sigma)$ times and taking the modal answer we may reduce the failure probability to $\delta/\binom{\sigma}{2}$ at the expense of increasing the number of traces by a factor $O(\log \sigma)$. We will use the resulting algorithm to reconstruct a sequence x from a larger alphabet as follows. For each pair $i,j \in [\sigma]$, if we delete all occurrences of other characters in the traces then we can reconstruct the subsequence $x^{i,j}$ of x consisting of i's and j's. By the union bound we can do this for all pairs with probability of failure at most δ. For the resulting subsequences it is possible to construct x, e.g., we can learn the position of the kth j in x by summing over i the number of occurrences of i's before the kth j in $x^{i,j}$.

The same approach works to prove the bound for random strings except that since the failure probability in this case is taken over both the randomness of the initial string and the traces, we can't first boost the probability of success. □

[1] That is, probability at least $1 - 1/\operatorname{poly}(n)$

Notation. We denote the Hamming distance between two strings u, v by $\Delta(u,v) = |\{i : u_i \neq v_i\}|$. We write $e \in_R S$ to denote that the element e is chosen uniformly at random from the set S. A t-substring of x is a string consisting of t consecutive characters of x. Given a substring w of a trace, we define the pre-image of w, to be the range of indices of x under consideration when w was generated, e.g.,

$$w = x_{i_j} x_{i_{j+1}} \dots x_{i_k} \qquad \text{and} \qquad I(w) = \{i_j, i_j + 1, i_j + 2, \dots, i_k\} .$$

We say substrings u, v of two different traces *overlap* if $I(u) \cap I(v) \neq \emptyset$. Lastly, we use the notation $x_{[a,b]}$ to denote the substring $x_a x_{a+1} \dots, x_b$. Let $\mathcal{B}_{n,p}$ denote the binomial distribution with n trials and probability p.

3 Average Case Reconstruction

In this section we assume that the original string x is chosen uniformly at random from the set $[\sigma]^n$. We first present a simpler approach to reconstruction when the deletion probability is $O(1/\log n)$. Previous approaches were generally based on determining the characters of x from left to right, e.g., trying to maintain pointers to corresponding characters in the different traces and using the majority to determine the next character of x. While the resulting algorithms were relatively straight-forward, the analysis was rather involved.

In contrast, our approach is based on finding all sufficiently-long substrings of x independently and the analysis for our approach is significantly shorter and intuitive. While this simplicity is appealing in its own right, it also allows us to generalize the algorithm in the following section and prove a new result in the case of constant deletion probability. We start with a simple lemma about random binary strings.

Lemma 2. *With high probability, every pair of t-substrings of a random sequence $x \in \{0,1\}^n$ differ in at least $t/3$ positions if $t > 94 \ln n$.*

Proof. Consider two arbitrary substrings $u = x_i \dots x_{i+t-2}$ and $v = x_j \dots x_{j+t-2}$. Let $z \in \{0,1\}^t$ be defined by $z_k = x_{i+k-1} \oplus x_{j+k-1}$. Note that $\Delta(u,v) = \sum_k z_k$ and that bits of z_i are fully independent. Hence, $E\left[\sum_k z_k\right] = t/2$ and by an application of the Chernoff bound, $\Pr\left[\sum_k z_k \leq t/3\right] \leq \exp(-t/24)$. Therefore, if $t > 94 \ln n$, then $\Pr\left[\Delta(u,v) \leq t/3\right] \leq 1/n^4$. Applying the union bound over all $\binom{n}{2}$ choices for substrings u and v establishes that the Hamming distance between all pairs is at least $t/3$ with probability at least $1 - 1/n^2$. □

3.1 Warmup: Inverse Logarithmic Deletion Probability

In this section we present a simple proof of the results by Batu et al. [1] when $p = O(1/\log n)$. For the rest of the section we let the deletion probability be $p \leq c_1/\log n$, number of traces be $m = c_2 \log n$ for constants $c_1, c_2 > 0$.

Basic Idea and Algorithm. The idea behind the approach is simple and intuitive. For $t = c_3 \log n$ where c_3 is some sufficiently large constant, the following statements hold with high probability:

1. The set of all t-substrings of a random string x, uniquely defines x.
2. w is a t-substring of x iff w is a t-substring of at least $3/4$ of the traces.

Therefore, it is sufficient to check each t-substring of each trace to see whether it appears in at least $3/4$ of all the traces. The next lemma establishes the first statement.

Lemma 3. *The set of t-substrings of $x \in_R [\sigma]^n$ uniquely define x whp.*

Proof. If all $(t-1)$-substrings of x are unique, then for a t-substring w starting at index i in x, there is a unique t-substrings starting at $i+1$. By repeating this process, we can recover the original string x. The fact that all $(t-1)$-substrings are unique with high probability follows from Lemma 2. □

The next two lemmas establish the if-and-only-if of the second bullet point.

Lemma 4. *Every $4t$-substring of x passes the test with high probability. In particular, none of the characters of any $4t$-substring are deleted in at least $3m/4$ of the traces.*

Proof. Let w be a $4t$-substring of x. Let F be the number of traces where w appears. The probability that w appears in a particular trace is at least $(1-p)^{4t} > 1 - 4pt > 7/8$ if $pt = c_1 c_3 < 1/32$. Hence, $E[F] > 7m/8$ and $\Pr[F \leq 3m/4] < e^{-m/168}$ by an application of the Chernoff bound. If $m > 2 \cdot 168 \ln n$, this probability is at most $1/n^2$. □

Lemma 5. *Any t-string that passes the test is a t-substring of x whp.*

Proof. We start with two simple claims that each hold with high probability:

1. *For any t-substrings w and v of different traces, if $w = v$ then w and v have overlapping pre-images.* This follows because the probability that two non-overlapping t-substrings are equal is $1/2^t$ by considering the randomness of x. There are less than $(mn)^2$ pairs of t-substrings and hence the claim doesn't hold with probability at most $(mn)^2/2^t \leq 1/n^2$ if $t > 4 \log n + 2 \log(c_2 \log n)$.
2. *The pre-image of any t-substring w of a trace has length at most $2t$.* The follows because the probability that more than half of the characters in a $2t$-substring of x are deleted is at most $\exp(-t/12)$ by an application of the Chernoff bound. Since there are at most mn^2 such sequences, the claim doesn't hold with probability at most $mn^2 \exp(-t/12) \leq 1/n^2$ if $t > 48 \ln n + 48 \ln(c_2 \log n)$.

Suppose w equals the substrings $w^1, w^2, \ldots, w^h$ in the other traces for $h \geq 3m/4 - 1$. It follows from the above claims that the pre-images of $w, w^1, w^2, \ldots, w^h$ are contained in a contiguous region of x of size $4t$. However, by Lemma 4 we know the corresponding substring of x was transmitted with deletions in at most $m/4$ of the traces. Therefore, at least $h - m/4 > 1$ of the substrings $w^1, w^2, \ldots, w^h$ correspond exactly to a t-strings of x. Hence w equals a substrings of x. □

Insertions, Deletions, and Substitutions. Viswanathan and Swaminathan [8] extended the above result to handle the case where, in addition to deletions, each character is substituted by a random character with probability α and random characters are inserted with probability q. Specifically, each character x_i is transformed independently as follows:

$$g(x_i) = \begin{cases} Sx_i \text{ with probability } 1-p-\alpha(1-p) \\ Sc \text{ with probability } \alpha(1-p) \\ S \text{ with probability } p \end{cases}$$

where $c \in_R [\sigma]$, $S \in_R [\sigma]^k$ and k is a random variable distributed as a Geometric random variable with parameter $1-q$. In particular,

$$\Pr[g(x_i) = x_i] \geq (1-p-\alpha(1-p))(1-q) \ ,$$

which is $1-\alpha-o(1)$ if $p, q = o(1)$. In this section we present a simple proof of Viswanathan and Swaminathan's result that $O(\log n)$ traces are sufficient for reconstruction if $p, q < c/\log n$ and $\alpha < c$ for some sufficiently small constant c.

Basic Idea and Algorithm. We extend the substring test as follows: w is a t-substring of x iff for some t-substring w' of a trace, there exists t-substrings $w_1, \ldots w_{3m/4}$ in different traces such that $\Delta(w', w_i) \leq 3\alpha t$ for all $i \in [3m/4]$ and

$$w = \text{average}(w_1, \ldots, w_{3m/4})$$

where *average* is taking the mode of each of the t character positions.

Lemma 6. *Every t-substring of x passes the test with high probability.*

Proof. Let w be an arbitrary t-substring of x and let $w' = g(w)$ be the resulting substring in some specific trace. In what follows we assume the constant c governing the deletion and insertion probabilities is sufficiently small. The probability no insertions or deletions occurred during the transmission of w is $(1-q-p+pq)^t \geq 6/7$ and by an application of the Chernoff bound, the number of substitutions is at most $3\alpha t/2$. Hence, by a further application of the Chernoff bound there are at least $5m/6$ traces that contain a t-substring whose Hamming distance is at most $3\alpha t/2$ from w. The Hamming distance between these traces is at most $3\alpha t$ by the triangle inequality. Lastly if these t-substrings are averaged character-wise then the resulting string equals w because with high probability each character of w is flipped in at most $1/3$ of the transmissions. □

Lemma 7. *Every t-string that passes the test is a t-substring of x whp.*

Proof. Suppose a trace contains a t-substring w' such that for some $h \geq 3m/4$, there exists $w_1, \ldots w_h$ in different traces such that $\Delta(w', w_i) \leq 3\alpha t < t/3$ for sufficiently small α. We infer that each w_i overlaps with w' since otherwise w_i and w' are random strings and will differ in at least $t/3$ places with high probability. Hence, each w_i comes from substring x' of x of length $4t$. When

x' was transmitted, it was transmitted without any insertions or deletions in at least of 9/10 of the traces with high probability. Hence, all but at most $m/10$ of the w_i resulted from transmission with no insertions or deletions. But appealing to Lemma 2 we deduce that these w_i actually correspond to the same t-substring of x; otherwise there would be a pair of different t-substrings of x that were sufficiently similar that after bits were flipped with only probability α then the strings would be closer than $6\alpha t$ apart. Hence, when averaging $w_1, \ldots w_h$ character-wise at most a $2\alpha + (m/10)/h \leq 2\alpha + 2/15 < 1/2$ fraction of characters will not be correct. Hence, the majority will be correct. □

3.2 Constant Deletion Probability

In this section we again restrict our attention to the deletion case but now consider p to be a small constant. In the previous two results, the crucial step was being able to identify t-substrings in different traces that were overlapping. Initially, it was sufficient to look for identical t-substrings but then we had to relax this to finding pairs of substrings that were close in Hamming distance. The main idea in this section is the observation that it is possible to find overlapping t-substrings by computing the length of the longest common subsequence between the substrings.

Lemma 8. *If $t = c \log n$ for some large constant $c > 0$, the following claims hold with high probability:*

- *For any two traces y and y' and any t-substring w in y, there exists a t-substring w' in y' such that* $\text{lcs}(w, w') \geq 0.99t$.
- *For any non-overlapping t-substrings w and v in different traces* $\text{lcs}(w, v) < 0.99t$.

Proof. For the first part of the lemma, note that the expected number of deletions during the transmission of a t-substring of x is pt and by an application of the Chernoff bound we may assume it is never larger than $2pt$ with high probability if t is sufficiently large multiple of $\log n$. Therefore, there are at least $(1 - 2p)t$ characters of some t-substring u of x in w. But any t-substring of y' whose pre-image covers u, will also have $(1 - 2p)t$ characters of u. Let w' be such a string. Then, $\text{lcs}(w, w') \geq (1 - 4p)t \geq 0.99t$ for sufficiently small constant p.

To prove the second part of the lemma suppose w, v are non-overlapping t-substrings. Because x is random and w, v are non-overlapping, w, v are independent random strings. Therefore,

$$\Pr\left[\text{lcs}(w, v) \geq 0.99t\right] < \binom{t}{0.99t}^2 1/2^{0.99t} < 2^{2tH(0.99) - 0.99t} < 2^{-0.8t}$$

where $H(p) = -p \log p - (1 - p) \log(1 - p)$. The first inequality follows by considering the $\binom{t}{0.99t}$ subsequences of each segment that might be equal. □

It is likely that the constants in the above lemma can be improved. However, one of the main ingredients in the proof is determining the length of the longest common subsequence of two random strings. Determining even the expected length is a long standing open question (see, e.g., [5] and references therein).

Reduction to Short Sequence Reconstruction. To prove the constant deletion result, the strategy is to reduce the problem of reconstructing a random $x \in_R [\sigma]^n$ to reconstructing $O(n)$ arbitrary strings each of length $O(\log n)$. To do this, we will have to assume that x is chosen randomly from a larger alphabet $\sigma = \Theta(\log n)$. It will then follow that

$$g(n, p, \sigma) \le f(O(\log n), p, \sigma) = \exp(\sqrt{\log n}\, \mathrm{poly}(\log\log n)) \ ,$$

by appealing to the bounds on the function f established in the next section. To establish this reduction we need the notion of a *useful character.*

Definition 1. *We say a character x_i from x is a* useful character *if:*

1. *The character was not deleted when generating the first trace.*
2. *$x_i \neq x_j$ for all $|i - j| \le 8t$, i.e., x_i is locally unique.*

The goal is to identify the occurrence of useful characters in the traces and then determine with high probability which characters correspond to the same x_i. The next lemma establishes that the number of non-useful characters between two consecutive useful characters is $O(\log n)$. Since each useful character will occur in all but about a p fraction of the traces, there are roughly a $(1 - p)^2$ fraction of traces that have any pair of consecutive useful characters. We then use the substrings of the traces between these useful characters to reconstruct the substring of x between the useful characters. We can then solve the sequence reconstruction problem on these substrings.

Note that because there are $O(n)$ substrings and each has length $O(\log n)$ we now need a reconstruction algorithm that works for all strings (rather than working for random strings with high probability). Note that the algorithm for reconstruction of arbitrary strings presented in Section 4 can be assumed to have exponentially small failure probability without any significant change in the number of traces required (i.e., repeating the algorithm $\mathrm{poly}(n)$ times to boost success probability is not a significant increase when the number of traces is already super-polynomial). This is important since we need the failure probability on length $O(\log n)$ instances to be $1/\mathrm{poly}(n)$ since there are $O(n/\log n)$ such instances.

Lemma 9. *With high probability, there exists a useful character in every r-substring of x if $r = 8t = 8c \log n$.*

Proof. Consider an arbitrary r-substring $x_{[i,i+r-1]}$. With high probability there exists more than $2r/3$ distinct characters in this substring if the alphabet is sufficiently large. Of these, at most $r/2$ can occur twice or more. Hence, there are at least $r/6$ characters that occur exactly once. Of these, $(1 - p)r/6 > r/7$ occur in the first trace in expectation and hence the probability that none of them appear in the first trace is at most $p^{r/7} < 1/n^2$ for sufficiently small constant p. □

Algorithm. The algorithm for finding corresponding characters is as follows:

- For each character a in the first trace, consider the t-substring w_1 of the first trace centered at this character (or as close as possible in the case when is a is near the start of end of the trace).
- *Find Overlapping Substrings:* Identify t-substrings of the other traces $w_2, \ldots, w_m$ such that each satisfies $\text{lcs}(w_1, w_i) \geq 0.99t$.
- *Check Local Uniqueness:* For each w_i consider the $8t$-substring w_i' of the same trace centered at w_i. If a occurs twice in any w_i' abort.
- *Match:* Otherwise, conclude that any occurrence of a in w_i corresponds to the same character of x.

The correctness of the algorithm follows from Lemma 8. Specifically, the lemma implies that with high probability the pre-images of every w_i are contained in a contiguous set of at most $4t$ indices. However, this contiguous set is a subset of the pre-image of each w_i'. Hence, if a occurs twice within the contiguous set the algorithm will abort. Otherwise, all occurrences of a in the w_i must correspond to the same index.

Relationship between *f* and *g*. We conclude this section by showing that the above relationship between f and g is almost tight.

Lemma 10. *For any p, $f(\frac{1}{2}\log_\sigma n, p, \sigma) \leq g(n, p, \sigma)$.*

Proof. By definition, there exists a reconstruction algorithm $\mathcal{A}$ that recovers a random n character string with high probability using $g(n, p, 2)$ traces.

Given a set of traces of an unknown string x, it is easy to simulate an equal number of traces of the concatenated string $a|x|b$ for arbitrary strings a and b. Given successful recovery of $a|x|b$, we can of course extract x.

Let $B = \frac{1}{2}\log_\sigma n$. To recover $x \in [\sigma]^B$ from a set of traces, we first uniformly at random choose integers $c, d \in \{0, 1, \ldots, n/B - 1\}$ subject to $c + d = n/B - 1$. We then choose $a \in [\sigma]^{cB}$ and $b \in [\sigma]^{dB}$ uniformly at random. We simulate the traces of $a|x|b$, run $\mathcal{A}$ on the results, and extract x.

This succeeds whenever $\mathcal{A}$ successfully recovers $a|x|b$. Let μ be the uniformly random distribution on $[\sigma]^n$ and μ' be the distribution of $a|x|b$. Because $\mathcal{A}$ succeeds with high probability on μ, it suffices to show that the total variation distance between μ and μ' is polynomially small.

By thinking of the n character string as n/B blocks of length B, another way to draw from μ would be to (1) let k be drawn from $\mathcal{B}_{n/B,1/\sigma^B}$, the binomial random variable with n/B trials of probability $1/\sigma^B$; (2) set k random blocks to have value x; (3) set every other block independently to have a uniform value other than x. One can draw from μ' in the same way, but setting $k = 1 + \mathcal{B}_{n/B-1,1/\sigma^B}$ in the first step. Therefore the total variation distance between μ and μ' is at most the distance between $\mathcal{B}_{n/B,1/\sigma^B}$ and $1 + \mathcal{B}_{n/B-1,1/\sigma^B}$. This is $O(1/\sqrt{n/(B\sigma^B)}) < O(n^{-1/3})$, which is polynomially small. □

3.3 Lower Bound

In this section we prove the first super-logarithmic lower bound on the value of $g(n, p, 2)$ for constant p. To do this we introduce two specific binary strings of length $2r$ where $r = O(\log n)$:

1. $w \in \{0,1\}^{2r}$ is the all zero string expect for a single 1 at position r
2. $w' \in \{0,1\}^{2r}$ is the all zero string expect for a single 1 at position $r+1$

The proof relies on the fact that distinguishing w and w' with probability greater than $1-\delta$ requires $\Omega(r \log(1/\delta))$ traces (this will be implied by Corollary 1) and each of w and w' occur $n^{\Omega(1)}$ times in a random binary string of length n. The intuition is then that δ needs to be inversely polynomial in n otherwise one of the occurrences of w will be confused with an occurrence of w' (or vice versa). The following theorem formalizes this argument.

Theorem 1. $g(n, p, 2) = \Omega(\log^2 n)$ *for constant* $p > 0$.

Proof. Set the length of w and w' to be $B = c \log n$ for some small constant c, i.e., $r = (c \log n)/2$. By Corollary 1, if $m < c_2 \log^2 n$ for sufficiently small constant c_2, then the total variation distance between (m traces of w) and (m traces of w') is less than $1 - 1/\sqrt{n}$. Thus we can draw a set of m traces of a uniformly random choice between w or w' by choosing something independent of that choice with probability $1/\sqrt{n}$.

We partition our vector of length n into n/B blocks of length B. For a random bit vector and sufficiently small $c < 1/2$ we have with high probability that more than $\sqrt{n}$ blocks will equal one of w and w'. Therefore the algorithm must succeed with high probability on a random bit vector conditioned on having more than $\sqrt{n}$ blocks of value w or w'.

Now, the trace of a bit vector is just the concatenation of the trace of the component blocks. We could sample a set of m traces by first deciding which blocks are one of w or w', then choosing for each such block whether it is w or w', then taking the m traces. The resulting set of m traces is independent of block's choice between w and w' with probability $1/\sqrt{n}$; hence with at least $1 - (1 - 1/\sqrt{n})^{\sqrt{n}} > 1/2$ probability, the set of m traces will be independent of the choice of at least one of the $\sqrt{n}$ blocks of value w or w'. If this is true, the algorithm can give the correct output with probability at most $1/2$; hence the algorithm can give the correct output with probability at most $3/4$ overall. Therefore we need $m = \Omega(\log^2 n)$ for correct recovery with high probability. □

What remains is to prove Corollary 1. We make use of the Hellinger distance, a convenient measure of distance between distributions. For two discrete distribution $P = (p_1, p_2, p_3, \ldots)$ and $Q = (q_1, q_2, q_3, \ldots)$, the *squared Hellinger distance* between P and Q is defined as $\mathrm{H}^2(P, Q) = \frac{1}{2} \sum_i (\sqrt{p_i} - \sqrt{q_i})^2$.

Hellinger distance has two nice properties: first, squared Hellinger distance is subadditive over product measures, so the squared Hellinger distance between (m samples of P) and (m samples of Q) is at most $mH^2(P, Q)$; and second, if $H(P, Q) = o(1)$ then the total variation distance between P and Q is $o(1)$. Hence if $H(P, Q) \le \varepsilon$, then it requires $\Omega(1/\varepsilon)$ samples to distinguish P and Q.

Lemma 11. *For any deletion probability $p = \Omega(1)$, the squared Hellinger distance between the distribution of a trace of w and the distribution of a trace of w' is $O(1/r)$.*

Proof. The distribution of a trace of w is

$$\mathrm{Tr}(w) \sim \begin{cases} \underbrace{0\dots0}_{\mathcal{B}_{2r-1,1-p}} & \text{with probability } p \\ \underbrace{0\dots0}_{\mathcal{B}_{r-1,1-p}}\ 1\underbrace{0\dots0}_{\mathcal{B}_{r,1-p}} & \text{with probability } 1-p \end{cases}$$

while the distribution of a trace of w' is the same, except swapping $\mathcal{B}_{r-1,p}$ and $\mathcal{B}_{r,p}$. Hence the squared Hellinger distance between the two traces is

$$\begin{aligned} H^2(\mathrm{Tr}(w), \mathrm{Tr}(w')) &= (1-p)H^2\left((\mathcal{B}_{r-1,1-p}, \mathcal{B}_{r,1-p}), (\mathcal{B}_{r,1-p}, \mathcal{B}_{r-1,1-p})\right) \\ &\le 2(1-p)H^2(\mathcal{B}_{r-1,1-p}, \mathcal{B}_{r,1-p}) \le O(1/r)\ . \end{aligned}$$

□

Corollary 1. *Consider any $r > 1$, $\delta < 1$, and deletion probability $p = \Omega(1)$. For some small constant $c > 0$, the total variation distance between $m = c^2 r \log(1/\delta)$ traces of w and m traces of w' is at most $1-\delta$.*

Proof. Let $y_1, \dots, y_m$ be traces of w and $z_1, \dots, z_m$ be traces of w' for $m = c^2 r \log(1/\delta)$ and a sufficiently small constant c. We will show that the total variation distance between $(y_1, \dots, y_m)$ and $(z_1, \dots, z_m)$ is less than $1-\delta$.

We partition $[m]$ into k groups of size cr, for $k = c\log(1/\delta)$. Within each group, by subadditivity of squared Hellinger distance and appealing to Lemma 11, we have that

$$H^2((y_1, \dots, y_{cr}), (z_1, \dots, z_{cr})) \le crH^2(\mathrm{Tr}(w), \mathrm{Tr}(w')) = O(c) < 1/10$$

for sufficiently small c. Then the total variation distance between $(y_1, \dots, y_{cr})$ and $(z_1, \dots, z_{cr})$ is bounded by $2H((y_1, \dots, y_{cr}), (z_1, \dots, z_{cr})) \le 2/3$.

Hence we may sample $(y_1, \dots, y_{cr})$ and $(z_1, \dots, z_{cr})$ in such a way that the two distributions are identical with probability at least $1/3$. If we do this for all k groups, we have that $(y_1, \dots, y_m) \sim (z_1, \dots, z_m)$ with probability at least $1/3^k > 2\delta$ for sufficiently small constant c. □

4 Arbitrary String Reconstruction

In this last section, we consider the problem of reconstructing an arbitrary binary[2] string $x \in \{0,1\}^n$ from random subsequences of length $\Theta(\sqrt{n}\log n)$ or equivalently when the deletion probability of each bit is $1 - c\sqrt{\log n/n}$ for some constant c. We prove the following result.

[2] Recall that Lemma 1 shows that this result can be extended to the non-binary case.

Theorem 2. *$f(n,p,2) \leq e^{\sqrt{n}\,\mathrm{polylog}\,n}$ if $p \leq 1 - c\sqrt{\frac{\log n}{n}}$ for some constant $c > 0$.*

Our result uses the following combinatorial result by Scott [7]. For $i \in \{1,2,3,\ldots\}$, let n_i be the number of length i subsequences of x that end with a 1, i.e., $n_i = \sum_{j=1}^{n} x_j \binom{j-1}{i-1}$. Scott showed that if $k \geq (1+o(1))\sqrt{n \log n}$, then there exists a unique binary solution to the equation $Px^T = n^T$ where $n = (n_1, n_2, \ldots, n_k)$ and P is the $k \times n$ matrix with $P_{ij} = \binom{j-1}{i-1}$. The next theorem follows immediately.

Theorem 3 (Scott [7]). *$\{n_i\}_{i\in[k]}$ uniquely define x if $k \geq (1+o(1))\sqrt{n\log n}$.*

Therefore it is sufficient to determine each n_i. To do this is we pick a random subsequence of length i from each of the m traces and let m_i be the number of them that end with a 1. We then estimate n_i by $\tilde{n}_i = \frac{m_i}{m}\binom{n}{i}$. The next lemma shows that if m is sufficiently large then $n_i = \tilde{n}_i$ with high probability.

Lemma 12. *If $m \geq 2n^{2i}\log(2n)$ then $\Pr[n_i \neq \tilde{n}_i] \leq 1/n^2$.*

Proof. First note that $E[m_i/m] = n_i/\binom{n}{i}$ and that m_i is the sum of independent boolean trials. By applying the Chernoff bound,

$$\Pr[|\tilde{n}_i - n_i| \geq 1] = \Pr\left[\left|m_i - \frac{mn_i}{\binom{n}{i}}\right| \geq \frac{m}{\binom{n}{i}}\right] \leq 2\exp\left(-\frac{m}{3n_i\binom{n}{i}}\right) < 2\exp\left(\frac{-m}{n^{2i}}\right) .$$

Hence, $m > 2n^{2i}\log 2n$ ensures this probability is less than $1/n^2$. □

Therefore, by an application of the union bound $2n^{2(1+o(1))\sqrt{n\log n}}$ traces are sufficient to compute all the necessary n_i with high probability.

References

1. Batu, T., Kannan, S., Khanna, S., McGregor, A.: Reconstructing strings from random traces. In: ACM-SIAM Symposium on Discrete Algorithms, pp. 910–918 (2004)
2. Dudík, M., Schulman, L.J.: Reconstruction from subsequences. J. Comb. Theory, Ser. A 103(2), 337–348 (2003)
3. Holenstein, T., Mitzenmacher, M., Panigrahy, R., Wieder, U.: Trace reconstruction with constant deletion probability and related results. In: ACM-SIAM Symposium on Discrete Algorithms, pp. 389–398 (2008)
4. Kannan, S., McGregor, A.: More on reconstructing strings from random traces: Insertions and deletions. In: IEEE International Symposium on Information Theory, pp. 297–301 (2005)
5. Lember, J., Matzinger, H.: Standard deviation of the longest common subsequence. The Annals of Probability 37(3), 1192–1235 (2009)
6. Pollard, D.: Asymptopia (2000), `http://www.stat.yale.edu/pollard/`
7. Scott, A.D.: Reconstructing sequences. Discrete Mathematics 175(1-3), 231–238 (1997)
8. Viswanathan, K., Swaminathan, R.: Improved string reconstruction over insertion-deletion channels. In: ACM-SIAM Symposium on Discrete Algorithms, pp. 399–408 (2008)

PReaCH: A Fast Lightweight Reachability Index Using Pruning and Contraction Hierarchies

Florian Merz and Peter Sanders

Karlsruhe Institute of Technology, Karlsruhe, Germany
sanders@kit.edu, flomerz@gmail.com

Abstract. We develop the data structure PReaCH (for Pruned Reachability Contraction Hierarchies) which supports *reachability queries* in a directed graph. PReaCH adapts the contraction hierarchy speedup techniques for shortest path queries to the reachability setting. The resulting approach is surprisingly simple and guarantees linear space and near linear preprocessing time. Orthogonally to that, we improve existing pruning techniques for the search by gathering more information from a single DFS-traversal of the graph. In particular, we show that more classes of node numberings can be used to obtain strong pruning information.

1 Introduction

Many applications are modelled using graphs of some kind. One of the most fundamental questions one may ask about a graph is whether there is a path between two given nodes. For example, in a network of papers with links expressing citations, one might ask whether one paper is based on another paper in some, possibly indirect, way. Further applications include semantic networks / RDF graphs, XML, and applications in bioinformatics networks such as protein-protein interactions, metabolic networks, and gene regulatory networks.

Reachability queries can be answered in linear time using any kind of graph exploration, e.g., by breadth first search. However, for many applications this is too slow since a large number of queries has to be answered. Assuming that the graph changes rarely, we can afford to do some preprocessing, i.e., we compute a data structure I that helps to accelerate later queries. I can be viewed as an index data structure. When comparing such preprocessing approaches one faces a tradeoff between at least three criteria – preprocessing time, the space needed for the index, and the query time. For example, the above query by BFS approach has zero preprocessing costs yet requires linear query time. On the other hand, precomputing all possible answers needs space quadratic in the number of nodes but allows constant time queries. Clearly, compromises are relevant here.

Our starting point was to transfer the rapid recent progress on speedup techniques for route planning to reachability indices. We settled on *Contraction Hierarchies* (CHs) [4] because they are one of the most successful such techniques and because we found an adaptation to the reachability problem with surprisingly low preprocessing time. In Section 3 it turns out that *Reachability*

Contraction Hierarchies (RCHs) are much simpler than shortest path CHs and guarantee near linear preprocessing time and linear space consumption. During preprocessing, RCHs just repeatedly remove nodes with in-degree or out-degree zero. Edges incident to nodes with in-degree (out-degree) zero are marked for forward (backward) exploration. Queries are based on bidirectional graph exploration. Forward (backward) exploration only has to consider edges marked as forward (backward) during preprocessing. This can lead to dramatic speedups since the average branching factor of the graph exploration is halved. In contrast to RCHs, shortest path CHs need to insert additional *shortcut edges* during construction which may lead to quadratic space and cubic preprocessing for graphs that do not have very pronounced hierarchical properties.

An equally important ingredient of PReaCH is a suite of heuristics for pruning graph exploration during a query. We adopt and improve techniques from GRAIL [12]: *Topological levels* are essentially a compressed form of a topological ordering. When the level of a node v is larger or equal to the level of a node t then there certainly is no path from v to t. This information can be used for both forward and backward search and we can calculate different topological levels to allow further pruning.

Further rules are derived from a DFS-numbering of the nodes. For each node v, we identify two (*full*) ranges of DFS numbers of nodes that are reachable from v and three (*empty*) ranges of DFS numbers that are not reachable from v. During a query, full ranges can be used to stop the search completely while the empty ranges can prune the search as in topological levels. All these ranges can be derived from a single DFS traversal of the graph. In Section 4 we report on extensive experiments showing that PReaCH performs very well.

Related Work

There has been intensive previous work on reachability indices so that we can only shortly mention the most closely related results and refer to full paper [8] and the master thesis of Florian Merz [7] for more details which also contain proofs and further experiments. GRAIL [12] is the most successful previous work that uses graph traversal for pruning search. At the cost of possibly more expensive preprocessing, even faster queries are possible using hop-labelling techniques which precompute subsets of reachable nodes for every node that have the property that reachability testing can be reduced to intersecting these sets [2,1,11]. Among those, we compare with PPL [11] and TF [1]. These comparisons can also be used for "transitive" comparisons with very recent results that were published after our experiments ([7]) were performed [9,5].

2 Preliminaries

Consider a directed graph $G = (V, E)$. Let $n = |V|$ and $m = |E|$. A reachability query (s, t) asks whether there is a path from s to t – in symbols $s \rightarrow t$. Queries with result `true`/`false` are called positive/negative, respectively. We restrict our attention to directed acyclic graphs (DAGs) because the reachability problem can be reduced to DAGs by contracting strongly connected components.

The reachability problem can be solved by running breadth first search (BFS) from s: When t is found, the BFS is aborted and `true` is returned. Otherwise, `false` is returned once all nodes reachable from s are exhausted. Reachability can also be tested using *bidirectional BFS* where BFS steps from s alternate with BFS steps from t on the *backward graph* $\bar{G} = (V, \bar{E})$ with $(u, v) \in \bar{E}$ whenever $(v, u) \in E$. When any node is reached from both sides, `true` is returned. When either search space is exhausted, `false` is returned. In the worst case, bidirectional BFS does twice the amount of work as unidirectional BFS and we also have considerable space overhead because we have to store both outgoing edges and incoming edges. However, in many positive queries and in negative queries where the backward search space is much smaller than the forward search space, we can be dramatically faster than unidirectional BFS.

Depth first search (DFS) explores the nodes of a graph in a recursive fashion: There is an outer loop through the nodes looking for *roots* r for an exploration of the unexplored nodes reachable from r. The recursive function $\mathsf{explore}(u)$ inspects the outgoing edges (u, v) and recursively calls itself when v has not been explored yet. DFS defines a spanning forest of the graph – one tree for each considered root. Let $\phi(v) \in 1..n$ define the order in which the nodes are explored by DFS.[1] Note that ϕ is a *preordering* of the nodes in each tree of the DFS forest. In particular, the nodes in a subtree T of the forest rooted at v have numbers starting at $\phi(v)$ and ending at $\phi(v) + |T| - 1$. Other works use a *preordering* of the same tree also known as *finishing times*.

A third useful way to explore the nodes of a DAG are *topological levels* [12]. *Sources* of the graph, i.e., nodes v with out-degree zero have level $L(v) = 0$. The remaining nodes have level $L(v) = 1 + \max\{L(u) : (u, v) \in E\}$. Similarly, we can define *backward topological levels* based on *sink* nodes of the graph, i.e., nodes with in-degree zero.

3 The PReaCH Reachability Index

Reachability Contraction Hierarchies. In general, a *Reachability Contraction Hierarchy* (RCH) can be defined based on any numbering $order : V \to 1..n$. We successively *contract* nodes with increasing $order(v)$. Contracting node v means removing it from the graph and possibly inserting new edges such that the reachability relation in the new graph is the same as in the previous graph, i.e., if the graph contains a path $\langle u, v, w \rangle$ and after removing v, w would no longer be reachable from u, we have to insert a new *shortcut* edge (u, w). The query algorithm does not use the contracted versions of the graph but only the ordering and the shortcuts. This algorithm is based on the observation that in an RCH it suffices to look at "up-down" paths:

[1] Throughout this paper $a..b$ is a shorthand for $\{a, \ldots, b\}$.

Theorem 1. *In a graph G, $s \rightarrow t$ if and only if in an RCH G' obtained from G there is a path of the form $\langle s = u_1, \ldots, u_k = v = w_1, \ldots, w_\ell = t\rangle$ such that $order(u_1) < \cdots < order(u_k)$ and $order(w_1) > \cdots > order(w_\ell)$.*

Hence, a query can be implemented by a variant of the bidirectional BFS from Section 2 where both searches only look at adjacent nodes with larger *order*-value than the current node. A further modification is that now both search spaces must be exhausted in order to safely return `false`.

We now exploit that any DAG contains source nodes (with in-degree zero) and sink nodes (with out-degree zero). Contracting such a node v never requires the insertion of shortcuts because shortcuts always bridge paths of the form $\langle u, v, w\rangle$ but such paths cannot exist because v either lacks incoming or outcoming edges. Hence from now on we restrict the considered orderings such that they only contract source or sink nodes. This implies a huge simplification and acceleration of preprocessing compared to general RCHs. In particular, it suffices to use a static graph data structure and we get linear time preprocessing except perhaps for deciding which source or sink nodes should be contracted next. In contrast, a general CH might have to insert a quadratic number of shortcuts and indeed this is a significant limitation for computing shortest path CHs for graphs with a less pronounced hierarchy than road networks. In other words, RCHs have a much wider applicability and robustness than shortest path CHs.

There are still many ways to define an RCH ordering. We use the total degree (in-degree plus out-degree) of the nodes in the input graph for deciding in which order to contract source and sink nodes, i.e., nodes with smallest degree are contracted first. The ordering is computed on-line – a priority queue holds the source and sink nodes of the current graph using their degree in the input graph as priority. In each iteration, a lowest priority node v is contracted. The idea behind our priority function is to delay contraction of high degree nodes and thus to limit the branching factor of the resulting query search spaces. Since no shortcuts are needed, the contraction process degenerates to a kind of graph traversal – a contracted node is not really removed but just marked as contracted and the degrees of its neighbors are decremented. Those nodes which become sources or sinks are inserted into the priority queue. Note that only $2n$ priority queue operations are performed. In particular, no decrease key operations such as in Dijkstra's shortest path algorithm are needed. The running time of this algorithm is "near linear" in several senses. Using a comparison based priority queue, the running time becomes $\mathcal{O}(m + n \log n)$ with a quite small constant in front of the $n \log n$ term. In theory, we could use faster integer priority queues, for example van-Emde Boas trees [10,6,3] which would yield running time $\mathcal{O}(m + n \log\log n)$.

Considering the RCH query algorithm, we can partition the edge set into two disjoint sets: edges (u, v) with $order(u) < order(v)$ will only be considered by the forward search and the remaining ones only in the backward search. We organize the graph data structure in such a way that forward and backward search can directly access the edges they need. This implies that each edge is stored only

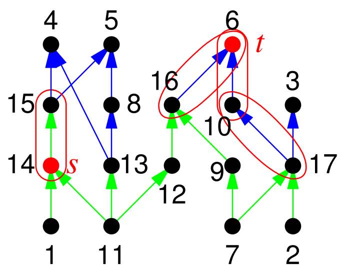


Fig. 1. Example RCH. Edges in the forward search space are light green and those in the backward search space are dark blue. The search spaces for a query from s to t are circled. Node labels specify the node ordering.

once while ordinary bidirectional search requires us to store each edge twice. Hence, RCHs save a considerable amount of space.

Figure 1 gives an example graph marking the forward and backward edges and the search spaces for an example s-t query.

Pruning Based on Topological Levels. The central observation in [12] on topological levels is very simple:

Lemma 1. $\forall u \neq v \in V : L(u) \geq L(v) \Rightarrow u \not\rightarrow v$.

We can apply Lemma 1 to v and t whenever we consider to explore a node v in the forward search. Analogously, we can apply it to s and v whenever we consider a node v in the backward search. Furthermore, we can apply the same reasoning to *backward topological levels.*

Pruning Based on Node Numbering. Consider a numbering $f : V \rightarrow 1..n$ of the nodes. One fairly general idea is to exploit the properties of f in order to store a compressed, approximate representation of the set of nodes reachable from each node. We aim for a rather rough approximation that can be computed in linear time and space for the entire graph and where we can test in constant time whether a node is in this approximated set. By applying this test everywhere during a query, we can nevertheless obtain a significant amount of improvement. More concretely, we will store a constant number of ranges of node numbers that are either empty or full. When the destination node t is in an empty range, the search does not have to continue there. When t is in a full range, the entire search can be stopped with a positive result. Note the asymmetry between these two cases. For positive queries, a positive test result has a much bigger potential for improvement. As for topological levels, we only describe the case for forward search. For backward search, the same reasoning is applied on the backward graph.

In most previous work this approach is only used for finishing time with respect to a DFS. For example, the conference version of GRAIL stores a single range of finishing times that must contain the target node. In a later version

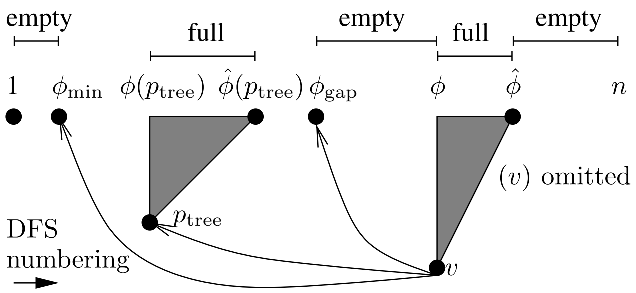


Fig. 2. Full and empty intervals derived from a DFS ordering ϕ. Shaded triangles indicate subtrees of the DFS tree.

GRAIL adds a positive range based on DFS numbering. GRAIL achieves additional pruning by working with several DFS searches. In [9] a method is explained to derive a tunable amount of information from DFS finishing times. We explain how to extract several interesting intervals from a single DFS numbering, obviating the need to compute and store finishing times.

Let $\phi(v)$ denote the DFS number of node v, and $\hat{\phi}(v)$ the largest DFS number of a node in the subtree of the DFS tree rooted at v. The properties of DFS ensure that the nodes in range(v):= $\phi(v)..\hat{\phi}(v)$ are all reachable from v (they form the subtree of the DFS tree which is rooted at v) and that no nodes with DFS number exceeding $\hat{\phi}(v)$ is reachable from v. Only this property of DFS numbering is already used in GRAIL [12]:

Lemma 2. $\forall v, t \in V : \phi(t) \in \text{range}(v) \Rightarrow v \to t$.

However, we also immediately get the following negative range:

Lemma 3. $\forall v, t \in V : \phi(t) > \hat{\phi}(v) \Rightarrow v \not\to t$.

Indeed, for any node w with $v \to w$, range(w) yields a range that we can use for positive pruning. We propose to actually compute and store the node outside range(v) with the *largest* such range (or $\perp$ if no such node exists). It turns out this can be done while computing the DFS numbering.

Lemma 4. *For any $v \in V$, consider the node $w = p_{\text{tree}}(v)$ with $v \to w$ and $w \notin \text{range}(v)$ which maximizes $|\text{range}(w)|$. When DFS on v finishes, w can be computed as w:= $\text{maxind}\,\{|\text{range}(p_{\text{tree}}(u))| : (v,u) \in E \wedge p_{\text{tree}}(u) \neq \perp\} \cup \{|\text{range}(u)| : (v,u) \in E \wedge \phi(u) < \phi(v)\}$.*

Similarly, we can obtain information that yields the empty interval to the left of any node reachable from v.

Lemma 5. *For any $v \in V$, let $\phi_{\min}(v)$ denote the smallest DFS number of a node reachable from v. When DFS on v finishes, $\phi_{\min}(v)$ can be computed as*

$$\phi_{\min}(v) := \min\{\phi_{\min}(w) : (v,w) \in E\} \cup \{\phi(v)\} \quad .$$

Finally, we compute the following empty range just to the left of v;

Lemma 6. *When DFS finishes for v, define*
$\phi_{\text{gap}}(v) := \max\left\{\hat{\phi}(w) : (v,w) \in E \wedge \phi(w) < \phi(v)\right\} \cup \{\phi_{\text{gap}}(w) : (v,w) \in E\}$.
Then $\phi_{\text{gap}}(v) + 1..\phi(v)$ is an empty range.

Figure 2 summarizes lemmas 1-6.

There are many ways to define a DFS ordering: We are free to choose the order in which we scan the nodes for starting recursive exploration and we can choose the order in which we inspect edges leaving a node being explored. Indeed, we could compute several DFS orderings and use all of them for pruning searches. Our current implementation uses only a single ordering thus minimizing preprocessing time and space. We do not have very strong heuristics for finding good orderings but there is one heuristics that seems to be useful: Make sure that most nodes are in a small number of trees because this leads to large positive intervals. It therefore makes sense to only uses sources of the graph as tree roots. In addition, we order the sources by the number of nodes reached from them during the search for topological levels.

4 Experiments

All experiments have been performed using a single core of an Intel Xeon X5550 running at 2.67GHz with 8MB Level 3 cache, 256kB Level 2 cache and 48GB of DDR3 RAM. The system ran Ubuntu 12.04.2 using a Linux kernel 3.5. The code has been compiled using gcc 4.8.2 with optimization level `O3`.

As far as sensible, we adopt instances and measurement conventions from previous work to improve comparability. In the tables, best values are bold. K and M are shorthands for 000 and 000 000, respectively.

4.1 Instances

We use graphs of five categories, largely adopted from [12,1]. Table 1 summarizes their properties. In addition we have added *Kronecker graphs* as a family of graphs that have become a standard in benchmarking graph algorithms and can be generated with arbitrary size. Besides the number of nodes and edges we give a number of further important parameters. We see that the *edge density* m/n is very small (even close to one) for many instances. We will see that the graphs with larger m/n can be much more difficult to handle. Another column gives the length d of the longest path which turns out to be fairly small for all instances except the Kronecker graphs. Finally, we indicate the fraction of positive queries in a random sample of 100 000 queries. It turns out that this fraction is close to zero for most instances. Since an application is not guaranteed to have the same small rate of positive queries, we explicitly use specially generated positive query instances in our experiments. Most experiments average times for 100 000 s-t reachability queries. We distinguish between positive and negative queries determined by picking a random s and then a random t (not) reachable from s.

Table 1. Instances used for our experiments. d is the maximal path length.

Dataset	Nodes	Edges	m/n	d	%pos
Kronecker					
kron12	2^{12}	117K	28.60	279	28
kron17	2^{17}	5069K	38.68	1354	19
kron22	2^{22}	184M	43.95	5821	13
large random					
rand100m5x	100M	500M	5	37	0.0
rand100m2x	100M	200M	2	21	0.0
rand10m10x	10M	100M	10	60	5.0
rand10m5x	10M	50M	5	35	0.0
rand10m2x	10M	20M	2	19	0.0
rand1m10x	1M	10M	10	59	10
rand1m5x	1M	5M	5	33	0.2
rand1m2x	1M	2M	2	19	0.0
large real					
citeseer	694K	312K	0.45	13	0.0
citeseerx	6 540K	15M	2.30	59	0.2
cit-patents	3 775K	17M	4.38	32	0.1
go-uniprot	6 968K	35M	4.99	21	0.0
uniprot22m	1 595K	1 595K	1.00	4	0.0
uniprot100m	16M	16M	1.00	9	0.0
uniprot150m	25M	25M	1.00	10	0.0
small real dense					
arxiv	6 000	67K	11.12	167	15
citeseer-sub	11K	44K	4.13	36	0.4
go	6 793	13K	1.97	16	0.2
pubmed	9 000	40K	4.45	19	0.7
yago	6 642	42K	6.38	13	0.2
small real sparse					
agrocyc	13K	14K	1.07	16	0.1
amaze	3 710	3 947	1.06	16	17
anthra	12K	13K	1.07	16	0.1
ecoo	13K	14K	1.08	22	0.1
human	39K	40K	1.01	18	0.0
kegg	3 617	4 395	1.22	26	20
mtbrv	9 602	10K	1.09	22	0.2
nasa	5 605	6 538	1.17	35	0.6
vchocyc	9 491	10K	1.09	21	0.1
xmark	6 080	7 051	1.16	38	1.4
stanford					
email-EuAll	231K	223K	0.97	7	5
p2p-Gnutella31	48K	55K	1.15	14	0.8
soc-LiveJournal1	971K	1 024K	1.05	24	21
web-Google	372K	518K	1.39	34	15
wiki-Talk	2 282K	2 312K	1.01	8	0.8

Table 2. Performance of PReaCH compared to GRAIL [12] with five DFS numberings, TF [1], and PPL [11]. The numbers for PReaCH are average execution time in ns for queries and total construction time in ms. The other numbers are slowdown (or space overhead for columns "ind") relative to PReaCH.

	PReaCH			GRAIL5			TF				PPL			
	+	−	constr	+	−	constr	+	−	constr	ind	+	−	constr	ind
cit-Patents	14K	1 519	**9 464**	3.88	3.71	2.31	0.06	0.35	22.94	16.70	**0.06**	**0.20**	26.20	2.30
citeseer	**40.07**	35.25	**305**	24.32	3.11	7.28	2.71	**0.26**	2.66	0.38	3.05	2.81	4.76	**0.19**
citeseerx	488	**131**	**7 127**	15.80	3.00	3.02	2.04	1.42	11.86	3.72	**0.54**	1.30	5.63	**0.28**
go-uniprot	450	54.59	**6 097**	3.41	1.93	5.78	2.50	**0.91**	10.37	**0.38**	**0.56**	3.71	4.30	0.48
uniprotenc-22m	**36.03**	**32.55**	**402**	22.25	2.76	13.38	1.79	1.15	5.53	0.41	3.15	3.62	6.89	**0.18**
uniprotenc-100m	**53.88**	**78.47**	**6 072**	21.11	2.30	12.09	2.60	1.21	6.56	0.41	3.56	2.55	4.74	**0.18**
uniprotenc-150m	**60.15**	**103**	**10K**	22.32	2.22	12.03	3.00	1.12	5.52	0.41	3.40	2.16	4.53	**0.18**
arxiv	408	181	**6.08**	5.52	3.02	2.82	0.65	2.38	1 128	23.93	**0.14**	**0.26**	5.83	**0.52**
citeseer-sub	126	60.06	**6.83**	5.15	2.04	3.23	**0.54**	**0.66**	15.91	1.30	0.59	0.77	5.27	**0.35**
go	49.85	30.59	**2.55**	5.50	1.90	3.81	**0.94**	**0.94**	15.29	0.67	1.21	1.22	5.67	**0.44**
pubmed	201	51.24	**5.34**	5.75	2.62	3.31	0.46	**0.79**	52.86	1.45	**0.38**	0.88	4.76	**0.40**
yago	**44.58**	19.39	**3.45**	9.76	2.66	4.06	2.10	**0.87**	16.91	0.68	1.15	1.96	4.76	**0.36**
agrocyc	**12.69**	**7.71**	**2.10**	18.82	4.99	7.63	17.10	2.09	15.64	0.76	2.19	3.82	8.05	**0.24**
amaze	**13.02**	**8.41**	**0.75**	19.66	3.27	6.87	1.31	1.13	12.69	0.41	1.95	2.45	7.05	**0.24**
anthra	**12.68**	**7.16**	**2.05**	18.30	5.23	7.70	13.14	2.18	22.89	0.71	2.16	3.95	8.24	**0.24**
ecoo	**13.80**	**7.24**	**2.07**	17.53	5.44	7.96	2.73	2.26	16.84	0.76	2.00	4.00	8.11	**0.24**
human	**13.34**	**7.65**	**6.82**	17.90	8.47	10.41	16.81	1.93	11.47	0.53	2.18	5.54	7.81	**0.24**
kegg	**13.17**	**8.95**	**0.76**	20.18	3.40	6.48	1.42	1.18	13.53	0.41	2.08	2.54	14.24	**0.24**
mtbrv	**12.64**	**7.24**	**1.64**	19.11	4.92	7.27	1.37	1.90	12.77	0.47	2.14	3.74	8.01	**0.24**
nasa	**23.01**	**21.07**	**1.38**	13.30	2.27	4.83	1.85	1.18	18.93	0.65	2.09	1.42	7.92	**0.41**
vchocyc	**13.24**	**7.73**	**1.57**	17.78	4.60	7.50	2.56	2.04	28.45	0.82	2.02	3.55	8.18	**0.24**
xmark	**41.37**	**21.71**	**1.31**	5.91	2.88	5.65	1.88	1.18	20.69	0.65	1.12	1.43	10.53	**0.41**
email-EuAll	**31.33**	28.41	**72.56**	20.89	3.17	9.01	2.95	**0.55**	2.10	0.35	2.92	2.56	5.56	**0.18**
p2p-Gnutella31	**34.42**	**10.05**	**14.80**	10.95	6.86	7.28	1.54	1.68	3.11	0.41	1.51	4.43	4.90	**0.24**
soc-LiveJournal1	**50.39**	**30.02**	**323**	15.06	3.45	8.88	2.18	1.29	2.00	0.41	2.73	2.82	5.39	**0.24**
web-Google	**51.81**	**42.06**	**211**	14.21	3.83	5.25	1.82	1.11	2.23	0.41	2.29	1.88	4.09	**0.24**
wiki-Talk	**61.53**	**34.85**	**925**	17.13	2.66	8.91	2.56	1.18	1.79	0.41	2.44	3.49	4.36	**0.18**

There are so many reachability indices that it is impossible to compare with all of them directly. We have therefore focused on three recent techniques that fare very well in comparison with others and seem to constitute the state of the art. GRAIL [12] is particularly interesting since, similar to PReaCH, it has guaranteed linear preprocessing time and space. The authors recommend a variant using data from five DFS traversals which we call GRAIL5 and which we use in most comparisons. Incidentally, GRAIL5 also uses about the same amount of space than PReaCH so that this additionally simplifies the analysis. In some experiments we also look at the more light weight variant with a single DFS and call it GRAIL.

TF [1] is a more recent labelling technique based on "folding" paths. It is particularly, useful for graphs with small value of d in Table 1. PPL [11] is also a labelling technique and very often achieves quite small labels, excellent query time and good preprocessing time. In particular, it can profit from long paths in the graph.

Table 2 summarizes the results giving absolute values for PReaCH and slowdown factors relative to PReaCH for the other heuristics. PReaCH dominates GRAIL5 with respect to both query time and preprocessing time (while using about the same space). The advantage is particularly pronounced for positive queries where the improvement is often more than an order of magnitude. The significant advantage of PReaCH over GRAIL5 with respect to preprocessing is surprising since both technique traverse the graph five times (for PReaCH: RCHs, forward/backward topological levels, forward/backward DFS) and since PReaCH has additional overhead for a priority queue. The reason may be implementation details or deteriorating cache efficiency due to the randomization of DFS used in GRAIL. Neither TF nor PPL dominate GRAIL5 because they often need much higher preprocessing time.

Comparing PReaCH with TF and PPL is more complicated. With respect to construction time, PReaCH is always the best algorithm – sometimes by orders of magnitude. For the most difficult instances TF ran out of memory. For `random10M10x` PPL was stopped after 9h. With respect to query time, PReaCH achieves the best values for 43 out of 72 cases while TF ranks second with 16 best values closely followed by PPL with 13 best values. Basically, for easy instances, PReaCH slightly outperforms the labeling techniques. For difficult instances which the labeling techniques can handle at all, they significantly outperform PReaCH but at the cost of very high preprocessing time (and space in case of TF). With respect to space consumption, PPL is the best in almost all cases. However, for dense random graphs and for `cit-Patents` much more space is needed than for PReaCH, i.e., PReaCH can still score for being more predictable with respect to space consumption.

The recent result from [9] also guarantees efficient preprocessing and seems to achieve better query performance than PReaCH in many cases. However, the means to achieve this seem sufficiently orthogonal from ours that we believe that a combination of both approaches could be the overall best. The results from

[5] seem to outperform PPL in many cases. However, as with all hop-labelling techniques, preprocessing becomes very expensive for some difficult instances.

5 Conclusion

Reachability indices that guarantee linear space and near linear proprocessing time have become more and more powerful in the recent years. The best implementations combine several techniques for pruning the search space. PReaCH makes several contributions to this family of techniques: RCHs, using several sets of topological levels, general techniques for deriving full and empty intervals from node numberings, and particular instantiations of these ideas for DFS numberings. PReaCH and related techniques can also be adapted to actually compute paths for positive queries: RCH queries explicitly generate a path that, due to the absence of shortcuts in our implementation, need not even be unpacked. Pruning rules involving empty intervals and topological levels are no problem since they are not applied on the path. The only problem are pruning rules involving full intervals. Since our full intervals are fully inferred from a neighbor, we can either store a pointer to that neighbor or traverse the neighbors until the interval is found.

The constant factor in the space consumption of PReaCH might be too expensive in some applications. However, we can derive very space efficient reachability indices from PReaCH also. For example using a variant of RCHs, we only need to store the graph itself plus a few bits telling where to split the edges stored with a node between forward and backward search space. Except for the Kronecker graphs, all the instances given in Table 1 need at most eight bits for representing a topological level. Hence, with two bytes per node one can support pruning with topological levels additionally.

Future Work

There are many opportunities for further improvements. Staying close to PReaCH, we can try to trade time for space by using several DFS orderings simultaneously for pruning the search. We could also derive more information from a single DFS by adapting the methods developed in [9] to DFS numbers or other node orderings. We can trade query time for preprocessing time by performing several DFS searches (with random tie breaking) and only use the best one for actually storing the index. Judging what "the best" is could be based on performing queries for a random sample. The same idea can be applied to the contraction hierarchy. More interesting would be more clever heuristics to find good DFS-orderings and RCH-orderings. For example, for DFS we could better approximate the tree sizes by actually performing complete DFS explorations from all sources before deciding what the first tree is going to be. For RCH-ordering, the simple, static priority function based on degree seems only like a very first attempt. For example, CHs for route planning [4] use an estimate of the (unpruned) search space size as an important term in the priority function.

Besides compressing topological levels as mentioned above, we can also compress the data derived from DFS traversal. It suffices to store $\hat{\phi}(v) - \phi(v)$ rather than $\hat{\phi}(v)$ which will be small for most nodes. We could for example represent only values between 0 and 254 directly using 255 as an escape value indicating that the true value can be found in a small hash table. We have already mentioned that at the cost of an additional indirection, the values $\phi(p_{\text{tree}}(v))$ and $\hat{\phi}(p_{\text{tree}}(v))$ do not need to be stored when we store $p_{\text{tree}}(v)$. All these measures combined would reduce the space requirement by about one third.

When scaling to even larger graphs, we would like to parallelize preprocessing. A certain degree of 'easy' parallelization is available by computing the RCH, topological levels and DFS based information independently. Moreover we can perform multiple DFS – using all its information or only the one that works best. For graphs with not too many topological levels, finding these levels can be done by peeling them off in parallel. A similar strategy works for CHs (see also [4]) for route planning. There is also intensive work on finding SCCs in parallel. For finding full intervals we could also use any preorder numbering of any spanning forest of the graph, e.g., based on BFS. Only for the empty intervals we would need a replacement for DFS with similarly useful properties.

References

1. Cheng, J., Huang, S., Wu, H., Fu, A.W.-C.: TF-Label: A topological-folding labeling scheme for reachability querying in a large graph (2013)
2. Cohen, E., Halperin, E., Kaplan, H., Zwick, U.: Reachability and distance queries via 2-hop labels. SIAM Journal on Computing 32(5), 1338–1355 (2003)
3. Dementiev, R., Kettner, L., Mehnert, J., Sanders, P.: Engineering a sorted list data structure for 32 bit keys. In: 6th Workshop on Algorithm Engineering & Experiments, New Orleans (2004)
4. Geisberger, R., Sanders, P., Schultes, D., Vetter, C.: Exact routing in large road networks using contraction hierarchies. Transp. Science 46(3), 388–404 (2012)
5. Jin, R., Wang, G.: Simple, fast, and scalable reachability oracle. Proc. VLDB Endow. 6(14), 1978–1989 (2013)
6. Mehlhorn, K., Näher, S.: Bounded ordered dictionaries in $O(\log \log N)$ time and $O(n)$ space. Information Processing Letters 35(4), 183–189 (1990)
7. Merz, F.: Engineering an efficient reachability algorithm for directed graphs. Master's thesis, Kalsruhe Institute of Technology (2013)
8. Merz, F., Sanders, P.: PReaCH: A fast lightweight reachability index using pruning and contraction hierarchies. Technical Report 1404.4465, arxiv (2014)
9. Seufert, S., Anand, A., Bedathur, S., Weikum, G.: Ferrari: Flexible and efficient reachability range assignment for graph indexing. In: 29th International Conference on Data Engineering (ICDE), pp. 1009–1020. IEEE (2013)
10. van Emde Boas, P.: Preserving order in a forest in less than logarithmic time. Information Processing Letters 6(3), 80–82 (1977)
11. Yano, Y., Akiba, T., Iwata, Y., Yoshida, Y.: Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths. In: 22nd Conf. on Information and Knowledge Management (CIKM), pp. 1601–1606. ACM (2013)
12. Yıldırım, H., Chaoji, V., Zaki, M.J.: GRAIL: a scalable index for reachability queries in very large graphs. VLDB Journal 21(4), 509–534 (2012)

Polynomial-Time Approximation Schemes for Circle Packing Problems[⋆]

Flávio K. Miyazawa[1], Lehilton L.C. Pedrosa[1], Rafael C.S. Schouery[1], Maxim Sviridenko[2], and Yoshiko Wakabayashi[3]

[1] Institute of Computing, University of Campinas, Brazil
[2] Yahoo! Labs, New York, NY
[3] Institute of Mathematics and Statistics, University of São Paulo, Brazil

Abstract. We consider the problem of packing a set of circles into a minimum number of unit square bins. We give an asymptotic approximation scheme (APTAS) when we have resource augmentation in one dimension, that is, we may use bins of height $1+\gamma$, for some small $\gamma > 0$. As a corollary, we also obtain an APTAS for the circle strip packing problem, whose objective is to pack a set of circles into a strip of unit width and minimum height. These are the first approximation schemes for these problems. Our algorithm is based on novel ideas of iteratively separating small and large items, and may be extended to more general packing problems. For example, we also obtain APTAS's for the corresponding problems of packing d-dimensional spheres under the L_p-norm.

1 Introduction

In the *circle bin packing problem*, we are given a list of circles $\mathcal{C} = \{C_1, \ldots, C_n\}$, where circle C_i has radius $r_i \leq 1/2$, for $1 \leq i \leq n$, and an unlimited number of identical square bins of unit side. A packing is a non-overlapping arrangement of circles (disks) into a set of bins, such that every circle is fully contained in a bin. The objective is to find a packing of $\mathcal{C}$ into a minimum number of bins. In the *circle strip packing problem* (or *the circular open dimension packing problem*), the set of circles $\mathcal{C}$ must be packed in a strip of unit width and unbounded height, and the objective is to obtain a packing of minimum height.

There are several results in the literature for packing problems involving circles, that are tackled using different methods, such as continuous and non-linear systems, and discrete methods [6]. Demaine, Fekete, and Lang [11] proved that it is NP-hard to decide whether a set of circles can be packed into a unit square, or into an equilateral triangle. Therefore, the circle bin packing problem and the circle strip packing problem are also NP-hard.

We are interested in polynomial-time approximation algorithms for the circle bin packing and circle strip packing problems. As it is usual for packing problems,

⋆ This work was partially supported by CNPq (grants 303987/2010-3, 306860/2010-4, 477203/2012-4, and 477692/2012-5), FAPESP (grants 2010/20710-4, 2013/02434-8, 2013/03447-6, and 2013/21744-8), and Project MaClinC of NUMEC at USP, Brazil.

the quality measure that we look for is the *asymptotic performance ratio*. Given an algorithm $\mathcal{A}$, and a problem instance I, we denote by $\mathcal{A}(I)$ the value of the solution produced by $\mathcal{A}$, and by $\mathrm{OPT}(I)$ the value of an optimum solution for I. A family of algorithms $\{\mathcal{A}_\varepsilon\}$ is an asymptotic polynomial-time approximation scheme (APTAS) if, for every instance I, and every constant $\varepsilon > 0$, algorithm $\mathcal{A}_\varepsilon$ generates a solution with cost $\mathcal{A}(I) \leq (1+\varepsilon)\mathrm{OPT}(I) + O(1)$.

Our results and techniques. In this paper, we give APTAS's for both the circle bin packing, and the circle strip packing. We consider the bin packing problem with resource augmentation in one dimension, that is, we may use bins of unit width and height $1+\gamma$, for some arbitrarily small $\gamma > 0$. To the best of our knowledge, this is the first work to give approximation guarantees for these problems. Our algorithm uses ideas that have appeared in the literature in several and new interesting ways. As usual in the packing of rectangular items, our algorithm distinguishes between "large" and "small" items. However, this distinction is dynamic, that is, an item can be considered small in one iteration, but large in a later one. There are two main novel ideas that differ from the approaches for rectangle packing, and that are needed for the circle packing. First, instead of packing large items using combinatorial brute-force algorithms, we reduce the packing of large circles to the problem of solving a semi-algebraic system, that can be solved with the aid of standard quantifier elimination algorithms. Second, to pack small items, we cut the free space of previous packings in smaller bins, and use the same algorithm for large items, recursively. We believe that our new algorithm can serve as insight for other packing problems, and, indeed, we present some possible generalizations.

Related works. In the literature of approximation algorithms, the majority of the works consider the packing of simple items into larger recipients, such as rectangular bins and strips. Most of the works that give approximation guarantees are interested in rectangular items or d-dimensional box. The packing problems involving circles are mainly considered through heuristics, or numerical methods, and, to our knowledge, there is no approximation algorithm for the circle bin packing or the circle strip packing problems. On the practical side, packing problems have numerous applications, such as packaging of boxes in containers, or cutting of material. An application of circular packing is, for example, obtaining a maximal coverage of radio towers in a geographical region [20].

The problem of finding the densest packing of equal circles into a square has been largely investigated using many different optimization methods. For an extensive book on this problem, and corresponding approaches, see [20]. The case of non-equal circles is considered in [13], that uses heuristics, such as genetic algorithm, to pack circles in a rectangular container. The circle strip packing has been considered using many approaches, such as branch-and-bound, meta-heuristics, etc. For a broad list of algorithms for the circle strip packing, and related circle packing problems, see [15] and references therein.

For the problem of packing rectangles into rectangular bins, there is a sequence of algorithms [8,7,1,4] that leads to the recent 1.405-approximation by Bansal and Khan [4]. For the bin packing of d-dimensional cubes, Kohayakawa *et al.* [17] showed an asymptotic ratio of $2 - (2/3)^d$, later improved to an APTAS by Bansal *et al.* [2]. For a survey on bin packing, see [9]. The best bound for the rectangle strip packing problem is an APTAS by Kenyon and Rémila [16]. For the 3-dimensional case, the first specialized algorithm for cubes has an asymptotic ratio of 2.361 [19], and the best result is an APTAS due to Bansal *et al.* [3].

The remainder of this paper is organized as follows. In Section 2, we discuss how to decide whether a set of n circles can all be packed in a rectangular bin using algebraic quantifier elimination. In Section 3, we give approximation algorithms for the case of "large" circles. In Section 4, we present an APTAS for the circle bin packing problem, and for the circle strip packing problem. Also, we comment on how the algorithm can be extended to more general problems, such as packing of spheres. In Section 5, we conclude with some final remarks.

2 Packing through Algebraic Quantifier Elimination

In this section, we consider the following *circle packing decision problem.* We are given numbers $h, w \in \mathbb{Q}_+$, and a set of n circles $\mathcal{C} = \{C_1, \ldots, C_n\}$, where circle C_i has radius r_i, with $2r_i \leq \min\{w, h\}$, $1 \leq i \leq n$. The objective is to decide whether the circles can be packed in a bin of size $w \times h$ (of width w and height h). In the case of a positive answer, a realization of the packing should also be returned. More precisely, for each circle C_i, $1 \leq i \leq n$, we want to find a point $(x_i, y_i) \in \mathbb{R}^2_+$ that represents the center of C_i in a rectangle whose bottom-left and top-right corners correspond to points $(0, 0)$ and (w, h), respectively.

The circle packing decision problem can be equivalently formulated as deciding whether there are real numbers $x_i, y_i \in \mathbb{R}_+$, $1 \leq i \leq n$, that satisfy

$$
\begin{aligned}
(x_i - x_j)^2 + (y_i - y_j)^2 &\geq (r_i + r_j)^2 && \text{for } 1 \leq i < j \leq n, && (1)\\
r_i \leq x_i &\leq w - r_i && \text{for } 1 \leq i \leq n, \text{ and} && (2)\\
r_i \leq y_i &\leq h - r_i && \text{for } 1 \leq i \leq n. && (3)
\end{aligned}
$$

The set of constraints (1) guarantees that no two circles intersect, and the sets of constraints (2) and (3) ensure that each circle has to be packed entirely in the bin that expands from the origin $(0, 0)$ to the point (w, h).

We observe that the set of solutions that satisfy (1)-(3) is a semi-algebraic set in the field of the real numbers. Thus, the circle packing decision problem corresponds to deciding whether this semi-algebraic set is empty. We also can rewrite the constraints in (1)-(3) as $f_i(x_1, y_1, ..., x_n, y_n) \geq 0$, for $1 \leq i \leq s$, where s is the total number of constraints, and $f_i \in \mathbb{Q}[x_1, y_1, ..., x_n, y_n]$ is a polynomial with rational coefficients. Then, the circle packing problem is equivalent to deciding the truth of the formula

$$(\exists x_1)(\exists y_1) \ldots (\exists x_n)(\exists y_n) \bigwedge_{i=0}^{s} f_i(x_1, y_1, ..., x_n, y_n) \geq 0. \qquad (4)$$

We can use any algorithm for the more general quantifier elimination problem to decide this formula. There are several algorithms for this problem, such as the algorithm of Tarski-Seidenberg Theorem [21], that is not elementary recursive, or the Cylindrical Decomposition Algorithm [10], that is doubly exponential in the number of variables. Since the formula corresponding to the circle packing problem contains only one block of variables (of existential quantifiers), we can use faster algorithms for the corresponding algebraic existential problem, such as the algorithms of Grigor'ev and Vorobjov [14], or of Basu, Pollack, and Roy [5].

Sampling points of the solution. Any of the algorithms above receiving formula (4) as input will return "true" if, and only if, there is some arrangement of circles $\mathcal{C}$ in a bin of size $w \times h$. When the answer is "true", we are also interested in a realization of such packing. The algorithms in [14,5] are based on critical points, that is, they also return a finite set of points that meets every semi-algebraic connected component of the semi-algebraic set. Thus, a realization of the packing can be obtained by choosing one of such points (that is a point of a connected component where all polynomials f_i, $1 \leq i \leq s$, are nonnegative).

Typically, a sample point is represented by a tuple $(f(x), g_0(x), \ldots, g_k(x))$ of $k+2$ univariate polynomials with coefficients in $\mathbb{Q}$, where k is the number of variables, and the value of the ith variable is $g_i(x)/g_0(x)$ evaluated at a real root of $f(x) = 0$. Since a point in a semi-algebraic set could potentially be irrational, we use the algorithm of Grigor'ev and Vorobjov [14], for which we have $g_0(x) = 1$, and thus an approximate rational solution of arbitrary precision can be readily obtained. In particular, the algorithm given in [14] implies the following result.

Theorem 1. *Let $f_1, ..., f_s \in \mathbb{Q}[x_1, y_1, ..., x_n, y_n]$ be polynomials with coefficients of bit-size at most m, and maximum degree 2. There is an algorithm that decides the truth of formula (4), with running time $m^{O(1)} s^{O(n^2)}$. In the case of affirmative answer, then the algorithm also returns polynomials $f, g_1, h_1, \ldots, g_n, h_n \in \mathbb{Q}[x]$ with coefficients of bit-size at most $m^{O(1)} s^{O(n)}$, and maximum degree $s^{O(n)}$, such that for a root x of $f(x) = 0$, the attribution $x_1 = g_1(x), y_1 = h_1(x), ..., x_n = g_n(x), y_n = h_n(x)$ is a realization of (4). Moreover, for any rational $\alpha > 0$, we can obtain $x'_1, y'_1, \ldots, x'_n, y'_n \in \mathbb{Q}$, such that $|x'_i - x_i| \leq \alpha$ and $|y'_i - y_i| \leq \alpha$, $1 \leq i \leq n$, with running time $(\log(1/\alpha)m)^{O(1)} s^{O(n^2)}$.*

3 Approximate Bin Packing of Large Circles

In this section, we consider the special case of circle bin packing when the minimum radius of a circle is at least a constant. For this case, the maximum number of circles that fit in a bin is constant, so we can use the algorithm of Theorem 1 to decide whether a given set of circles can be packed in a bin in constant time. Since Theorem 1 only gives us rational solutions that are close to real packings, we start with the next definition to deal with approximate circle bin packings.

Definition 1. *Let w, h be positive numbers, and $\mathcal{C} = \{C_1, \ldots, C_n\}$ be a set of circles, such that each circle C_i, $1 \leq i \leq n$, has radius r_i, with $2r_i \leq \min\{w, h\}$.*

For a number $\varepsilon \geq 0$, we say that a set of points (x_i, y_i), $1 \leq i \leq n$, is an ε-packing of $\mathcal{C}$ into a rectangular bin of width w and height h, if the following hold.

$$\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \geq r_i + r_j - \varepsilon \geq 0 \quad \textit{for } 1 \leq i < j \leq n, \tag{5}$$
$$r_i - \varepsilon \leq x_i \leq w - r_i + \varepsilon \quad \textit{for } 1 \leq i \leq n, \textit{ and} \tag{6}$$
$$r_i - \varepsilon \leq y_i \leq h - r_i + \varepsilon \quad \textit{for } 1 \leq i \leq n. \tag{7}$$

We adopt the following strategy to fix intersections of an approximate bin packing: (a) first we shift the x-coordinate of all circles that intersect the left or right border until they are fully contained in the bin, (b) then we iteratively lift each circle in order of the y-coordinate by an appropriate distance so that it does not intersect circles considered previously. This leads to the next theorem.

Theorem 2. *Given a set of circles $\mathcal{C} = \{C_1, \ldots, C_n\}$, such that each circle C_i, $1 \leq i \leq n$, has radius $r_i \in \mathbb{Q}_+$, and $2r_i \leq \min\{w, h\}$, and a corresponding εh-packing of $\mathcal{C}$ in a bin of width w and height h for some $\varepsilon \in (0, 1]$, we can find a packing of $\mathcal{C}$ in a bin of width w and height $(1 + n\sqrt{6\varepsilon})h$ in linear time.*

Definition 2. *Let $w, h \in \mathbb{Q}_+$ be positive numbers, and let $\mathcal{C} = \{C_1, \ldots, C_n\}$ be a set of circles, such that each circle C_i, $1 \leq i \leq n$, has radius $r_i \in \mathbb{Q}_+$, and $2r_i \leq \min\{w, h\}$. We denote by $\mathrm{OPT}_{w\times h}(\mathcal{C})$ the minimum number of rectangular bins of width w and height h that are necessary to pack $\mathcal{C}$.*

Now, we obtain an approximation algorithm for the bin packing of large circles, that is, assuming that the radius of each circle is greater than a given constant. In this case, the maximum number of circles that fit in a bin is at most a constant, M, so we can partition the set of circles into a small number of groups with approximate sizes, and enumerate all patterns of groups with no more than M circles. Then, we may apply the algorithm of Theorem 1 to list which patterns correspond to feasible packings, and use integer programming in fixed dimension to find out how many bins of each pattern are necessary to cover all circles. Since a similar approach has already been used for bin packing [12], and the following theorem uses analogous arguments, the proof is omitted.

Theorem 3. *Let $w, h \in \mathbb{Q}_+$ be positive numbers, and let $\mathcal{C} = \{C_1, \ldots, C_n\}$ be a set of circles, such that each circle C_i, $1 \leq i \leq n$, has radius $r_i \in \mathbb{Q}_+$, and $2r_i \leq \min\{w, h\}$. Assume that $\min_{1\leq i\leq n} r_i \geq \delta$, for some constant δ. For any constants $\varepsilon, \gamma \in (0, 1]$, there is a polynomial-time algorithm that packs $\mathcal{C}$ into at most $(1 + \varepsilon)\mathrm{OPT}_{w\times h}(\mathcal{C})$ rectangular bins of width w and height $(1 + \gamma)h$.*

4 An Asymptotic PTAS for Circle Bin Packing

In this section, we consider the bin packing problem of circles of any size. The main idea works as follows. First, we will use the algorithm from Section 3 and obtain a packing of “large” circles into bins of the original size. Then, we consider bins with a small fraction of the original size, and solve the problem of packing the "small" circles in such bins recursively. To obtain a solution of the original problem, we place each obtained small bin into the free space of the packing obtained for large circles. The key idea is that, if the sizes of the small bins are much smaller than the large circles, then the waste of space in the packing of the large circles is proportional to a fraction of the area of the large circles. Moreover, if the size of such small bin is also much larger than the small circles, then restricting the packing of small circles to small bins does not increase much the cost of a solution.

4.1 The Algorithm

In the following, if B is a circle or rectangle, then we denote by $Area(B)$ the area of B. Also, if D is a set, then $Area(D) = \sum_{B \in D} Area(B)$. We give first a formal description in Algorithm 1; an informal description is given thereafter.

Algorithm 1. *Circle bin packing algorithm*

Consider the parameters r and γ, such that r is a positive integer multiple of 3, and γ a number in $(0, 1]$. The algorithm receives a set of circles $\mathcal{C} = \{C_1, \ldots, C_n\}$, and numbers w, h, such that $w \leq h$, and hr/w is an integer. Moreover, each circle C_i, $1 \leq i \leq n$, has radius $r_i \in \mathbb{Q}_+$, with $2r_i \leq w$. The algorithm returns a packing of $\mathcal{C}$ into a set of bins of width w and height $(1 + \gamma)h$.

```
1. Let $\varepsilon = 1/r$;
2. For every integer $i \geq 0$, define $G_i = \{C_j \in \mathcal{C} : \varepsilon^{2i} w \geq 2r_j > \varepsilon^{2(i+1)} w\}$;
3. For each $0 \leq j < r$, define $H_j = \{C \in G_i : i \equiv j \pmod{r}\}$;
4. Find an integer $t$ such that $Area(H_t) \leq \varepsilon Area(\mathcal{C})$;
5. Place each circle of $H_t$ into its bounding box, and pack them in separate
   bins of width $w$ and height $(1 + \gamma)h$ using NFDH strategy [18];
6. For every integer $j \geq 0$, define $S_j = \{C \in G_i : t+(j-1)r+1 \leq i \leq t+jr-1\}$;
7. Define $w_0 = w$, $h_0 = h$ and $w_j = h_j = \varepsilon^{2(t+(j-1)r)+1} w$ for every $j \geq 1$;
8. Let $F_0 = \emptyset$;
9. For every $j \geq 0$:
   (a) Use the algorithm of Theorem 3 to obtain a packing of circles $S_j$ into
       bins of width $w_j$ and height $(1 + \gamma)h_j$. Let $P_j$ be the set of such bins;
   (b) Let $A_j$ be a set of $\max\{|P_j| - |F_j|, 0\}$ new empty bins of width $w_j$ and
       height $(1 + \gamma)h_j$;
   (c) Place each bin of $P_j$ over one distinct bin of $F_j \cup A_j$;
   (d) Set $F_{j+1} = \emptyset$, and $U_j = \emptyset$;    ($U_j$ is used only in the analysis)
   (e) For each bin $B$ of $F_j \cup A_j$:
       – Let $V$ be the set of bins corresponding to the cells of the grid with
         cells of width $w_{j+1}$ and height $(1 + \gamma)h_{j+1}$ over $B$;
       – Add to $F_{j+1}$ all bins in $V$ that do not intersect any circle of $S_j$.
       – Add to $U_j$ all bins in $V$ that intersect a circle of $S_j$.
   (f) If all circles are packed, go to step 10.
10. Place the bins $A_0, A_1, \ldots$ into the minimum number of bins of width $w$ and
    height $(1 + \gamma)h$.
```

$$\begin{array}{|l r r l l|c|}
 & & & & & H_t\colon \\
S_0\colon & & G_0, & \dots & G_{t-1} & G_t \\
S_1\colon & G_{t+1}, & G_{t+2}, & \dots & G_{t+r-1} & G_{t+r} \\
S_2\colon & G_{t+r+1}, & G_{t+r+2}, & \dots & G_{t+2r-1} & G_{t+2r} \\
S_j\colon & G_{t+(j-1)r+1}, & G_{t+jr+2}, & \dots & G_{t+jr-1} & G_{t+jr} \\
 & & & \vdots & & \vdots
\end{array}$$

Fig. 1. Partitioning of the set of circles

Notice that the assumption that hr/w is integer is without loss of generality, since we could round up the height h_0 otherwise. The algorithm partitions the set of circles into sets $H_t, S_0, S_1, \dots$, as in Figure 1. For each j, we consider the subproblem of packing the circles of S_j into bins of size $w_j \times h_j$. The circles in S_j are large when compared to bins of size $w_j \times h_j$, and so we can use the algorithm of Theorem 3. Notice that the bins considered in the next iteration have size $w_{j+1} \times h_{j+1}$, and are much smaller than the circles of S_j. Also, since the circles in H_t are considered separately, each remaining unpacked circle (of $S_{j+1}, S_{j+2}, \dots$) fits in a bin of size $w_{j+1} \times h_{j+1}$.

In each iteration $j \geq 0$, the algorithm keeps a set F_j of free bins of size $w_j \times (1+\gamma)h_j$ obtained from previous iterations. We obtain a packing of circles of S_j into a set of bins P_j. Then, we place such bins over the free space of F_j, or over additional bins A_j, if necessary. The set of sub-bins of $F_j \cup A_j$ of size $w_{j+1} \times (1+\gamma)h_{j+1}$ that intersect circles of S_j are included in the set U_j, and the remaining sub-bins are saved in F_{j+1} for the next iteration.

4.2 Analysis

Consider a bin B of width w_B and height h_B. Given w and h, we say that B *respects* $w \times h$ if either $w_B = w$, and $h_B = h$, or $w_B = h_B = \varepsilon^{2(t+(j-1)r)+1}w$ for some $j \geq 1$. Similarly, if D is a set of bins, then we say that D respects $w \times h$ if every $B \in D$ respects $w \times h$. In what follows, we assume that we have run Algorithm 1, giving as input positive numbers $w, h \in \mathbb{Q}_+$, with $w \leq h$, a set of circles $\mathcal{C} = \{C_1, \dots, C_n\}$, such that each circle C_i, $1 \leq i \leq n$, has radius $r_i \in \mathbb{Q}_+$, and $2r_i \leq \min\{w, h\}$, and parameters $r \in \mathbb{Z}_+$, $\gamma \in (0, 1]$.

Definition 3. *Let $j \geq 0$. If B' is a bin that respects $w \times h$, then we denote by $\mathrm{Gr}'_j(B')$ the set of bins in the grid with cells of width w_j and height h_j over B', and, if D' is a set of bins that respects $w \times h$, then $\mathrm{Gr}'_j(D') = \cup_{B' \in D'} \mathrm{Gr}'_j(B')$.*

Definition 4. *If B is a rectangle or circle, then define $\mathrm{N}'(B) = Area(B)/(wh)$. Also, if D is a set of rectangles or circles, then define $\mathrm{N}'(D) = \sum_{B \in D} \mathrm{N}'(B)$.*

If a set of bins D respects $w \times h$, then bins of D can be easily combined into bins of size $w \times h$ using almost the same area. Thus, $\mathrm{N}'(D)$ is an estimate on the number of bins of size $w \times h$ needed to pack D, as in the following remark.

Remark 1. If there is a packing of $\mathcal{C}$ into a set of bins D' that respects $w \times h$, then there is a packing of $\mathcal{C}$ in $\lceil \mathrm{N}'(D') \rceil$ bins of width w and height h.

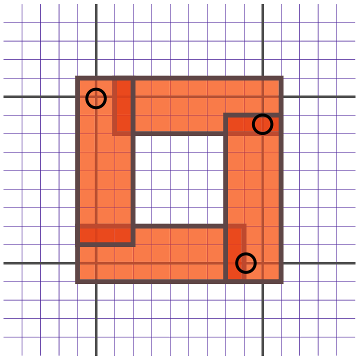
For some j, the area of U_j is not fully used, since there might be cells of U_j that partially intersect circles of S_j. This waste is bounded by the next lemma.

Lemma 1. *Let $C \in S_j$ be a circle packed in a bin B that respects $w \times h$, and let $D \subseteq \mathrm{Gr}'_{j+1}(B)$ be the subset of bins in the grid that intersect circle C, but are not contained in C. Then $\mathrm{N}'(D) \leq 16\varepsilon \mathrm{N}'(C)$.*

In the following, we show that requiring that each set of circles S_j be packed into grid bins of size $w_j \times h_j$ does not increase much the solution cost. This fact is central to the algorithm, since it allows iteratively packing sets S_j's.

To show these properties, we will transform an optimal packing *Opt* of $\mathcal{C}$ into a packing D with the desired properties. The idea is moving circles of S_j that intersect lines of the grid of size $w_j \times h_j$ to free bins that respect the grid. The next algorithm keeps the invariant that, at the start of iteration $j \geq 1$, the set R_j contains free space to pack all such intersecting circles. Steps (3a)-(3c) move intersecting circles to bins of R_j, and steps (3d)-(3f) make sure that there are enough free bins in R_{j+1} respecting the grid of size $w_{j+1} \times h_{j+1}$.

```
1. Let $R_1$ be a set of $12\varepsilon(wh)/(w_1h_1)|Opt|$ new bins of width $w_1$ and height $h_1$;
2. Let $D_0 = Opt \cup R_1$;
3. For each $j \geq 1$:
   (a) Let $L_j = \emptyset$;
   (b) For each bin $B \in \mathrm{Gr}'_j(Opt)$:
       i. Let $W$ be the set of circles in $S_j$ that intersect the boundary of $B$;
       ii. Let $V$ be a set of 4 new bins (2 of width $3\varepsilon w_j$ and height $h_j$, and 2
           of width $w_j$ and height $3\varepsilon h_j$) placed over the boundary of $B$, so that
           each circle in $W$ is contained in one bin of $V$ (see Figure 2);
       iii. For each cell $B' \in \mathrm{Gr}'_{j+1}(V)$, let $\phi(B')$ be the cell of $\mathrm{Gr}'_{j+1}(Opt)$
            under $B'$;
       iv. Remove each circle of $W$ from the packing $D_j$ and pack it over one
           bin of $V$ preserving the arrangement;
       v. Add $V$ to $L_j$;
   (c) Make groups of $r/3$ bins (of equal sizes) of $L_j$ forming a new bin of
       width $w_j$ and height $h_j$, and place this bin over one distinct bin of $R_j$;
   (d) Let $R_{j+1} = \emptyset$;
   (e) Let $N_j = \emptyset$;
   (f) For each bin $B \in \mathrm{Gr}'_{j+1}(L_j)$, consider the cases:
       i. If $B$ does not intersect any circle of $S_j$, then add $B$ to $R_{j+1}$;
       ii. If $B$ is contained in some circle of $S_j$, then add $\phi(B)$ to $R_{j+1}$;
       iii. If $B$ intersects, but is not contained in a circle of $S_j$, then create a
            new bin of width $w_{j+1}$ and height $h_{j+1}$ and add to $R_{j+1}$, and to $N_j$;
   (g) Let $D_j = D_{j-1} \cup N_j$;
   (h) If $\mathcal{C} \setminus H_t = S_0 \cup \cdots \cup S_j$, make $D = D_j$, and stop.
```



Each circle of W has diameter at most εw_j. To rearrange the rectangles into bins of size $w_j \times w_j$ (of R_j), we use one side of length w_j. To ensure every circle is in a rectangle, the other side has length $3\varepsilon w_j$ (see lower circle).

Fig. 2. Removing circles that intersect lines

First, we note that the procedure is well defined. It is enough to check that R_j has free space to pack bins of L_j. Indeed, it is not hard to obtain the following.

Claim 1. For every $j \geq 1$, $Area(L_j) = Area(R_j) = Area(R_1)$.

Now, it will be shown that the algorithm produces a modified solution with the desired properties.

Claim 2. At the end of iteration $j \geq 0$, the following statements hold:

1. D_j is a packing of $\mathcal{C}$;
2. for each $\ell = 0, \ldots, j$, there is a packing of S_ℓ into a set $P'_\ell \subseteq \mathrm{Gr}'_\ell(D_j)$ of bins of width w_ℓ and height h_ℓ;
3. the bins in $R_{j+1} \subseteq \mathrm{Gr}'_{j+1}(D_j)$ do not intersect any circle of $\mathcal{C}$.

Proof. By induction on j. For $j = 0$, the statements are clear. So let $j \geq 1$, and assume that the statements are true for $j - 1$.

Statement 1: Clearly, L_j is a packing of the circles that were removed from the original packing D_{j-1}. Since r is a multiple of 3 and by Claim 1, the step (3c) is well defined, and thus we can place rectangles of L_j over bins of R_j. After step (3c), we have a bin packing of $\mathcal{C}$, since, by hypothesis, the set R_j did not intersect any circle at the beginning of iteration j. This shows statement 1.

Statement 2: Since, at the end of iteration, each circle of S_j that intersected a line of the grid $\mathrm{Gr}'_j(D_{j-1})$ is completely contained in a bin of $R_j \subseteq \mathrm{Gr}'_j(D_j)$, we obtain statement 2.

Statement 3: If step (3(f)i) or step (3(f)iii) is executed, then we add a free bin to R_{j+1}. Thus, we only need to argue that whenever step (3(f)ii) is executed, the bin $\phi(B)$ does not intersect any circle. Let C be the circle that contains B, so that at the beginning of the iteration, $\phi(B)$ was contained in C. Since C was moved to L_j, $\phi(B)$ does not intersect any circle when step (3(f)ii) is executed. This completes the proof. □

Finally, we may calculate the cost of the modified solution D.

Claim 3. $\mathrm{N}'(D) \leq (1 + 28\varepsilon)\mathrm{OPT}_{w\times h}(\mathcal{C})$.

Proof. Let m be the number of iterations of the algorithm that modifies *Opt*. Notice that D is the disjoint union of *Opt*, R_1, N_1, ..., N_m. We get

$$\begin{aligned}\mathrm{N}'(D) &= \mathrm{N}'(\mathit{Opt}) + \mathrm{N}'(R_1) + \textstyle\sum_{j=1}^{m} \mathrm{N}'(N_j)\\ &\leq \mathrm{N}'(\mathit{Opt}) + 12\varepsilon(wh)|\mathit{Opt}|/(wh) + \textstyle\sum_{j=1}^{m} 16\varepsilon \mathrm{N}'(S_j)\\ &\leq \mathrm{N}'(\mathit{Opt}) + 12\varepsilon\mathrm{N}'(\mathit{Opt}) + 16\varepsilon\mathrm{N}'(\mathcal{C})\\ &= (1 + 28\varepsilon)\mathrm{OPT}_{w\times h}(\mathcal{C}),\end{aligned}$$

where the first inequality comes from Lemma 1. □

By combining the last two claims, one can obtain the following lemma.

Lemma 2. *There is a packing of* $\mathcal{C} \setminus H_t$ *into a set of bins* D *that respects* $w \times h$ *with* $\mathrm{N}'(D) \leq (1+28\varepsilon)\mathrm{OPT}_{w\times h}(\mathcal{C})$*, such that for every* $j \geq 0$*, there is a packing of* S_j *into a set of bins* $P'_j \subseteq \mathrm{Gr}'_j(D)$.

In addition to requiring that each set of circles S_j be packed into bins of the grid with cells of width w_j and height h_j, we also require that the bins used to pack $S_{j+1}, S_{j+2}, \ldots$ do not intersect circles of $S_1, \ldots, S_{j-1}$. Again, this restriction only increases the cost of a solution by a small fraction of the optimal value, as shown by the next lemma. The proof is similar to that of the previous lemma.

Lemma 3. *There is a packing of* $\mathcal{C} \setminus H_t$ *into a set of bins* D *that respects* $w \times h$ *with* $\mathrm{N}'(D) \leq (1+44\varepsilon)\mathrm{OPT}_{w\times h}(\mathcal{C})$*, such that for every* $j \geq 0$*, there is a packing of* S_j *into a set of bins* $P'_j \subseteq \mathrm{Gr}'_j(D)$*. Moreover, if* $B \in P'_j$*, then* B *does not intersect any circle* $C_i \in S_\ell$ *for* $\ell < j$.

Now, we obtain our main theorem, that states that Algorithm 1 is an APTAS for the circle bin packing problem. The proof is rather long, and thus it is left to the full version. The major technique is comparing the solution generated by the algorithm to an optimal solution modified by Lemmas 2 and 3. The key observation is that, for each subset of circles S_j, $j \geq 0$, the packing P_j generated by the algorithm is not much larger than the packing P'_j of the modified optimal solution. Also, although we use Theorem 3 to pack circles of any size, the radius of the smallest circle is large when compared to the considered bins, so the running time is polynomial.

Theorem 4. *Let* $w, h \in \mathbb{Q}_+$ *be positive numbers, and let* $\mathcal{C} = \{C_1, \ldots, C_n\}$ *be a set of circles, such that each circle* C_i*,* $1 \leq i \leq n$*, has radius* $r_i \in \mathbb{Q}_+$*, and* $2r_i \leq \min\{w, h\}$*. For any given constants* $\varepsilon, \gamma \in (0, 1]$*, we can obtain in polynomial time a packing of* $\mathcal{C}$ *into at most* $(1 + \varepsilon)\mathrm{OPT}_{w\times h}(\mathcal{C}) + 2$ *rectangular bins of width* w *and height* $(1 + \gamma)h$.

It is simple to extend Theorem 4 to the circle strip-packing.

Theorem 5. *Let $\mathcal{C} = \{C_1, \ldots, C_n\}$ be a set of circles, such that each circle C_i, $1 \leq i \leq n$, has radius $r_i \in \mathbb{Q}_+$, and $2r_i \leq 1$. For any given constant $\varepsilon \in (0, 1]$, we can obtain in polynomial time a packing of $\mathcal{C}$ in a strip of unit width and height $(1+\varepsilon)\mathrm{OPTS}(\mathcal{C}) + O(1/\varepsilon)$, where $\mathrm{OPTS}(\mathcal{C})$ is the height of the minimum packing of $\mathcal{C}$ in a strip of unit width.*

Generalizations. Algorithm 1 does not depend on the items being circles. The crucial assumptions are that the packing of "large" items can be (approximately) solved, and that the space wasted by sub-bins that partially intersect an item is bounded by a fraction of its area. For example, Algorithm 1 can be generalized to deal with more general items, such as d-dimensional spheres under the L_p-norm, for $p \geq 1$. The packing of large spheres is done by algebraic quantifier algorithms, and bounding the waste is similar to the case of circles. It is also possible to pack items in augmented bins of different shapes. For example, Algorithm 1 may be slightly modified, so that it can deal with packing of spheres into spheres.

5 Final Remarks

We presented the first approximation algorithms for the circle bin packing problem using augmented bins, and the circle strip packing problem. We obtained asymptotic approximation schemes for circle packings exploring novel ideas, such as iteratively distinguishing large and small items, and carefully using the free space left after packing large items. We believe that our algorithm can lead to further results for related problems, and we have already presented some possible generalizations. Also, our use of algebraic quantifier elimination algorithms exemplifies how results from algebra can be successfully used in the context of optimization. Using these algorithms helped us to avoid discretization algorithms, whose running time would depend exponentially on resource augmentation parameter $1/\gamma$, and allowed the packing of more general items, such as L_p-norm spheres, in a simple and concise framework.

We note that, although the quantifier elimination algorithms we used give a precise representation of a packing in a non-augmented bin, the returned solution may possibly contain irrational coordinates. To provide solutions with rational numbers, we use approximate coordinates with arbitrary precision. This is the only reason why we used augmented bins, and thus resource augmentation can be avoided in a more general computational model. We left open the question to determine if it is always possible to obtain a rational solution to the problem of packing a set of circles of rational radii in a non-augmented bin of rational dimensions.

Acknowledgement. We would like to thank Frank Vallentin for providing us with insights and references on the cylindric algebraic decomposition and other algebraic notions.

References

1. Bansal, N., Caprara, A., Sviridenko, M.: A New Approximation Method for Set Covering Problems, with Applications to Multidimensional Bin Packing. SIAM J. on Computing 39(4), 1256–1278 (2010)
2. Bansal, N., Correa, J.R., Kenyon, C., Sviridenko, M.: Bin Packing in Multiple Dimensions: Inapproximability Results and Approximation Schemes. Mathematics of Operations Research 31(1), 31–49 (2006)
3. Bansal, N., Han, X., Iwama, K., Sviridenko, M., Zhang, G.: A harmonic algorithm for the 3D strip packing problem. SIAM J. on Computing 42(2), 579–592 (2013)
4. Bansal, N., Khan, A.: Improved Approximation Algorithm for Two-Dimensional Bin Packing. In: SODA 2014, pp. 13–25 (2014)
5. Basu, S., Pollack, R., Roy, M.F.: On the Combinatorial and Algebraic Complexity of Quantifier Elimination. J. ACM 43(6), 1002–1045 (1996)
6. Birgin, E.G., Gentil, J.M.: New and improved results for packing identical unitary radius circles within triangles, rectangles and strips. Computers & Op. Research 37(7), 1318–1327 (2010)
7. Caprara, A.: Packing d-dimensional bins in d stages. Mathematics of Operations Research 33(1), 203–215 (2008)
8. Chung, F., Garey, M., Johnson, D.: On Packing Two-Dimensional Bins. SIAM Journal on Algebraic Discrete Methods 3(1), 66–76 (1982)
9. Coffman, J.E.G., Csirik, J., Galambos, G., Martello, S., Vigo, D.: Bin Packing Approximation Algorithms: Survey and Classification. In: Pardalos, P.M., Du, D.Z., Graham, R.L. (eds.) Handbook of Combinatorial Optimization, pp. 455–531. Springer, New York (2013)
10. Collins, G.E.: Quantifier elimination for real closed fields by cylindrical algebraic decompostion. In: Brakhage, H. (ed.) GI-Fachtagung 1975. LNCS, vol. 33, pp. 134–183. Springer, Heidelberg (1975)
11. Demaine, E.D., Fekete, S.P., Lang, R.J.: Circle Packing for Origami Design Is Hard. In: Proc. of the 5th Inter. Conference on Origami in Science, pp. 609–626 (2010)
12. Fernandez de la Vega, W., Lueker, G.S.: Bin packing can be solved within $1 + \varepsilon$ in linear time. Combinatorica 1(4), 349–355 (1981)
13. George, J.A., George, J.M., Lamar, B.W.: Packing different-sized circles into a rectangular container. European J. of Operational Research 84(3), 693–712 (1995)
14. Grigor'ev, D.Y., Vorobjov Jr., N.N.: Solving systems of polynomial inequalities in subexponential time. Journal of Symbolic Computation 5(1-2), 37–64 (1988)
15. Hifi, M., M'Hallah, R.: A literature review on circle and sphere packing problems: Models and methodologies. Advances in Operations Research 2009, 1–22 (2009)
16. Kenyon, C., Rémila, E.: A Near-Optimal Solution to a Two-Dimensional Cutting Stock Problem. Mathematics of Operations Research 25(4), 645–656 (2000)
17. Kohayakawa, Y., Miyazawa, F., Raghavan, P., Wakabayashi, Y.: Multidimensional Cube Packing. Algorithmica 40(3), 173–187 (2004)
18. Meir, A., Moser, L.: On packing of squares and cubes. Journal of Combinatorial Theory 5(2), 126–134 (1968)
19. Miyazawa, F., Wakabayashi, Y.: Approximation algorithms for the orthogonal z-oriented three-dimensional packing problem. J. on Comp. 29(3), 1008–1029 (2000)
20. Szabó, P.G., Markót, M.C., Csendes, T., Specht, E., Casado, L., García, I.: New-Approaches-to-Circle-Packing-in-a-Square-Book. Springer (2007)
21. Tarski, A.: A decision method for elementary algebra and geometry. University of California Press (1951)

Document Retrieval on Repetitive Collections*

Gonzalo Navarro[1], Simon J. Puglisi[2], and Jouni Sirén[1]

[1] Center for Biotechnology and Bioengineering, Department of Computer Science, University of Chile, Chile
{gnavarro,jsiren}@dcc.uchile.cl

[2] Department of Computer Science, University of Helsinki, Finland
puglisi@cs.helsinki.fi

Abstract. Document retrieval aims at finding the most important documents where a pattern appears in a collection of strings. Traditional pattern-matching techniques yield brute-force document retrieval solutions, which has motivated the research on tailored indexes that offer near-optimal performance. However, an experimental study establishing which alternatives are actually better than brute force, and which perform best depending on the collection characteristics, has not been carried out. In this paper we address this shortcoming by exploring the relationship between the nature of the underlying collection and the performance of current methods. Via extensive experiments we show that established solutions are often beaten in practice by brute-force alternatives. We also design new methods that offer superior time/space trade-offs, particularly on repetitive collections.

1 Introduction

The *pattern matching* problem, that is, preprocessing a text collection so as to efficiently find the occurrences of patterns, is a classic in Computer Science. The optimal suffix tree solution [18] dates back to 1973. Suffix arrays [10] are a simpler, near-optimal alternative. Surprisingly, the natural variant of the problem called *document listing*, where one wants to find simply in which texts of the collection (called the *documents*) a pattern appears, was not solved optimally until almost 30 years later [11]. Another natural variant, the *top-k documents* problem, where one wants to find the k *most relevant* documents where a pattern appears, for some notion of relevance, had to wait for other 10 years [6,15].

A general problem with the above indexes is their size. While for moderate-sized collections (of total length n) their linear space (i.e., $O(n)$ words, or $O(n \log n)$ bits) is affordable, the constant factors multiplying the linear term make the solutions prohibitive on large collections. In this aspect, again, the pattern matching problem has had some years of advantage. The first compressed

* This work is funded in part by: Fondecyt Project 1-140796 (first author); Basal Funds FB0001, Conicyt, Chile (first and third authors); the Jenny and Antti Wihuri Foundation, Finland (third author); and by the Academy of Finland through grants 258308 and 250345 (CoECGR) (second author).

suffix arrays (CSAs) appeared in the year 2000 (see [14]) and since then have evolved until achieving, for example, asymptotically optimal space in terms of high-order empirical entropy and time slightly over the optimal. There has been much research on similarly compressed data structures for document retrieval (see [13]). Since the foundational paper of Hon et al. [6], results have come close to using just $o(n)$ bits on top of the space of a CSA and almost optimal time.

Compressing in terms of statistical entropy is adequate in many cases, but it fails in various types of modern collections. *Repetitive* document collections, where most documents are similar, in whole or piecewise, to other documents, naturally arise in fields like computational biology, versioned collections, periodic publications, and software repositories (see [12]). The successful pattern matching indices for these types of collections use grammar or Lempel-Ziv compression, which exploit repetitiveness [2,3]. There are only a couple of document listing indices for repetitive collections [4,1], and none for the top-k problem.

Although several document retrieval solutions have been implemented and tested in practice [16,7,3,4], no systematic practical study of how these indexes perform, depending on the collection characteristics, has been carried out.

A first issue is to determine under what circumstances specific document listing solutions actually beat brute-force solutions based on pattern matching. In many applications documents are relatively small (a few kilobytes) and therefore are unlikely to contain many occurrences of a given pattern. This means that in practice the number of pattern occurrences (*occ*) may not be much larger than the number of documents the pattern occurs in (*docc*), and therefore pattern matching-based solutions may be competitive.

A second issue that has been generally neglected in the literature is that collections have different kinds of repetitiveness, depending on the application. For example, one might have a set of distinct documents, each one internally repetitive piecewise, or a set of documents that are in whole similar to each other. The repetition structure can be linear (each document similar to a previous one) as in versioned collections, or even tree-like, or completely unstructured, as in some biological collections. It is not clear how current document retrieval solutions behave depending on the type of repetitiveness.

In this paper we carry out a thorough experimental study of the performance of most existing solutions to document listing and top-k document retrieval, considering various types of real-life and synthetic collections. We show that brute-force solutions are indeed competitive in several practical scenarios, and that some existing solutions perform well only on some kinds of repetitive collections, whereas others present a more stable behavior. We also design new and superior alternatives for top-k document retrieval.

2 Background

Let $T[1,n]$ be a concatenation of a collection of d documents. We assume each document ends with a special character \$ that is lexicographically smaller than any other character of the alphabet. The *suffix array* of the collection is an array

$\mathsf{SA}[1,n]$ of pointers to the suffixes of T in lexicographic order. The *document array* $\mathsf{DA}[1,n]$ is a related array, where $\mathsf{DA}[i]$ is the identifier of the document containing $T[\mathsf{SA}[i]]$. Let $B[1,n]$ be a bitvector, where $B[i]=1$ if a new document begins at $T[i]$. We can map text positions to document identifiers by: $\mathsf{DA}[i] = \mathsf{rank}_1(B,\mathsf{SA}[i])$, where $\mathsf{rank}_1(B,j)$ is the number of 1-bits in prefix $B[1,j]$.

In this paper, we consider indexes supporting four kinds of queries: 1) $\mathsf{find}(P)$ returns the range $[sp,ep]$, where the suffixes in $\mathsf{SA}[sp,ep]$ start with pattern P; 2) $\mathsf{locate}(sp,ep)$ returns $\mathsf{SA}[sp,ep]$; 3) $\mathsf{list}(P)$ returns the identifiers of documents containing pattern P; and 4) $\mathsf{topk}(P,k)$ returns the identifiers of the k documents containing the most occurrences of P. CSAs support the first two queries. $\mathsf{find}()$ is relatively fast, while $\mathsf{locate}()$ can be much slower. The main time/space tradeoff in a CSA, the *suffix array sample period*, affects the performance of $\mathsf{locate}()$ queries. Larger sample periods result in slower and smaller indexes.

Muthukrishnan's document listing algorithm [11] uses an array $\mathsf{C}[1,n]$, where $\mathsf{C}[i]$ points to the last occurrence of $\mathsf{DA}[i]$ in $\mathsf{DA}[1,i-1]$. Given a query range $[sp,ep]$, $\mathsf{DA}[i]$ is the first occurrence of that document in the range iff $\mathsf{C}[i] < sp$. A *range minimum query* (RMQ) structure over C is used to find the position i with the smallest value in $\mathsf{C}[sp,ep]$. If $\mathsf{C}[i] < sp$, the algorithm reports $\mathsf{DA}[i]$, and continues recursively in $[sp,i-1]$ and $[i+1,ep]$. Sadakane [17] improved the space usage with two observations: 1) if the recursion is done in preorder from left to right, $\mathsf{C}[i] \geq sp$ iff document $\mathsf{DA}[i]$ has been seen before, so array C is not needed; and 2) array DA can also be removed by using $\mathsf{locate}()$ and B instead.

Let $\mathsf{lcp}(S,T)$ be the length of the *longest common prefix* of sequences S and T. The LCP array of $T[1,n]$ is an array $\mathsf{LCP}[1,n]$, where $\mathsf{LCP}[i] = \mathsf{lcp}(T[\mathsf{SA}[i-1],n], T[\mathsf{SA}[i],n])$. We obtain the *interleaved LCP array* $\mathsf{ILCP}[1,n]$ by building separate LCP arrays for each of the documents, and interleaving them according to the document array. As $\mathsf{ILCP}[i] < |P|$ iff position i contains the first occurrence of $\mathsf{DA}[i]$ in $\mathsf{DA}[sp,ep]$, we can use Sadakane's algorithm with RMQs over ILCP instead of C [4]. If the collection is repetitive, we can get a smaller and faster index by building the RMQ only over the run heads in ILCP.

3 Algorithms

In this section we review *practical* methods for document listing and top-k document retrieval. For a more detailed review see, e.g., [13].

Brute force. These algorithms sort the document identifiers in range $\mathsf{DA}[sp,ep]$ and report each of them once. Brute-D stores DA in $n\log d$ bits, while Brute-L retrieves the range $\mathsf{SA}[sp,ep]$ with $\mathsf{locate}()$ and uses bitvector B to convert it to $\mathsf{DA}[sp,ep]$. Both algorithms can also be used for top-k retrieval by computing the frequency of each document identifier and then sorting by frequency.

Sadakane. This is a family of algorithms based on Sadakane's improvements [17] to Muthukrishnan's algorithm [11]. Sada-C-L is the original algorithm of Sadakane, while Sada-C-D uses an explicit document array instead of retrieving the document identifiers with $\mathsf{locate}()$. Sada-I-L and Sada-I-D are otherwise the same, respectively, except that they build the RMQ over ILCP [4] instead of C.

Wavelet tree. A *wavelet tree* over a sequence can be used to quickly list the distinct values in any substring, and hence a wavelet tree over DA can be a good solution for many document retrieval problems. The best known implementation of wavelet tree-based document listing [16] can use plain, entropy-compressed [14], and grammar-compressed [8] bitvectors in the wavelet tree. Our WT uses a heuristic similar to the original WT-alpha [16], multiplying the size of the plain bitvector by 0.81 and the size of the entropy-compressed bitvector by 0.9, before choosing the smallest one for each level of the tree.

For top-k retrieval, WT combines the wavelet tree used in document listing with a space-efficient implementation [16] of the top-k trees of Hon et al. [6]. Out of the alternatives investigated by Navarro and Valenzuela [16], we tested the greedy algorithm, LIGHT and XLIGHT encodings for the trees, and sampling parameter $g' = 400$. In the results, we use the slightly smaller XLIGHT.

Precomputed document listing. PDL [4] builds a sparse suffix tree for the collection, and stores the answers to document listing queries for the nodes of the tree. For long query ranges, we compute the answer to the list() query as a union of a small number of stored answer sets. The answers for short ranges are computed by using Brute-L. PDL-BC is the original version, using a web graph compressor [5] to compress the sets. If a subset S' of document identifiers occurs in many of the stored sets, the compressor creates a grammar rule $X \to S'$, and replaces the subset with X. We chose block size $b = 256$ and storing factor $\beta = 16$ as good general-purpose parameter values. We extend PDL in Section 4.

Grammar-Based. Grammar [1] is an adaptation of a grammar-compressed self-index [2] for document listing. Conceptually similar to PDL, Grammar uses Re-Pair [8] to parse the collection. For each nonterminal symbol in the grammar, it stores the set of document identifiers whose encoding contains the symbol. A second round of Re-Pair is used to compress the sets. Unlike most of the other solutions, Grammar is an independent index and needs no CSA to operate.

Lempel-Ziv. LZ [3] is an adaptation of self-indexes based on LZ78 parsing for document listing. Like Grammar, LZ does not need a CSA.

Grid. Grid [7] is a faster but usually larger alternative to WT. It can answer top-k queries quickly if the pattern occurs at least twice in each reported document. If documents with just one occurrence are needed, Grid uses a variant of Sada-C-L to find them. We also tried to use Grid for document listing, but the performance was not good, as it usually reverted to Sada-C-L.

4 Extending Precomputed Document Listing

In addition to PDL-BC, we implemented another variant of precomputed document listing [4] that uses Re-Pair [8] instead of the biclique-based compressor.

In the new variant, named PDL-RP, each stored set is represented as an increasing sequence of document identifiers. The stored sets are compressed with Re-Pair, but otherwise PDL-RP is the same as PDL-BC. Due to the multi-level grammar generated by Re-Pair, decompressing the sets can be slower in PDL-RP than in PDL-BC. Another drawback comes from representing the sets as sequences: when the collection is non-repetitive, Re-Pair cannot compress the sets very well. On the positive side, compression is much faster and more stable.

We also tried an intermediate variant, PDL-set, that uses Re-Pair-like set compression. While ordinary Re-Pair replaces common substrings ab of length 2 with grammar rules $X \to ab$, the compressor used in PDL-set searches for symbols a and b that occur often in the same sets. Treating the sets this way should lead to better compression on non-repetitive collections, but unfortunately our current compression algorithm is still too slow with non-repetitive collections. With repetitive collections, the size of PDL-set is very similar to PDL-RP.

Representing the sets as sequences allows for storing the document identifiers in any desired order. One interesting order is the top-k order: store the identifiers in the order they should be returned by a topk() query. This forms the basis of our new PDL structure for top-k document retrieval. In each set, document identifiers are sorted by their frequencies in decreasing order, with ties broken by sorting the identifiers in increasing order. The sequences are then compressed by Re-Pair. If document frequencies are needed, they are stored in the same order as the identifiers. The frequencies can be represented space-efficiently by first run-length encoding the sequences, and then using differential encoding for the run heads. If there are b suffixes in the subtree corresponding to the set, there are $O(\sqrt{b})$ runs, so the frequencies can be encoded in $O(\sqrt{b} \log b)$ bits.

There are two basic approaches to using the PDL structure for top-k document retrieval. We can set $\beta = 0$, storing the document sets for all suffix tree nodes above the leaf blocks. This approach is very fast, as we need only decompress the first k document identifiers from the stored sequence. It works well with repetitive collections, while the total size of the document sets becomes too large with non-repetitive collections. We tried this approach with block sizes $b = 64$ (PDL-64 without frequencies and PDL-64+F with frequencies) and $b = 256$ (PDL-256 and PDL-256+F).

Alternatively, we can build the PDL structure normally with $\beta > 1$, achieving better compression. Answering queries is now slower, as we have to decompress multiple document sets with frequencies, merge the sets, and determine the top k. We tried different heuristics for merging only prefixes of the document sequences, stopping when a correct answer to the top-k query could be guaranteed. The heuristics did not generally work well, making brute-force merging the fastest alternative. We used block size $b = 256$ and storing factors $\beta = 2$ (PDL-256-2) and $\beta = 4$ (PDL-256-4). Smaller block sizes increased both index size and query times, as the number of sets to be merged was generally larger.

5 Experimental Data

We did extensive experiments with both real and synthetic collections.[1] The details of the collections can be found in the full version of the paper[2], where we also describe how the search patterns were obtained.

[1] See `http://www.cs.helsinki.fi/group/suds/rlcsa/` for datasets and full results.
[2] `http://arxiv.org/abs/1404.4909`

Most of our document collections were relatively small, around 100 MB in size, as the WT implementation uses 32-bit libraries, while Grid requires large amounts of memory for index construction. We also used larger versions of some collections, up to 1 GB in size, to see how collection size affects the results. In practice, collection size was more important in top-k document retrieval, as increasing the number of documents generally increases the $docc/k$ ratio. In document listing, document size is more important than collection size, as the performance of Brute depends on the $occ/docc$ ratio.

Real collections. Page and Revision are repetitive collections generated from a Finnish language Wikipedia archive with full version history. The collection consists of either 60 pages (small) or 280 pages (large), with a total of 8834 or 65565 revisions. In Page, all revisions of a page form a single document, while each revision becomes a separate document in Revision. Enwiki is a nonrepetitive collection of 7000 or 90000 pages from a snapshot of the English language Wikipedia. Influenza is a repetitive collection containing the genomes of 100000 or 227356 influenza viruses. Swissprot is a nonrepetitive collection of 143244 protein sequences used in many document retrieval papers (e.g., [16]). As the full collection is only 54 MB, there is no large version of Swissprot.

Synthetic collections. To explore the effect of collection repetitiveness on document retrieval performance in more detail, we generated three types of synthetic collections, using files from the Pizza & Chilli corpus[3].

DNA is similar to Influenza. Each collection has 1, 10, 100, or 1000 base documents, 100000, 10000, 1000, or 100 variants of each base document, respectively, and mutation rate $p = 0.001, 0.003, 0.01, 0.03$, or 0.1. We generated the base documents by mutating a sequence of length 1000 from the DNA file with zero-order entropy preserving point mutations, with probability $10p$. We then generated the variants in the same way with mutation rate p.

Concat is similar to Page. We read 10, 100, or 1000 base documents of length 10000 from the English file, and generated 1000, 100, or 10 variants of each base document, respectively. The variants were generated by applying zero-order entropy preserving point mutations with probability 0.001, 0.003, 0.01, 0.03, or 0.1 to the base document, and all variants of a base document were concatenated to form a single document. We also generated collections similar to Revision by making each variant a separate document. These collections are called Version.

6 Experimental Results

We implemented Brute, Sada, and PDL ourselves[4], and modified existing implementations of WT, Grid, Grammar, and LZ for our purposes. All implementations were written in C++. Details of our test machine are in the full version.

As our CSA, we used RLCSA [9], a practical implementation of a CSA that compresses repetitive collections well. The locate() support in RLCSA includes

[3] `http://pizzachili.dcc.uchile.cl/`

[4] Available at `http://www.cs.helsinki.fi/group/suds/rlcsa/`

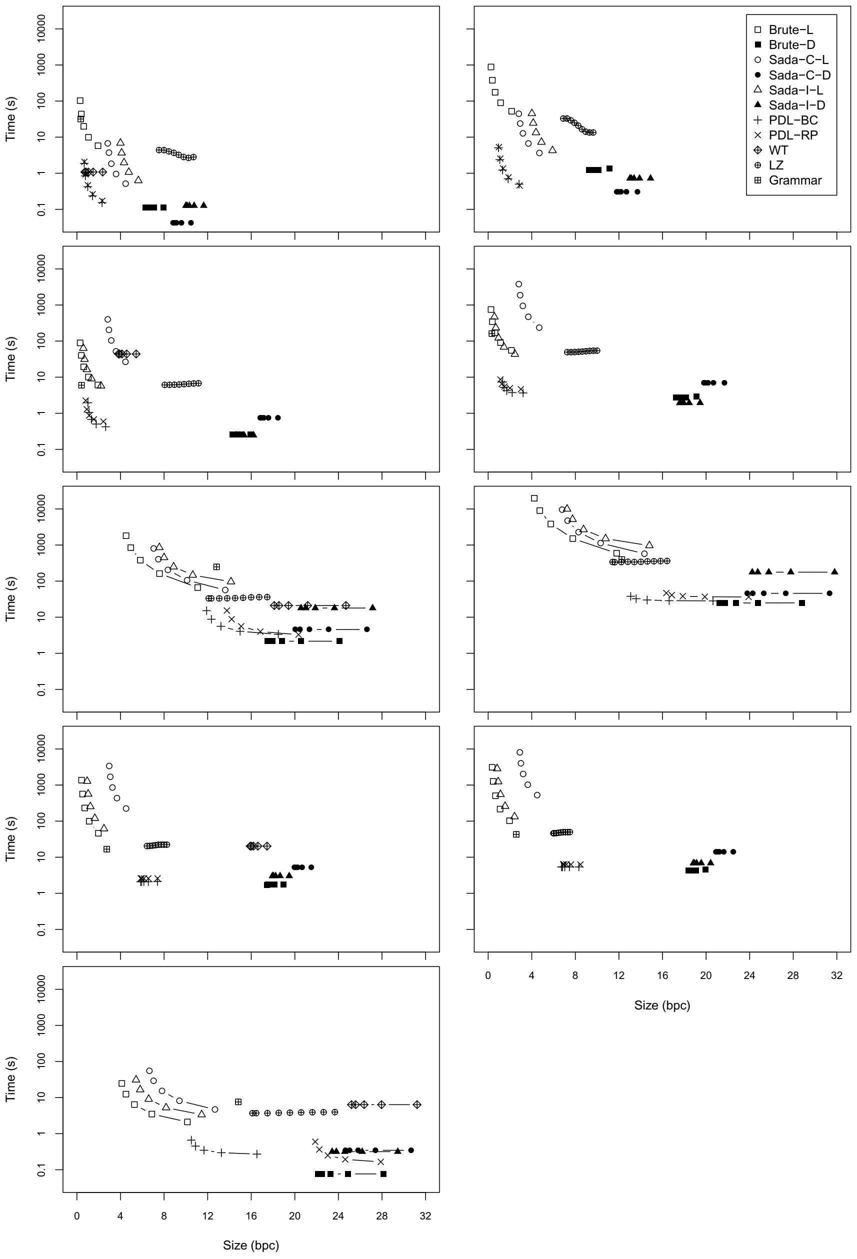


Fig. 1. Document listing on small (left) and large (right) real collections. Total size of the index in bits per character and time required to run the queries in seconds. From top to bottom, Page, Revision, Enwiki, Influenza, and Swissprot.

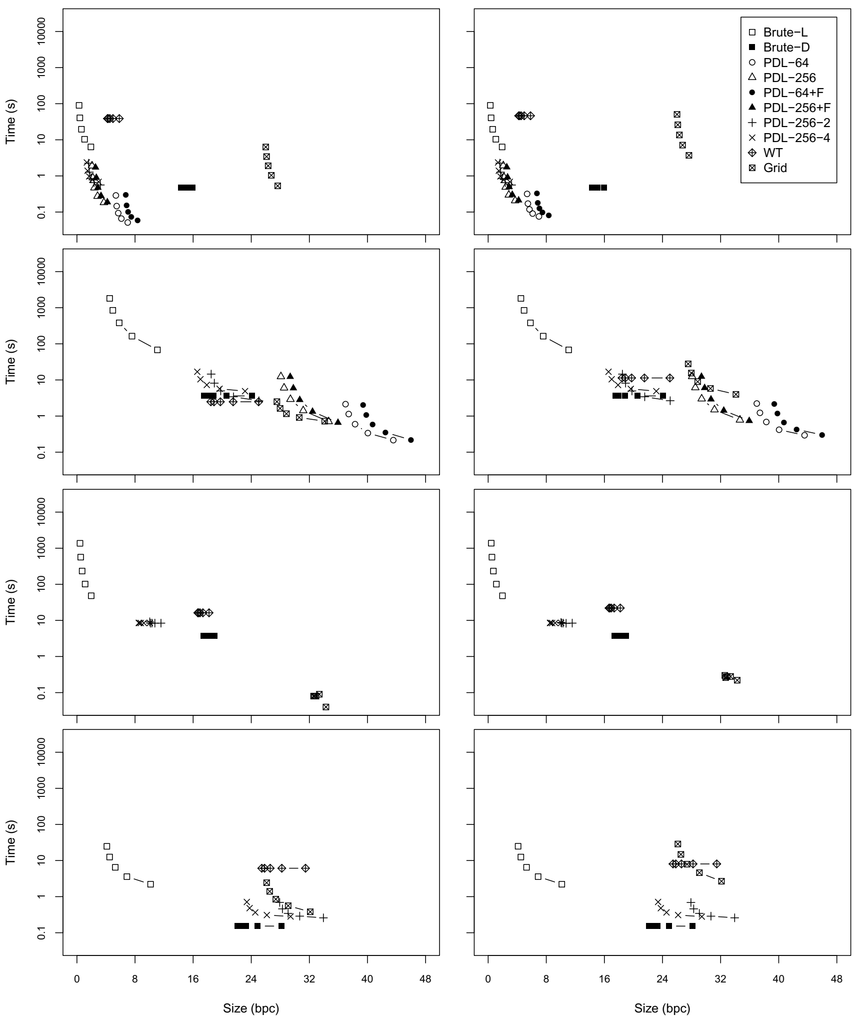


Fig. 2. Top-k document retrieval with $k = 10$ (left) and $k = 100$ (right) on small real collections. Total size of the index in bits per character and time required to run the queries in seconds. From top to bottom, Revision, Enwiki, Influenza, and Swissprot. Page is left out due to the low number of documents in that collection.

optimizations for long query ranges and repetitive collections, which is important for Brute-L and Sada-I-L. We used suffix array sample periods $8, 16, 32, 64, 128$ for non-repetitive collections and $32, 64, 128, 256, 512$ for repetitive ones.

For algorithms using a CSA, we broke the $\mathsf{list}(P)$ and $\mathsf{topk}(P, k)$ queries into a $\mathsf{find}(P)$ query, followed by a $\mathsf{list}([sp, ep])$ query or $\mathsf{topk}([sp, ep], k)$ query, respectively. The measured times do not include the time used by the $\mathsf{find}()$ query. As this time is common to all solutions using a CSA, and negligible compared to the time used by Grammar and LZ, the omission does not affect the results.

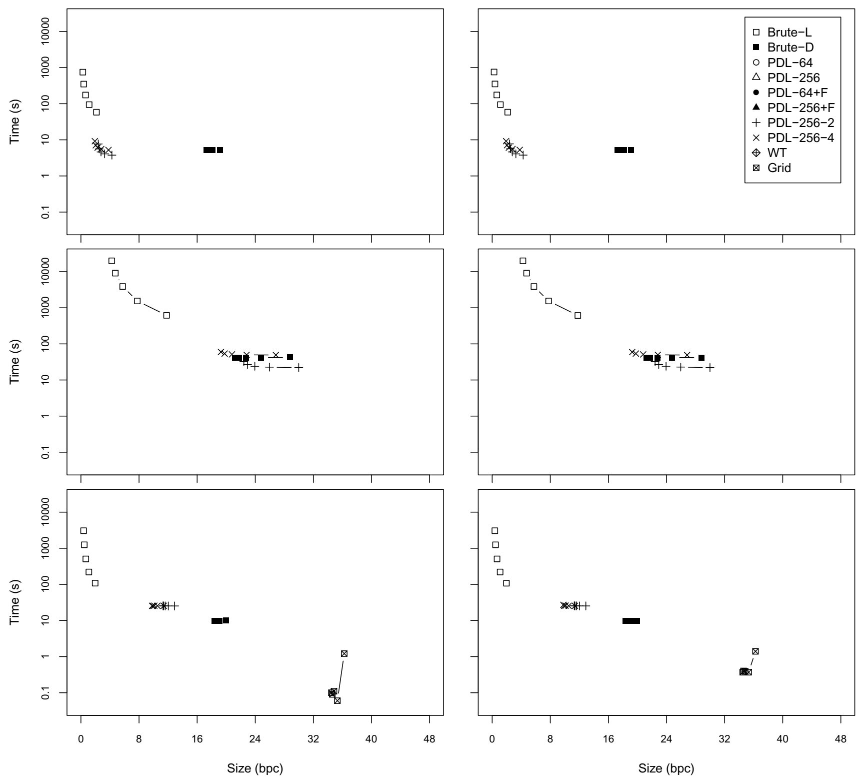


Fig. 3. Top-k document retrieval with $k = 10$ (left) and $k = 100$ (right) on large real collections. Total size of the index in bits per character and time required to run the queries in seconds. From top to bottom, Revision, Enwiki, and Influenza. Page is left out due to the low number of documents in that collection.

Document Listing with Real Collections. Figure 1 contains the results for document listing with real collections. For most of the indexes, the time/space trade-off is based on the SA sample period. LZ's trade-off comes from a parameter specific to that structure involving RMQs (see [3]). Grammar has no trade-off.

Of the small indexes, Brute-L is usually the best choice. Thanks to the locate() optimizations in RLCSA and the small documents, Brute-L beats Sada-C-L and Sada-I-L, which are faster in theory due to using locate() more selectively. When more space is available, PDL-BC is a good choice, combining fast queries with moderate space usage. Of the bigger indexes, one storing the document array explicitly is usually even faster than PDL-BC. Grammar works well with Revision and Influenza, but becomes too large or too slow elsewhere.

Top-k Document Retrieval. Results for top-k document retrieval on real collections are shown in Figures 2 and 3. Time/space trade-offs are again based on the suffix array sample period, while PDL also uses other parameters (see Section 4). We could not build PDL with $\beta = 0$ for Influenza or the large collections, as the total size of the stored sets was more than 2^{32}, which was too much for

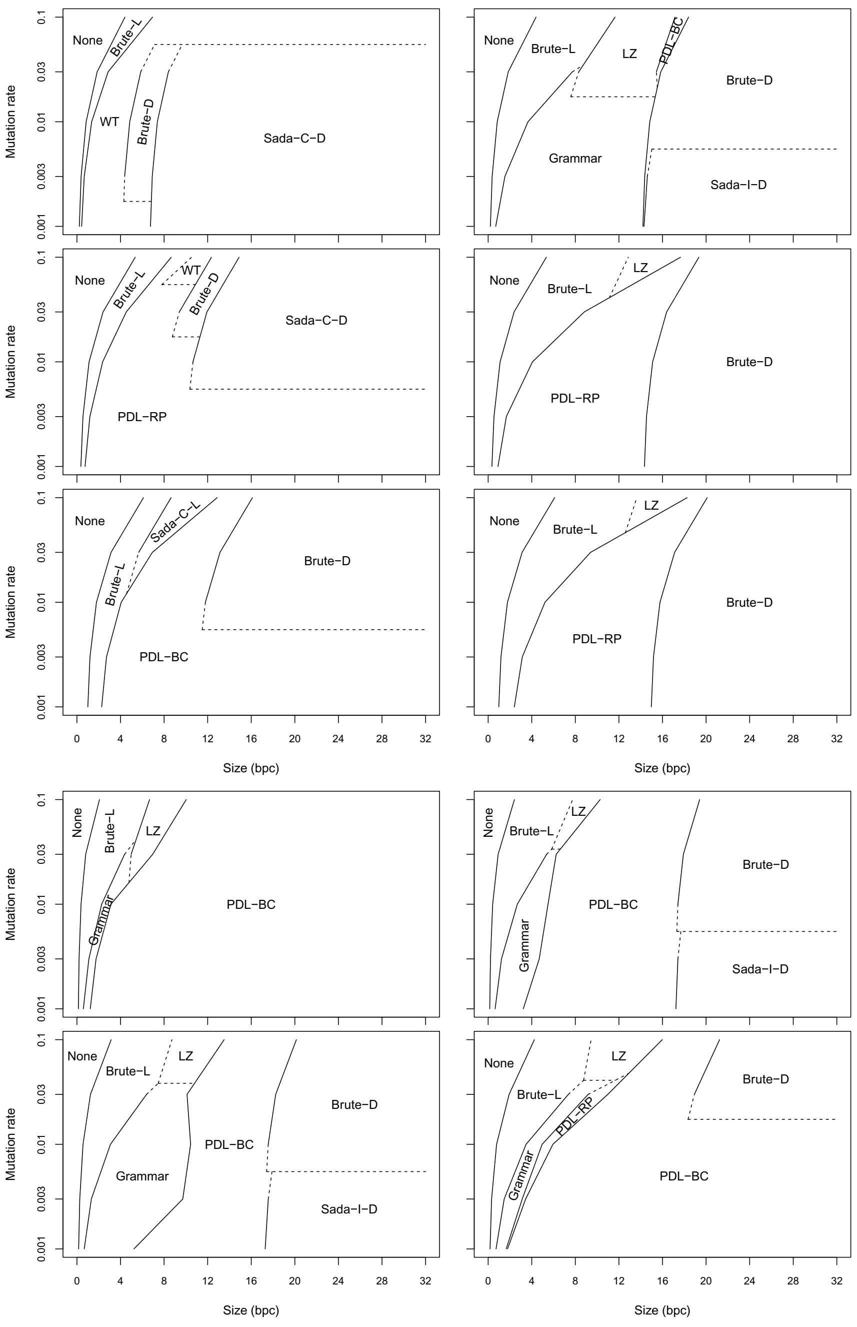


Fig. 4. Document listing with synthetic collections. The fastest solution for a given size in bits per character and a mutation rate. Top group: from top to bottom 10, 100, and 1000 base documents with Concat (left) and Version (right). Bottom group: DNA with 1 (top left), 10 (top right), 100 (bottom left), and 1000 (bottom right) base documents. None denotes that no solution can achieve that size.

our Re-Pair compressor. WT was only built for the small collections, while Grid construction used too much memory on the larger Wikipedia collections.

On Revision, PDL dominates the other solutions. On Enwiki, both WT and Grid have good trade-offs with $k = 10$, while Brute-D and PDL beat them with $k = 100$. On Influenza, some PDL variants, Brute-D, and Grid all offer good trade-offs. On Swissprot, the brute-force algorithms win clearly. PDL with $\beta = 0$ is faster, but requires far too much space (60-70 bpc — off the chart).

Document Listing with Synthetic Collections. Figure 4 shows our document listing results with synthetic collections. Due to the large number of collections, the results for a given collection type and number of base documents are combined in a single plot, showing the fastest algorithm for a given amount of space and a mutation rate. Solid lines connect measurements that are the fastest for their size, while dashed lines are rough interpolations.

The plots were simplified in two ways. Algorithms providing a marginal and/or inconsistent improvement in speed in a very narrow region (mainly Sada-C-L and Sada-I-L) were left out. When PDL-BC and PDL-RP had very similar performance, only one of them was chosen for the plot.

On DNA, Grammar was a good solution for small mutation rates, while LZ was good with larger mutation rates. With more space available, PDL-BC became the fastest algorithm. Brute-D and Sada-I-D were often slightly faster than PDL, when there was enough space available to store the document array. On Concat and Version, PDL was usually a good mid-range solution, with PDL-RP being usually smaller than PDL-BC. The exceptions were the collections with 10 base documents, where the number of variants (1000) was clearly larger than the block size (256). With no other structure in the collection, PDL was unable to find a good grammar to compress the sets. At the large end of the size scale, algorithms using an explicit DA were usually the fastest choice.

7 Conclusions

Most document listing algorithms assume that the total number of occurrences of the pattern is large compared to the number of document occurrences. When documents are small, such as Wikipedia articles, this assumption generally does not hold. In such cases, brute-force algorithms usually beat dedicated document listing algorithms, such as Sadakane's algorithm and wavelet tree-based ones.

Several new algorithms have been proposed recently. PDL is a fast and small solution, effective on non-repetitive collections, and with repetitive collections, if the collection is structured (e.g., incremental versions of base documents) or the average number of similar suffixes is not too large. Of the two PDL variants, PDL-BC has a more stable performance, while PDL-RP is faster to build. Grammar is a small and moderately fast solution when the collection is repetitive but the individual documents are not. LZ works well when repetition is moderate.

We adapted the PDL structure for top-k document retrieval. The new structure works well with repetitive collections, and is clearly the method of choice on the versioned Revision. When the collections are non-repetitive, brute-force algorithms remain competitive even on gigabyte-sized collections. While some dedicated algorithms can be faster, the price is much higher space usage.

References

1. Claude, F., Munro, J.I.: Document listing on versioned documents. In: Kurland, O., Lewenstein, M., Porat, E. (eds.) SPIRE 2013. LNCS, vol. 8214, pp. 72–83. Springer, Heidelberg (2013)
2. Claude, F., Navarro, G.: Improved grammar-based compressed indexes. In: Calderón-Benavides, L., González-Caro, C., Chávez, E., Ziviani, N. (eds.) SPIRE 2012. LNCS, vol. 7608, pp. 180–192. Springer, Heidelberg (2012)
3. Ferrada, H., Navarro, G.: A Lempel-Ziv compressed structure for document listing. In: Kurland, O., Lewenstein, M., Porat, E. (eds.) SPIRE 2013. LNCS, vol. 8214, pp. 116–128. Springer, Heidelberg (2013)
4. Gagie, T., Karhu, K., Navarro, G., Puglisi, S.J., Sirén, J.: Document listing on repetitive collections. In: Fischer, J., Sanders, P. (eds.) CPM 2013. LNCS, vol. 7922, pp. 107–119. Springer, Heidelberg (2013)
5. Hernández, C., Navarro, G.: Compressed representation of web and social networks via dense subgraphs. In: Calderón-Benavides, L., González-Caro, C., Chávez, E., Ziviani, N. (eds.) SPIRE 2012. LNCS, vol. 7608, pp. 264–276. Springer, Heidelberg (2012)
6. Hon, W.-K., Shah, R., Vitter, J.: Space-efficient framework for top-k string retrieval problems. In: Proc. FOCS, pp. 713–722 (2009)
7. Konow, R., Navarro, G.: Faster compact top-k document retrieval. In: Proc. DCC, pp. 351–360 (2013)
8. Larsson, N.J., Moffat, A.: Off-line dictionary-based compression. In: Proceedings of the IEEE Data Compression Conference, vol. 88(11), pp. 1722–1732 (2000)
9. Mäkinen, V., Navarro, G., Sirén, J., Välimäki, N.: Storage and retrieval of highly repetitive sequence collections. J. Comp. Bio. 17(3), 281–308 (2010)
10. Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. SIAM J. Computing 22(5), 935–948 (1993)
11. Muthukrishnan, S.: Efficient algorithms for document retrieval problems. In: Proc. SODA, pp. 657–666 (2002)
12. Navarro, G.: Indexing highly repetitive collections. In: Smyth, B. (ed.) IWOCA 2012. LNCS, vol. 7643, pp. 274–279. Springer, Heidelberg (2012)
13. Navarro, G.: Spaces, trees and colors: The algorithmic landscape of document retrieval on sequences. ACM Computing Surveys 46(4), article 52 (2014)
14. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys 39(1), art. 2 (2007)
15. Navarro, G., Nekrich, Y.: Top-k document retrieval in optimal time and linear space. In: Proc. SODA, pp. 1066–1078 (2012)
16. Navarro, G., Valenzuela, D.: Space-efficient top-k document retrieval. In: Klasing, R. (ed.) SEA 2012. LNCS, vol. 7276, pp. 307–319. Springer, Heidelberg (2012)
17. Sadakane, K.: Succinct data structures for flexible text retrieval systems. J. Discrete Algorithms 5, 12–22 (2007)
18. Weiner, P.: Linear pattern matching algorithm. In: Proc. 14th Annual IEEE Symposium on Switching and Automata Theory, pp. 1–11 (1973)

An Improved Analysis of the Mömke-Svensson Algorithm for Graph-TSP on Subquartic Graphs*

Alantha Newman

CNRS-Université Grenoble Alpes and G-SCOP, F-38000 Grenoble, France
firstname.lastname@grenoble-inp.fr

Abstract. Recently, Mömke and Svensson presented a beautiful new approach for the traveling salesman problem on a graph metric (graph-TSP), which yielded a $\frac{4}{3}$-approximation guarantee on subcubic graphs as well as a substantial improvement over the $\frac{3}{2}$-approximation guarantee of Christofides' algorithm on general graphs. The crux of their approach is to compute an upper bound on the minimum cost of a circulation in a particular network, $C(G,T)$, where G is the input graph and T is a carefully chosen spanning tree. The cost of this circulation is directly related to the number of edges in a tour output by their algorithm. Mucha subsequently improved the analysis of the circulation cost, proving that Mömke and Svensson's algorithm for graph-TSP has an approximation ratio of at most $\frac{13}{9}$ on general graphs.

This analysis of the circulation is local, and vertices with degree four and five can contribute the most to its cost. Thus, hypothetically, there could exist a subquartic graph (a graph with degree at most four at each vertex) for which Mucha's analysis of the Mömke-Svensson algorithm is tight. In this paper, we show that this is not the case and that Mömke and Svensson's algorithm for graph-TSP has an approximation guarantee of at most $\frac{46}{33}$ on subquartic graphs. To prove this, we present a different method to upper bound the minimum cost of a circulation on the network $C(G,T)$. Our approximation guarantee actually holds for all graphs that have an optimal solution to a standard linear programming relaxation of graph-TSP with subquartic support.

1 Introduction

The *metric* traveling salesman problem (TSP) is one of the most well-known problems in the field of combinatorial optimization and approximation algorithms. Given a complete graph, $G = (V,E)$, with non-negative edge weights that satisfy the triangle inequality, the goal is to compute a minimum cost tour of G that visits each vertex exactly once. Christofides' algorithm, dating from almost four decades ago, yields a tour with cost no more than $3/2$ times that of an optimal tour [Chr76]. It remains a major open problem to improve upon this approximation factor.

* Supported in part by LabEx PERSYVAL-Lab (ANR–11-LABX-0025).

Recently, there have been many exciting developments relating to *graph*-TSP. In this setting, we are given an unweighted graph $G = (V, E)$ and the goal is to find the shortest tour that visits each vertex *at least* once. This problem is equivalent to the special case of metric TSP where the shortest path distances in G define the metric. It is also equivalent to the problem of finding a connected, Eulerian multigraph in G with the minimum number of edges.

A promising approach to improving upon the factor of 3/2 for metric TSP is to round a linear programming relaxation known as the Held-Karp relaxation [HK70]. A lower bound of 4/3 on its integrality gap can be demonstrated using a family of graph-TSP instances. Even in this special case of metric TSP, graph-TSP had also long resisted significant progress before the recent spate of results.

1.1 Recent Progress on Graph-TSP

In 2005, Gamarnik et al. presented an algorithm for graph-TSP on cubic 3-edge connected graphs with an approximation factor of $3/2 - 5/389$ [GLS05], thus proving that Christofides' approximation factor of 3/2 is not optimal for this class of graphs. Their approach is based on finding a cycle cover for which they can upper bound the number of components. This general approach was also taken by Boyd et al. who combined it with polyhedral ideas to obtain approximation guarantees of 4/3 for cubic graphs and 7/5 for subcubic graphs, i.e. graphs with degree at most three at each vertex [BSvdSS11]. Shortly afterwards, Oveis Gharan et al. proved that a subtle modification of Christofides' algorithm has an approximation guarantee of $3/2 - \epsilon_0$ for graph-TSP on general graphs, where ϵ_0 is a fixed constant with value approximately 10^{-12} [GSS11].

Mömke and Svensson then presented a beautiful new approach for graph-TSP, which resulted in a substantial improvement over the 3/2-approximation guarantee of Christofides [MS11]. Their approach also lead to a surprisingly simple algorithm with an 4/3-approximation guarantee for subcubic graphs. We will discuss their algorithm in more detail in Section 1.2, since our paper is directly based on their approach. Ultimately, they were able to prove an approximation guarantee of 1.461 for graph-TSP. Mucha subsequently gave an improved analysis, thereby proving that Mömke and Svensson's algorithm for graph-TSP actually has an approximation ratio of at most 13/9 [Muc12]. Sebő and Vygen introduced an approach for graph-TSP based on ear decompositions and matroid intersection, which incorporated the techniques of Mömke and Svensson, and improved the approximation ratio to 7/5, where it currently stands [SV12]. For the special case of k-regular graphs, Vishnoi gave an algorithm for graph-TSP with an approximation guarantee that approaches 1 as k increases [Vis12].

Some of the new techniques for graph-TSP have also lead to progress on the metric s-t-path TSP, in which the goal is to find a path between two fixed vertices that visits every vertex at least once. Recent results improved upon the previously best-known bound of 5/3 for the s-t-path TSP due to Hoogeveen [Hoo91] in the special case of s-t-path graph-TSP [MS11, Muc12, SV12, Gao13] as well as in the case of general metrics [AKS12, Seb13].

1.2 Mömke-Svensson's Approach to Graph-TSP

Christofides' algorithm for graph-TSP finds a spanning tree of the graph and adds to it a J-join, where J is the set of vertices that have odd degree in the spanning tree. Since the spanning tree is connected, the resulting subgraph is clearly connected, and since the J-join corrects the parity of the spanning tree, the resulting subgraph is Eulerian. In contrast, the recent approach of Mömke and Svensson is based on removing an odd-join of the graph, which yields a possibly disconnected Eulerian subgraph. Thus, to maintain connectivity, one must double, rather than remove, some of the edges in the odd-join. The key step in proving the approximation guarantee of the algorithm is to show that many edges will actually be removed and relatively few edges will be doubled, resulting in a connected, Eulerian subgraph with few edges. Using techniques of Naddef and Pulleyblank [NP81], Mömke and Svensson show how to sample an odd-join of size $|E|/3$, where E is the subset of edges in the support of the linear programming relaxation for graph-TSP (see section 2.1). The number of edges that are doubled to guarantee connectivity is directly related to the minimum cost circulation of particular network, referred to as $C(G,T)$, which Mömke and Svensson construct based on the input graph G, an optimal solution to a linear programming relaxation for graph-TSP, and a carefully chosen spanning tree T. Lemma 4.1 from [MS11] relates the size of the solution for their algorithm to the minimum cost circulation of this network.

Lemma 1. [MS11] *Given a 2-vertex connected graph G and a depth first search tree T of G, let C^* be a minimum cost circulation for $C(G,T)$ of cost $c(C^*)$. Then there is a spanning Eulerian multigraph in G with at most $\frac{4}{3}n + \frac{2}{3}c(C^*)$ edges.*

We defer a precise description of the circulation network $C(G,T)$ to Section 2, where we formulate it using different notation from that in [MS11]. For the moment, we emphasize that if one can prove a better upper bound on the value of $c(C^*)$, then this directly implies improved upper bounds on the number of edges in a tour output by Mömke and Svensson's algorithm.

1.3 Our Contribution

We consider the graph-TSP problem for *subquartic* graphs, i.e. graphs in which each vertex has degree at most four. As pointed out in Lemma 2.1 of [MS11], we can assume that these graphs are 2-vertex connected. The best-known approximation guarantee for these graphs is inherited from the general case, even when the graph is 4-regular, and is therefore 7/5 due to Sebő and Vygen. For subquartic graphs, we give an improved upper bound on the minimum cost of a circulation for $C(G,T)$. Using Lemma 1, this leads to an improved approximation guarantee of 46/33 for graph-TSP on these graphs. Before we give an overview of our approach, we first explain our motivation for studying graph-TSP on this restricted class of graphs.

As mentioned in Section 1.1, graph-TSP is now known to be approximable to within 4/3 for subcubic graphs. So, on the one hand, trying to prove the same guarantee for subquartic graphs is arguably a natural next step. Additionally, it is a well-motivated problem to study the graph-TSP on sparse graphs, because the support of an optimal solution to the standard linear programming relaxation (reviewed in Section 2.1) has at most $2n-1$ edges (see Theorem 4.9 in [CFN85]). Thus, any graph that corresponds to the support of an optimal solution to the standard linear program has average degree less than four.

However, our actual motivation for studying graphs with degree at most four has more to do with understanding the Mömke-Svensson algorithm than with an abstract interest in subquartic graphs. The basic approach to computing an upper bound on the minimum cost circulation in $C(G,T)$ used in both [MS11] and [Muc12] is to specify flow values on the edges of $C(G,T)$ that are functions of an optimal solution to the linear programming relaxation for graph-TSP on the graph G. The cost of the circulation obtained using these values can be analyzed in a local, vertex by vertex manner. Mucha showed that vertices with degree four or five potentially increase the cost of the circulation the most [Muc12]. In fact, one could hypothetically construct a tight example for Mucha's analysis of the Mömke-Svensson algorithm on a graph where each vertex has degree at most four (or where each vertex has degree at most five). Thus it seems worthwhile to determine if the cost of the circulation can be improved on subquartic graphs. Our results actually hold for a slightly more general class of graphs than subquartic graphs: they hold for any graph that has an optimal solution to the standard linear programming relaxation of graph-TSP with subquartic support.

1.4 Organization

In Section 2.1, we discuss the standard linear programming relaxation for graph-TSP, and in Section 2.2, we present notation and definitions necessary for defining the circulation network $C(G,T)$. In Section 3, we show that if, for a subquartic graph, the optimal solution to the linear program has value equal to the number of vertices in G, then the network $C(G,T)$ has a circulation of cost zero, implying that the Mömke-Svensson algorithm has an approximation ratio of 4/3. This observation provides us with some intuition as to how one may attempt to design a better circulation for general subquartic graphs.

In Section 4, we describe two different methods to obtain feasible circulations. In Section 4.1, we detail the method used by Mömke-Svensson and Mucha, which becomes somewhat simpler in the special case of subquartic graphs. This method directly uses values from the optimal solution to the linear program to obtain flow values on edges in the network. In Section 4.2, we present a new method, which "rounds" the values from the optimal solution to the linear program. The latter circulation alone leads to an improved analysis over 13/9 for subquartic graphs, but it does not improve on the best-known guarantee of 7/5. However, as we finally show in Section 5, if we take the best of the two circulations, we can show that at least one of the circulations will lead to an approximation guarantee of at most 46/33.

We remark that our notation differs from that in [MS11] or [Muc12], even though we are using exactly the same circulation network and we use their approach for obtaining the feasible circulation described in Section 4.1. This different notation allows us to more easily analyze the tradeoff between the two circulations. Due to space constraints, this extended abstract is missing many proofs, which can be found in the full version.

2 Preliminaries: Notation and Definitions

Let $G = (V, E)$ be an undirected graph with maximum degree four, a property often referred to as *subquartic*. Throughout this paper, we make use of the following well-studied linear programming relaxation for graph-TSP.

2.1 Linear Program for Graph-TSP

For a graph $G = (V, E)$, the following linear program is a relaxation of graph-TSP. We refer to Section 2 of [MS11] for a discussion of its derivation and history.

$$\begin{aligned} \min \sum_{e \in E} y_e & \\ y(\delta(S)) &\geq 2 \quad \text{for } \emptyset \neq S \subset V, \\ y &\geq 0. \end{aligned}$$

We denote this linear program by $LP(G)$ and we denote the value of an optimal solution for $LP(G)$ by $OPT_{LP}(G)$. Let n be the number of vertices in V. We can assume that G has the following two properties: (i) $|E| \leq 2n - 1$, and (ii) G is 2-vertex connected. Assumption (i) is based on the fact that any extreme point of $LP(G)$ has at most $2n-1$ edges (see Theorem 4.9 in [CFN85]), and restricting the graph to the edges in the support of an extreme point with optimal value does not increase the optimal value $OPT_{LP}(G)$. Assumption (ii) is based on Lemma 2.1 from [MS11]. We note that the two theorems we just cited may have to be applied multiple times to guarantee that G has the desired properties (i) and (ii).

Lemma 2. *Let $G = (V, E)$ be a 2-edge connected graph. Then there exists $x \in LP(G)$, $x \leq 1$ minimizing the sum of coordinates of a vector in $LP(G)$.*

From Lemma 2, we define $x \in \mathbb{R}^{|E|}$ to be an optimal solution for $LP(G)$ with the following properties: (i) the support of x contains at most $2n - 1$ edges, (ii) the support of x is 2-vertex connected, and (iii) $x \leq 1$. We will refer to the set of values $\{x_e\}$ for $e \in E$ as x-values. Let $\sum_{e \in E} x_e = OPT_{LP}(G) = (1 + \epsilon)n$ for some ϵ, where $0 \leq \epsilon \leq 1$. We will eventually make use of the following definitions.

Definition 1. *The* excess *x-value $\epsilon(v)$ at a vertex v is the amount by which the total value on the adjacent edges exceeds 2, i.e $\epsilon(v) = x(\delta(v)) - 2$.*

Definition 2. *A vertex $v \in V$ is called* heavy *if $x(\delta(v)) > 2$.*

The following fact will be useful in our analysis. If $OPT_{LP}(G) = (1+\epsilon)n$, then,

$$\sum_{v \in V} x(\delta(v)) = \sum_{v \in V} (2 + \epsilon(v)) \quad = \quad 2(1+\epsilon)n.$$

This implies, $\sum_{v \in V} \epsilon(v) = 2\epsilon n$.

2.2 Spanning Trees and Circulations

Let us recall some useful definitions from the approach of Mömke and Svensson [MS11] that we use throughout this paper.

Definition 3. *A* greedy DFS tree *is a spanning tree formed via a depth-first search of G. If there is a choice as to which edge to traverse next, the edge with the highest x-value is chosen.*

For a given graph G and an optimal solution to $LP(G)$, let T denote a greedy DFS tree. Let $B(T) \subset E$ denote the set of back edges with respect to the tree T. Each edge in T will be directed away from the root of the tree T and each edge in $B(T)$ will be directed towards the root of T. We use the notation (i,j) to denote an edge directed from i to j. Note that once we have fixed a tree T, all edges in E can be viewed as directed edges. When we wish to refer to an undirected edge in E, we use the notation $ij \in E$. With respect to the greedy DFS tree T, we have the following definitions.

Definition 4. *An* internal *node in T is a vertex that is neither the root of T nor a leaf in T. We use T_{int} to denote this subset of vertices.*

Definition 5. *An* expensive *vertex is a vertex in T_{int} with two incoming edges that belong to $B(T)$. We use T_{exp} to denote this subset of vertices.*

As we will see in Lemma 4, expensive vertices are the vertices that can contribute to the cost of $C(G,T)$. The root can also contribute a negligible value of either one or two to the cost of $C(G,T)$. For the sake of simplicity, we ignore the contribution of the root in most of our calculations.
Fact. The number of expensive vertices is bounded as follows: $|T_{exp}| \leq n/2$.

Definition 6. *A* branch *vertex in T is a vertex with at least two outgoing tree edges.*

Lemma 3. *A branch vertex is not expensive.*

Definition 7. *A* tree cut *is the partition of the vertices of the tree T induced when we remove an edge $(u,v) \in T$.*

For each edge $(i,j) \in B(T)$, let $b(i,j) \leq 1$ be a non-negative value.

Definition 8. *Consider a tree cut corresponding to edge $(u,v) \in T$ and remove all back edges $(w,u) \in B(T)$, where w belongs to the subtree of v in T. We say that the remaining back edges that cross this tree cut* cover *the cut. If the total b-value of the edges that cover the cut is at least 1, then we say that this tree cut is* satisfied by *b.*

We extend this definition to the vertices of T.

Definition 9. *A vertex v in T is* satisfied by b *if for each adjacent outgoing edge in T, the corresponding tree cut is satisfied by b. On the other hand, if there is at least one adjacent outgoing edge whose corresponding tree cut is not satisfied by b, then the vertex v is* unsatisfied by b.

Mömke and Svensson define a circulation network, $C(G,T)$ (see Section 4 of [MS11]), and use the cost of a feasible circulation to upper bound the length of a TSP tour in G. (See Lemma 1.)

Lemma 4. *Let $b : B(T) \to [0,1]$. If each internal vertex in T is satisfied by b, then there is a feasible circulation of $C(G,T)$ whose cost is upper bounded by the following function:*

$$\sum_{j \in T_{exp}} \max \left\{ 0, \left(\sum_{i:(i,j) \in B(T)} b(i,j) \right) - 1 \right\}. \tag{1}$$

Although finding b-values for the back edges that satisfy all the vertices is equivalent to finding a feasible circulation of $C(G,T)$, and we could have stuck to the notation presented in [MS11], we believe our notation results in a clearer presentation of our main theorems.

3 Subquartic Graphs: $OPT_{LP}(G) = n$

We now show that in the special case when $OPT_{LP}(G) = n$ (i.e. $\epsilon = 0$), there is a circulation with cost zero. Note that if $|E| = n$, then each edge in E must have x-value 1. Thus, G is a Hamiltonian cycle. If $|E| > n$, then we can show that we can find a greedy DFS tree T for G such that each edge $ij \in E$ with x-value $x_{ij} = 1$ (a "1-edge") belongs to T.

Lemma 5. *When $OPT_{LP}(G) = n$ and $|E| > n$, there is a greedy DFS tree T such that all 1-edges are in T.*

For the rest of Section 3, let T denote a greedy DFS tree in which all 1-edges are tree edges.

Lemma 6. *If $OPT_{LP}(G) = n$ and each back edge $(i,j) \in B(T)$ is assigned value $f(i,j) = 1/2$, then each vertex in T_{int} is satisfied by f.*

Lemma 7. *If $OPT_{LP}(G) = n$, setting $f(i,j) = 1/2$ for each edge $(i,j) \in B(T)$ yields a circulation with cost zero.*

Theorem 1. *If $OPT_{LP}(G) = n$ and G is a subquartic graph, then G has a TSP tour of length at most $4n/3$.*

4 Subquartic Graphs: General Case

In this section, we consider the general case of subquartic graphs. For a graph $G = (V, E)$, suppose $OPT_{LP}(G) = (1 + \epsilon)n$ for some $\epsilon > 0$. There is a fixed greedy DFS tree T as defined in Section 2.2. If we assign values to the edges in $B(T)$, then the only vertices that can add to the cost function are the expensive vertices, as we have defined them, since the maximum value allowed on an edge is one. Let $x(i, j) = x_{ij}$ for all back edges in $B(T)$. Recall that the $\{x_{ij}\}$ values are obtained from the solution to $LP(G)$ in Section 2.1.

Lemma 8. *A vertex v in T_{int} has at most one outgoing tree edge whose corresponding tree cut is not satisfied by x.*

Definition 10. *A vertex $v \in T_{int}$ that is satisfied by x is called* LP-satisfied.

Definition 11. *A vertex $v \in T_{int}$ that is not satisfied by x is called* LP-unsatisfied.

Lemma 9. *An expensive vertex is LP-satisfied.*

Lemma 10. *An LP-unsatisfied vertex is heavy.*

The reason we emphasize that an LP-unsatisfied vertex is heavy is that we can use the excess x-value of this vertex to increase an edge that covers the unsatisfied tree cut corresponding to one of its adjacent outgoing edges so that this tree cut becomes satisfied. We also wish to use the excess x-value of an expensive vertex to pay for some of its contribution to the cost function incurred by the back edges coming into the vertex. For each vertex v, we want to use the quantity $\epsilon(v)$ at most once. This will be guaranteed by the fact that LP-unsatisfied vertices and expensive vertices are disjoint sets.

4.1 The x-Circulation

In this section, we use the x-values to obtain an upper bound on the cost of a circulation, essentially following the arguments of Mömke and Svensson [MS11] and Mucha [Muc12]. We present the analysis here, since we refer to it in Section 5 when we analyze the cost of taking the best of two circulations. Also, the arguments can be somewhat simplified due to the subquartic structure of the graph, which is useful for our analysis.

For each back edge in $B(T)$, set $x(i, j) = x_{ij}$, where $x \in \mathbb{R}^{|E|}$ is an optimal solution to $LP(G)$. (For a vertex $j \notin T_{exp}$, we can actually set $x(i, j) = 1$, since there is at most one incoming back edge to vertex j, but this does not change the worst-case analysis.)

Definition 12. *For each vertex $j \in T_{exp}$, let $x_{min}(j) \leq x_{max}(j)$ denote the x-values of the two incoming back edges to vertex j. Let $c_x(j) = x_{min}(j) + x_{max}(j) - 1 - \epsilon(j)$.*

Lemma 11. *For an expensive vertex $j \in T_{exp}$, the following holds:*

$$2 \cdot x_{max}(j) + x_{min}(j) \leq 2 + \epsilon(j).$$

We will show that there is a function $x' : B(T) \to [0, 1]$ such that each vertex in T is satisfied by x' and the cost of the circulation can be bounded by:

$$\sum_{j \in T_{exp}} \max\left\{0, \left(\sum_{i:(i,j) \in B(T)} x'(i,j)\right) - 1\right\} \leq \sum_{j \in T_{exp}} \max\{0, c_x(j)\} + \sum_{j \in T} \epsilon(j) \quad (2)$$

Lemma 12. *The value $c_x(j)$ can be upper bounded as follows:*

$$c_x(j) \leq \frac{x_{min}(j)}{2} - \frac{\epsilon(j)}{2} \;\leq\; 1 - x_{min}(j).$$

Lemma 13. *For a vertex $j \in T_{exp}$, $c_x(j) \leq 1/3$.*

To make the circulation feasible, we need to increase the x-values of some of the back edges in $B(T)$ so that all of the LP-unsatisfied vertices become satisfied. By Lemma 10, these vertices are heavy. Thus, we will use $\epsilon(v)$ for an LP-unsatisfied vertex v to "pay" for increasing the x-value on an appropriate back edge. For ease of notation, we now set $x'(u, v) = x(u, v)$ for all $(u, v) \in B(T)$. We will update the $x'(u, v)$ values so that each LP-unsatisfied vertex is satisfied by x'.

Consider an LP-unsatisfied, non-branch vertex $j \in T$, and consider the tree cut corresponding to the single edge (j, t_2) outgoing from j in T. Let $S \subseteq B(T)$ denote the edges that cover this tree cut. Let $(i, j), (j, k) \in B(T)$ represent the adjacent back edges, and let $(t_1, j) \in T$ denote the incoming tree edge. Recall that in this tree cut, both edges (j, t_2) and (i, j) are removed and the remaining edges in $B(T)$ that cross this cut *cover* it. We have:

$$x(j, t_2) + x(j, k) + x(t_1, j) + x(i, j) = 2 + \epsilon(j).$$

Since,

$$x(S) + x(j, t_2) + x(i, j) \;\geq\; 2, \quad x(S) + x(j, k) + x(t_1, j) \;\geq\; 2,$$

it follows that

$$2 \cdot x(S) \;\geq\; 2 - \epsilon(j) \quad \Rightarrow \quad x(S) \;\geq\; 1 - \epsilon(j)/2.$$

Let $(u, v) \in S$ be an arbitrary edge in S. We will update the value of $x'(u, v)$ as follows:

$$x'(u, v) := \min\{1,\ x'(u, v) + \epsilon(j)/2\}.$$

We use this notation, because a back edge's value can be increased multiple times in the process of satisfying all LP-unsatisfied vertices.

If j is an LP-unsatisfied branch vertex, then it must have two outgoing edges in T (call them (j, t_2) and (j, t_3)) and one incoming back edge $(i, j) \in B(T)$.

Let (t_1, j) denote the incoming tree edge. Suppose that vertex i is in the subtree hanging from t_2 in T. Then consider the tree cut corresponding to edge (j, t_2), i.e. remove edges (j, t_2) and (i, j). Let $S \subset B(T)$ denote the back edges that cover this tree cut. Then we have,

$$x(S) + x(j, t_2) + x(i, j) \ \geq \ 2, \quad x(S) + x(j, t_3) + x(t_1, j) \ \geq \ 2.$$

We can conclude that $x(S) \geq 1 - \epsilon(j)/2$. Thus, as we did previously, we can increase the x'-value of some edge in S by the quantity $\epsilon(j)/2$. The following Lemma follows by the construction of the x' values.

Lemma 14. *The cost of satisfying all of the LP-unsatisfied vertices is at most* $\sum_{j \in T \setminus T_{exp}} \epsilon(j)/2$*. In other words:*

$$\sum_{(u,v) \in B(T)} (x'(u, v) - x(u, v)) \leq \sum_{j \in T \setminus T_{exp}} \frac{\epsilon(j)}{2}.$$

Since all vertices in T are now satisfied by x', the x'-values can be used to compute an upper bound on the cost of a feasible circulation of $C(G, T)$.

Theorem 2. *The function* $x' : B(T) \to [0, 1]$ *corresponds to a feasible circulation of* $C(G, T)$ *with cost at most:*

$$\sum_{j \in T_{exp}} \max\{0, \ c_x(j)\} + \sum_{j \in T} \epsilon(j).$$

Theorem 3. *When* $OPT_{LP}(G) = (1 + \epsilon)n$*, there is a feasible circulation for* $C(G, T)$ *with cost at most* $n/6 + 2\epsilon n$*.*

4.2 The f-Circulation

Now we describe a new method to obtain a feasible circulation, i.e. how to obtain values $f'(i, j)$ for each edge $(i, j) \in B(T)$ such that each vertex in T is satisfied by f'. The values will be used to demonstrate an improved upper bound on the cost of a circulation of $C(G, T)$ when G is a subquartic graph. In this section, we will prove the following theorem, which implies that the Mömke-Svensson algorithm has an approximation guarantee of 17/12 for graph-TSP on subquartic graphs.

Theorem 4. *When* $OPT_{LP}(G) = (1 + \epsilon)n$*, there is a feasible circulation for* $C(G, T)$ *with cost at most* $n/8 + 2\epsilon n$*.*

Consider a vertex $v \in T_{exp}$. If both incoming back edges had f-value 1/2, then this vertex would not contribute anything to the cost of the circulation. Thus, on a high level, our goal is to find f-values that are as close to half as possible, while at the same time not creating any additional unsatisfied vertices. The f-value therefore corresponds to a decreased x-value if the x-value is high, and an increased x-value if the x-value is low. A set of f-values corresponding to decreased x-values may pose a problem if they correspond to the set of back

$$\boxed{\begin{array}{rcl} x_{ij} > 3/4 & \Rightarrow & f(i,j) = 2x_{ij} - 1, \\ x_{ij} < 1/4 & \Rightarrow & f(i,j) = 2x_{ij}, \\ 1/4 \le x_{ij} \le 3/4 & \Rightarrow & f(i,j) = 1/2. \end{array}}$$

Fig. 1. Rules for creating the f-values from the x-values

edges that cover an LP-unsatisfied vertex. However, in Section 4.1, we only used $\epsilon(j)/2$ to satisfy an LP-unsatisfied vertex j. We can actually use at least $\epsilon(j)$. This observation allows us to decrease the x-values. We use the rules shown in Figure 1 to determine the values $f : B(T) \to [0,1]$.

Lemma 15. *If a vertex v is LP-satisfied, then it is satisfied by f.*

Definition 13. *For each vertex $j \in T_{exp}$, let $c_f(j) = \sum_{i:(i,j)\in B(T)} f(i,j) - 1 - \epsilon(j)$.*

For ease of notation, set $f'(u,v) = f(u,v)$ for all $(u,v) \in B(T)$.

Lemma 16. *For an LP-unsatisfied vertex $v \in T_{int}$, if we increase by the amount $\epsilon(v)$ the f'-value of an edge that covers its unsatisfied tree cut, then vertex v will be satisfied by f'.*

Lemma 17. *For $j \in T_{exp}$, if $x_{min}(j), x_{max}(j) \ge 1/2$ or if $x_{min}(j), x_{max}(j) \le 3/4$, then $c_f(j) \le 0$.*

Lemma 18. *If $x_{max}(j) \ge 3/4$ and $0 < x_{min}(j) \le 1/2$, then $c_f(j) \le \min\{x_{min}(j),\ 1/2 - x_{min}(j)\}$.*

Similar to the approach taken in Section 4.1, any LP-unsatisfied vertex can have the value of the edges covering the unsatisfied tree cut by adding $\epsilon(v)$ to one of the edges covering the cut. We now have the following theorem.

Theorem 5. *When $OPT_{LP}(G) = (1+\epsilon)n$, there is a feasible circulation for $C(G,T)$ with cost at most $n/8 + 2\epsilon n$.*

5 Combining the x- and the f-circulations

We can classify each vertex in T_{exp} according to the value of $x_{min}(j)$. Intuitively, if many vertices contribute a lot, say 1/3 to the x-circulation, then they will not contribute a lot of the f-circulation, and vice versa.

$x_{min}(j)$	$c_x(j)$	$c_f(j)$
$[0, 1/4]$	$x_{min}(j)/2$	$x_{min}(j)$
$[1/4, 1/2]$	$x_{min}(j)/2$	$1/2 - x_{min}(j)$
$[1/2, 2/3]$	$x_{min}(j)/2$	0
$[2/3, 1]$	$1 - x_{min}(j)$	0

Theorem 6. *When $OPT_{LP}(G) = (1+\epsilon)n$, there is a feasible circulation for $C(G,T)$ with cost at most $n/11 + 2\epsilon n$.*

Theorem 7. *The approximation guarantee of the Mömke-Svensson algorithm on subquartic graphs is at most* 46/33.

6 Final Remarks

We note that we can obtain a slightly better approximation ratio by allowing an larger coefficient in front of the amount ϵn in Theorem 6. However, the improvement we obtain from this is extremely small (approximately 1.393) and not worth the technical equations.

Acknowledgements. We wish to thank Sylvia Boyd, Satoru Iwata, R. Ravi, András Sebő and Ola Svensson for helpful discussions and comments. This work was done in part while the author was a member of the THL2 group at EPFL.

References

[AKS12] An, H.-C., Kleinberg, R., Shmoys, D.B.: Improving Christofides' algorithm for the *s*-*t*-path TSP. In: Proceedings of the 44th Symposium on Theory of Computing, pp. 875–886. ACM (2012)

[BSvdSS11] Boyd, S., Sitters, R., van der Ster, S., Stougie, L.: TSP on cubic and subcubic graphs. In: Günlük, O., Woeginger, G.J. (eds.) IPCO 2011. LNCS, vol. 6655, pp. 65–77. Springer, Heidelberg (2011)

[CFN85] Cornuéjols, G., Fonlupt, J., Naddef, D.: The traveling salesman problem on a graph and some related integer polyhedra. Mathematical Programming 33(1), 1–27 (1985)

[Chr76] Christofides, N.: Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, DTIC Document (1976)

[Gao13] Gao, Z.: An LP-based-approximation algorithm for the *s*-*t* path graph traveling salesman problem. Operations Research Letters 41(6), 615–617 (2013)

[GLS05] Gamarnik, D., Lewenstein, M., Sviridenko, M.: An improved upper bound for the TSP in cubic 3-edge-connected graphs. Operations Research Letters 33(5), 467–474 (2005)

[GSS11] Gharan, S.O., Saberi, A., Singh, M.: A randomized rounding approach to the traveling salesman problem. In: 2011 IEEE 52nd Annual Symposium on Foundations of Computer Science (FOCS), pp. 550–559. IEEE (2011)

[HK70] Held, M., Karp, R.M.: The traveling-salesman problem and minimum spanning trees. Operations Research 18(6), 1138–1162 (1970)

[Hoo91] Hoogeveen, J.A.: Analysis of Christofides' heuristic: some paths are more difficult than cycles. Operations Research Letters 10(5), 291–295 (1991)

[MS11] Mömke, T., Svensson, O.: Approximating graphic TSP by matchings. In: IEEE 52nd Annual Symposium on Foundations of Computer Science, pp. 560–569 (2011)

[Muc12] Mucha, M.: 13/9-approximation for graphic TSP. In: Dürr, C., Wilke, T. (eds.) STACS. LIPIcs, vol. 14, pp. 30–41. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2012)

[NP81] Naddef, D., Pulleyblank, W.R.: Matchings in regular graphs. Discrete Mathematics 34(3), 283–291 (1981)

[Seb13] Sebő, A.: Eight-fifth approximation for the path TSP. In: Goemans, M., Correa, J. (eds.) IPCO 2013. LNCS, vol. 7801, pp. 362–374. Springer, Heidelberg (2013)

[SV12] Sebő, A., Vygen, J.: Shorter tours by nicer ears: 7/5-approximation for graphic TSP, 3/2 for the path version, and 4/3 for two-edge-connected subgraphs. arXiv:1201.1870 (2012)

[Vis12] Vishnoi, N.K.: A permanent approach to the traveling salesman problem. In: 2012 IEEE 53rd Annual Symposium on Foundations of Computer Science, pp. 76–80. IEEE (2012)

The Input/Output Complexity of Sparse Matrix Multiplication[⋆]

Rasmus Pagh and Morten Stöckel

IT University of Copenhagen
{pagh,mstc}@itu.dk

Abstract. We consider the problem of multiplying sparse matrices (over a semiring) where the number of non-zero entries is larger than main memory. In the classical paper of Hong and Kung (STOC '81) it was shown that to compute a product of dense $U \times U$ matrices, $\Theta\left(U^3/(B\sqrt{M})\right)$ I/Os are necessary and sufficient in the I/O model with internal memory size M and memory block size B.

In this paper we generalize the upper and lower bounds of Hong and Kung to the sparse case. Our bounds depend of the number $N = \mathtt{nnz}(A) + \mathtt{nnz}(C)$ of nonzero entries in A and C, as well as the number $Z = \mathtt{nnz}(AC)$ of nonzero entries in AC.

We show that using $\tilde{O}\left(\frac{N}{B}\min\left(\sqrt{\frac{Z}{M}}, \frac{N}{M}\right)\right)$ I/Os, AC can be computed with high probability. This is tight (up to polylogarithmic factors) when only semiring operations are allowed, even for dense rectangular matrices: We show a lower bound of $\Omega\left(\frac{N}{B}\min\left(\sqrt{\frac{Z}{M}}, \frac{N}{M}\right)\right)$ I/Os.

While our lower bound uses fairly standard techniques, the upper bound makes use of "compressed matrix multiplication" sketches, which is new in the context of I/O-efficient algorithms, and a new matrix product size estimation technique that avoids the "no cancellation" assumption.

1 Introduction

In this paper we consider the fundamental problem of multiplying matrices that are *sparse*, that is, the number of nonzero entries in the input matrices (but not necessarily the output matrix) is much smaller than the number of entries. Matrix multiplication is a fundamental operation in computer science and mathematics, due to the wide range of applications and reductions to it — e.g. computing the determinant and inverse of a matrix, or Gaussian elimination. Matrix multiplication has also seen lots of use in non-obvious applications such as bioinformatics [24], computing matchings [22,18] and algebraic reasoning about graphs, e.g. cycle counting [2,3].

[⋆] This work is supported by the Danish National Research Foundation under the Sapere Aude program.

Matrix multiplication in the general case has been widely applied and studied in a pure math context for decades. In an algorithmic context matrix multiplication is known to be computable using $O(n^\omega)$ ring operations, for some constant ω between 2 and 3. The first improvement over the trivial cubic algorithm was achieved in 1969 in the seminal work of Strassen [23] showing $\omega \leq \log_2 7$ and most recently Vassilevska Williams [26] improved this to $\omega < 2.373$.

Matrix multiplication over a semiring, where additive inverses cannot be used, is better understood. In the I/O model introduced by Aggarwal and Vitter [1] the optimal matrix multiplication algorithm for the dense case already existed (see Section 1.2) and since then sparse-dense and sparse-sparse combinations of vector and matrix products have been studied, e.g. in [6,12,19].

The main contribution of this paper is a tight bound for matrix multiplication over a semiring in terms of the number of nonzero entries in the input and output matrices, generalizing the classical result of Hong and Kung on dense matrices [14] to the sparse case.

1.1 Preliminaries

Let $A \in R^{U\times U}$ and $C \in R^{U\times U}$ be matrices of U rows and U columns and let every entry $[A]_{i,j}, [C]_{i',j'} \in R$ for semiring R. Further for matrix A let A_{i*} denote row i of A and let A_{*j} denote column j of A. The matrix product AC, where each entry $[AC]_{i,j}$, $i,j \in [U]$ is given as $[AC]_{i,j} = \sum_k [A]_{i,k}[C]_{k,j}$. A nonzero term $[A]_{i,k}[C]_{k,j}$ is referred to as an *elementary product*. We say that there is *no cancellation* of terms when $[AC]_{i,j} = 0$ implies that $[A]_{i,k}[C]_{k,j} = 0$ for all k. For *sparse* semiring matrix multiplication, the number of entry pairs with nonzero product measures the number of operations performed up to a constant factor assuming optimal representation of the matrices. Specifically, let $\sum_{k=1}^{n} |\{j \mid [A]_{j,k} \neq 0\}||\{i \mid [C]_{k,i} \neq 0\}|$ be the number of such nonzero pairs of matrix entries. Finally let $\texttt{nnz}(A) = |\{i,j \mid [A]_{i,j} \neq 0\}|$ denote the number of nonzero entries of matrix A. When no explicit base is stated, logarithms in this paper are base 2.

External Memory Model. This model of computation [1] is an abstraction of a two-level memory hierachy: We have an internal memory holding M data items ("words") and a disk of infinite size holding the remaining data. Transfers between internal memory and disk happen in blocks of B words, and a word must be in internal memory to be manipulated. The cost of an algorithm in this model is the number of block transfers (I/Os) done by the algorithm. We will use $\text{sort}(n) = O((n/B)\log_{M/B}(n/B))$ as shorthand for the sorting complexity of n data items in the external memory model and $\tilde{O}(\cdot)$-notation to suppress polylogarithmic factor in input size N and matrix dimension U.

We assume that a word is big enough to hold a matrix element from a semiring as well as the matrix coordinates of that element, i.e., a block holds B matrix elements. We restrict attention to algorithms that work with semiring elements as an abstract type, and can only copy them, and combine them using semiring operations. We refer to this restriction as the *semiring I/O model.* Our upper bound uses a slight extension of this model in which equality check is allowed, which allows us to take advantage of *cancellations*, i.e., inner products in the matrix product that are zero in spite of nonzero elementary products.

The Problem We Solve. Given matrices $A \in R^{U \times U}$ and $C \in R^{U \times U}$ containing $\mathtt{nnz}(A)$ and $\mathtt{nnz}(C)$ non-zero semiring elements, respectively, we wish to output a sparse representation of the matrix product AC in the external memory model. We are dealing with sparse matrices represented as a list of tuples of the form $(i, j, [A]_{i,j})$, where $[A]_{i,j} \in R$ is a (nonzero) matrix entry. To produce output we must call a function $\mathtt{emit}(e)$ for every nonzero entry $(i, j, (AC)_{i,j})$ of AC. We only allow $\mathtt{emit}(\cdot)$ to be called once on each output element, but impose no particular order on the sequence of outputs.

We note that the algorithm could be altered to write the entire output before termination by, instead of calling $\mathtt{emit}(\cdot)$, simply writing the output element to a disk buffer, outputting all $\mathtt{nnz}(AC)$ elements using $O(\mathtt{nnz}(AC)/B)$ additional I/Os. However, in some applications such as database systems (see [5]) there may not be a need to materialize the matrix product on disk, so we prefer the more general method of generating output.

1.2 Related Work

The external memory model was introduced by Aggarwal and Vitter in their seminal paper [1], where they provide tight bounds for a collection of central problems.

An I/O-optimal matrix multiplication algorithm for dense semiring matrices was achieved by Hong and Kung [14]: Group the matrices into $k\sqrt{M} \times k\sqrt{M}$ submatrices where constant k is picked such that three $\sqrt{M} \times \sqrt{M}$ matrices fit into internal memory. This reduces the problem to $O((U^3/\sqrt{M})^3)$ matrix products that fit in main memory, costing $O(M/B)$ I/Os each, and hence $O(U^3/B\sqrt{M})$ in total [10]. Hong and Kung also provided a tight lower bound $\Omega(U^3/B\sqrt{M})$ that holds for algorithms that work over a semiring. (It does not apply to algorithms that make use of subtraction, such as fast matrix multiplication methods, for which the blocking method described above yields an I/O complexity of $U^\omega/(M^{\omega/2-1}B)$ I/Os.)

For sparse matrix multiplication the previously best upper bound [5], shown for Boolean matrix products but claimed for any semiring, is $\tilde{O}(N\sqrt{\mathtt{nnz}(AC)}/BM^{1/8})$.

It seems that this bound requires "no cancellation of terms" (or more specifically, the output sensitivity is with respect to the number of output entries that have a nonzero elementary product). Our new upper bound of this paper improves upon this: The Monte Carlo algorithm of Theorem 1 has strictly lower I/O complexity for the entire parameter space and makes no assumptions about cancellation.

An important subroutine in our algorithm is dense-vector sparse matrix multiplication: For a vector y and sparse matrix S we can compute their product using optimal $\tilde{O}((\mathtt{nnz}(S) + \mathtt{nnz}(y))/B)$ I/Os [6] - this holds for arbitrary layouts of the vector and matrix on disk.

Our algorithm has an interesting similarity to Williams and Yu's recent output sensitive matrix multiplication algorithm [25, Section 6]. Their algorithm works by splitting the matrix product into 4 submatrices of equal dimension, running a randomized test to determine which of these subproblems contain a nonzero entry. Recursing on the non-zero submatrices, they arrive at an output sensitive algorithm. We perform a similar recursion, but the splitting is computed differently in order to recurse in a balanced manner, such that each subproblem at a given level of the recursion outputs approximately the same number of entries in the matrix product.

Size estimation of the number of nonzeros in matrix products was used by Cohen [9,8] to compute the order of multiplying several matrices to minimize the total number of operations. For constant error probability this algorithm uses $O(\varepsilon^{-2}N)$ operations in the RAM model to perform the size estimation. For $\varepsilon > 4/\,\texttt{nnz}(AC)^{1/4}$ Amossen et al [4] improved the running time to be expected $O(N)$ in the RAM model and expected $O(\text{sort}(N))$ in the I/O model. Contrary to the approaches of [4,9,8] our new size estimation algorithm presented in Section 2 is able to deal with cancellation of terms, and it uses $\tilde{O}(\varepsilon^{-3}N/B)$ I/Os. Informally, the main idea of our size estimation algorithm is to multiply a sequence of vectors x with certain properties onto AC but in the computationally inexpensive order $(xA)C$, in order to produce linear sketches of the rows (columns) of AC.

1.3 New Results

We present a new upper bound in the I/O model for sparse matrix multiplication over semirings. Our I/O complexity is at least a factor of roughly $M^{3/8}$ better than that of [5]. We show the following theorem:

Theorem 1. *Let $A \in R^{U\times U}$ and $C \in R^{U\times U}$ be matrices with entries from a semiring R, and let $N = \texttt{nnz}(A)+\texttt{nnz}(C)$, $Z = \texttt{nnz}(AC)$. There exist algorithms (a) and (b) such that:*

(a) emits the set of nonzero entries of AC, and uses $O\left(N^2/(MB)\right)$ I/Os.
(b) emits the set of nonzero entries of AC with probability at least $1-1/U$, using $\tilde{O}\left(N\sqrt{Z}/(B\sqrt{M})\right)$ I/Os.

For every A and C, using $\tilde{O}\left(N/B\right)$ I/Os we can determine with probability at least $1-1/U$ if one of the two I/O bounds is significantly lower, i.e., distinguish between $N\sqrt{Z}/(B\sqrt{M}) > 2N^2/(MB)$ and $2N\sqrt{Z}/(B\sqrt{M}) < N^2/(MB)$.

The above theorem makes no assumptions about cancellation of terms. In particular, $\texttt{nnz}(AC)$ can be smaller than the number of output entries that have nonzero elementary products.

Our second main contribution is a new lower bound on sparse matrix multiplication in the semiring I/O model.

Theorem 2. *For all positive integers N and $Z < N^2$ there exist matrices A and C with $\texttt{nnz}(A), \texttt{nnz}(C) \leq N$, $\texttt{nnz}(AC) \leq Z$, such that computing AC in the semiring I/O model requires $\Omega\left(\min\left(\frac{N^2}{MB}, \frac{N\sqrt{Z}}{\sqrt{M}B}\right)\right)$ I/Os.*

Since we can determine and run the algorithm satisfying the minimum complexity of the lower bound, our bounds match. We note however, that equality tests are disallowed in the lower bound model and allowed in the upper bound model, but we conjecture that allowing equality tests would not weaken the lower bound, making the bounds tight.

2 Matrix Output Size Estimation

We present a method to estimate column/row sizes of a matrix product AC, represented as a sparse matrix. In particular, for a column C_{*k} (or analogously row A_{k*}) we are interested in estimating the number of nonzeros $\texttt{nnz}(A[C]_{*k})$ ($\texttt{nnz}([A]_{k*}B)$). We note that there are no assumptions about (absence of) cancellation of terms in the following. We show the existence of the following algorithms (proof omitted due to space constraints).

Lemma 1. *Let $A \in R^{U\times U}$ and $C \in R^{U\times U}$ be matrices with entries from semiring R, $N = \texttt{nnz}(A) + \texttt{nnz}(C)$ and let $0 < \varepsilon, \delta \leq 1$. We can compute estimates $z_1, \ldots, z_k$ using $\tilde{O}(\varepsilon^{-3}N/B)$ I/Os and $O(\varepsilon^{-3}N \log(U/\delta) \log U)$ RAM operations such that with probability at least $1-\delta$ it holds that $(1-\varepsilon)\,\texttt{nnz}([AC]_{*k}) \leq z_k \leq (1+\varepsilon)\,\texttt{nnz}([AC]_{*k})$ for all $1 \leq k \leq U$.*

Corollary 1. *Let $A \in R^{U\times U}$ and $C \in R^{U\times U}$ be matrices with entries from semiring R, $N = \texttt{nnz}(A) + \texttt{nnz}(C)$ and let $0 < \varepsilon, \delta \leq 1$. We can compute $\hat{Z}$ in $\tilde{O}(\varepsilon^{-3}N/B)$ I/Os and $O(\varepsilon^{-3}N \log(U/\delta) \log U)$ RAM operations such that with probability at least $1-\delta$ it holds that $(1-\varepsilon)\,\texttt{nnz}(AC) \leq \hat{Z} \leq (1+\varepsilon)\,\texttt{nnz}(AC)$.*

At a high level, the algorithm is similar in spirit to Cohen [9,8], but uses linear F_0 sketches (see e.g. [11,15]) that serve the purpose of capturing cancellation of terms.We will make use of a well-known F_0-sketching method [11,16], where $F_0(f)$ denotes the number of non-zero entries in a vector f. Let S be a data stream of items of the form $((i,j),r)$, where $(i,j) \in U \times U$ and $r \in R$. The stream defines a vector indexed by $U \times U$ (which can also be thought of as a matrix), where entry (i,j) is the sum of all ring elements r that occurred with index (i,j) in the stream. For a matrix $S \in R^{U\times U}$ the number of distinct indices is the sum of distinct indices over all column vectors $F_0(S) = \sum_{i\in[U]} F_0(S_{i*})$. One can compute in space $O(\varepsilon^{-3} \log n \log \delta^{-1})$ [11,16] a *linear* sketch over x that can output a number $\hat{z}$, where $(1-\varepsilon)F_0 \leq \hat{z} \leq (1+\varepsilon)F_0$ with constant probability.

High-level algorithm description. We compute a linear sketch F followed by the matrix product $v = FAC$. From v for a given T we can distinguish between a column having more than $(1+\varepsilon)T$ and less than $(1-\varepsilon)T$ nonzero entries - we repeat this procedure for suitable values of T to achieve the final estimate. We use the following distinguishability result:

Fact 3. *([16], Section 2.1) There exists a projection matrix $M \in \{0,1\}^{n\times d}$ such that for each frequency vector $f \in R^{1\times n}$ we can estimate $F_0(f)$ from fM.*

In particular, for fixed $T' > 0$, $0 < \varepsilon', \delta' \leq 1$ with probability $1 - \delta'$ we can distinguish the cases $F_0(f) > (1 + \varepsilon')T'$ and $F_0(f) < (1 - \varepsilon')T'$ using space $d = O(\varepsilon'^{-2} \log \delta'^{-1})$.

We will apply this distinguishability sketch (matrix M from Fact 3) to the columns of the product AC, since $F_0(AC) > (1+\varepsilon)T$ implies $\mathtt{nnz}(AC) > (1+\varepsilon)T$ and analogously for the second case. This follows trivially from the definition of F_0 and the number of nonzeroes in a matrix product. From Fact 3 we have a sketch $F \in \{0,1\}^{d \times U}$ which when multiplied with a matrix $S \in R^{U \times U}$ can distinguish $\mathtt{nnz}(S_{*k}) > (1 + \varepsilon)T$ from $\mathtt{nnz}(S_{*k}) < (1 - \varepsilon)T$ with probability $1 - \delta$ using the columns $[FS]_{*k}$.

3 Cache-Aware Upper Bound

As in the previous section let $A \in R^{U \times U}$ and $C \in R^{U \times U}$ be matrices with entries from a semiring R, and let $N = \mathtt{nnz}(A) + \mathtt{nnz}(C)$ be the input size.

3.1 Output Insensitive Algorithm

We first describe algorithm (a) of Theorem 1, which is insensitive to the number of output entries $\mathtt{nnz}(AC)$. It works as follows: First put the entries of C in column-major order by lexicographic sorting. For every row a_i of A with more than $M/2$ nonzeros, compute the vector-matrix product a_iC in time $\tilde{O}(N/B)$ using the algorithm of [6]. There can be at most $2N/M$ such rows, so the total time spent on this is $\tilde{O}(N^2/(MB))$. The remaining rows of A are then gathered in groups with between $M/2$ and M nonzero entries per group. In a single scan of C (using column-major order) we can compute the product of each such row with the matrix C. The number of I/Os is $O(N/B)$ for each of the at most $2N/M$ groups, so the total complexity is $\tilde{O}(N^2/(MB))$.

3.2 Monte Carlo Algorithm Overview

We next describe algorithm (b) of Theorem 1 The algorithm works by first performing a step of coloring, the purpose of which is to split the matrix product into submatrices, each of which can be computed efficiently. The overall idea is to color matrix rows A using c colors and for each of the c sets of colored rows we color matrix C also using c colors, such that every combination of colored rows from A and colored columns from C yields a low number of non-zero output entries. Then, a "compressed" matrix multiplication algorithm (described by Lemma 2) is used to compute the output entries of every such combination. The number c of colors needed to achieve this, to be specified later, depends on an estimate of $\mathtt{nnz}(AC)$, found using Corollary 1.

A technical hurdle is that there might be rows of A and columns of C that we cannot color because they generate too many entries in the output. However, it turns out that we can afford to handle such rows/columns in a direct way using vector-matrix multiplication.

3.3 Compressed Matrix Multiplication in the I/O Model

Let $\gamma > 0$ be a suitably small constant, and define $r = 4\gamma M/\log U$. We now describe an I/O-efficient algorithm for matrix products AC with $\texttt{nnz}(AC) \leq \gamma M/\log U = r/4$ nonzeros. If A is stored in column-major order and C is stored in row-major order, the algorithm makes just a single scan over the matrices.

The algorithm is a variation of the one found in [19], adapted to the semiring I/O model. Specifically, for some constant ℓ and $t = 1, \ldots, \ell \log U$ let $h_t, h'_t : [U] \to [r]$ be pairwise independent hash functions. The algorithm computes the following $\ell \log U$ polynomials of degree at most $2r$:

$$p_t(x) = \sum_{k=1}^{U} \left(\sum_{i=1}^{U} A_{i,k} x^{h_t(i)} \right) \left(\sum_{j=1}^{U} C_{k,j} x^{h'_t(j)} \right) .$$

It is not hard to see that the polynomial $\sum_{i=1}^{U} A_{i,k} x^{h_t(i)}$ can be computed in a single scan over column i of A, using space r. Similarly, we can compute the polynomial $\sum_{j=1}^{U} C_{k,j} x^{h'_t(j)}$ in space r by scanning row j of C. As soon as both polynomials have been computed, we multiply them and add the result to the sum of products that will eventually be equal to $p_t(x)$. This requires additional space $2r$, for a total space usage of $4r$.

Though a computationally less expensive approach is described in [19], we present a simple method that (without using any I/Os) uses the polynomials $p_t(x)$, $t = 1, \ldots, \ell \log U$, to compute the set of entries in AC with probability $1 - U^{-3}$. For every i and j, to compute the value of $[AC]_{i,j}$ consider the coefficient of $x^{h_t(i)+h'_t(j)}$ in p_t, for $t = 1, \ldots, \ell \log U$. For suitably chosen c, with probability $1 - U^{-5}$ the value $[AC]_{i,j}$ is found in the majority of these coefficients. The majority coefficient can be computed using just equality checks among semiring elements [7]. The analysis in [19] gives us, for a suitable choice of γ and ℓ, the following:

Lemma 2. *Suppose matrix A is stored in column-major order, and C is stored in row-major order. There exists an algorithm in the semiring I/O model augmented with equality test, and an absolute constant $\gamma > 0$, such that if $\texttt{nnz}(AC) < \gamma M/\log U$ the algorithm outputs the nonzero entries of AC with probability $1 - U^{-3}$, using just a single scan over the input matrices.*

3.4 Computing a Balanced Coloring

Let color set S_i contain rows A_{k*} that are assigned color i, and for each color i assigned to rows of A let color set $S_j^{(i)}$ contain columns C_{*k} that are assigned color j. Also, let $A|S_i$ be the input matrix A restricted to contain only elements in rows from S_i (and analogously for C and $S_j^{(i)}$).

The goal of the coloring step is to assign the colors such that for every pair of color sets $(S_i, S_j^{(i)})$, $1 \leq i, j \leq c$ it holds that $\texttt{nnz}((A|S_i)(C|S_j^{(i)})) < \gamma M/\log U$. This can be seen as coloring the rows of A once and the columns of C c times.

Lemma 3. *Let $A \in R^{U\times U}$ and $C \in R^{U\times U}$ be matrices with $N = \mathtt{nnz}(A) + \mathtt{nnz}(C)$ nonzero entries.*
Using $\tilde{O}\left(\frac{N\sqrt{\mathtt{nnz}(AC)}}{B\sqrt{M}}\right)$ I/Os a coloring with $c = \sqrt{\frac{\mathtt{nnz}(AC)\log U}{M}} + O(1)$ colors can be computed that assigns a color to rows of A and for each such color i, assigns colors to columns of C such that:

1. *For every $i, j \in [c]$ it holds that $\mathtt{nnz}\left((A|S_i)(C|S_j^{(i)})\right) < M/\log U$.*
2. *Rows from A and columns from C that are not in some color sets S_i and $S_j^{(i)}$ has had their nonzero output entries emitted.*

Proof. At a high level, the coloring will be computed by recursively splitting the matrix rows in two disjoint parts to form matrices A_1 and A_2 where A_1 contains the nonzeros from the first $t-1$ rows, for some t, and A_2 contains the nonzeros from the last $U-t$ rows. Row number t, the "splitting row", will be removed from consideration by generating the corresponding part of the output using I/O-efficient vector-matrix multiplication. We wish to choose t such that:

I $\mathtt{nnz}(A_1C) \in \left[(1-\log^{-1}U)\,\mathtt{nnz}(AC)/2; (1+\log^{-1}U)\,\mathtt{nnz}(AC)/2\right]$.
II $\mathtt{nnz}(A_2C) \in \left[(1-\log^{-1}U)\,\mathtt{nnz}(AC)/2; (1+\log^{-1}U)\,\mathtt{nnz}(AC)/2\right]$.

And after $\log c + O(1)$ recursive levels of such splits, we will have $O(c)$ disjoint sets of rows from A. For each such set we then compute disjoint column sets of C in the same manner, and we argue below that this gives us subproblems with output size $\mathtt{nnz}(AC)/c^2 = M/\log U$, where each subproblem corresponds exactly to a pair of color sets as described above.

In order to compute the row number t around which to perform the split, we invoke the estimation algorithm from Corollary 1 with $\varepsilon = \log^{-1}U$ such that for every row in $[AC]_{k*}$ we have access to an estimate $\hat{z}_k$ where it holds with probability at least $1-U^{-l}$ (for fixed $l > 0$ chosen to get sufficiently low error probability):

$$\hat{z}_k \in \left[(1-\log^{-1}U)\,\mathtt{nnz}([AC]_{k*})/2; (1+\log^{-1}U)\,\mathtt{nnz}([AC]_{k*})/2\right]. \qquad (1)$$

In particular for any set of rows r we have that

$$(1-\log^{-1}U)\,\mathtt{nnz}\left(\sum_{i\in r}[AC]_{i*}\right) \le \sum_{i\in r}\hat{z}_i \le (1+\log^{-1}U)\,\mathtt{nnz}\left(\sum_{i\in r}[AC]_{i*}\right). \quad (2)$$

We will now argue that if we can create a split of the rows such that (I) and (II) hold, then when the splitting procedure terminates after $\log c + O(1)$ recursive levels, we have that for each pair of colors it is the case that $(A|S_i)(C|S_j^{(i)}) < M/\log N$. Consider the case where each split is done with the maximum positive error possible, i.e., on recursive level q we have divided the $\mathtt{nnz}(AC)$ nonzeros into subproblems where each are of size at most $\mathtt{nnz}(AC)(1/2 + 1/(2\log U))^q$.

After $\log c + O(1)$ recursive levels we have subproblem size:

$$\begin{aligned}\mathtt{nnz}(AC)\left(\frac{1}{2}+\frac{1}{2\log U}\right)^{\log c^2} &= \mathtt{nnz}(AC)2^{-\log c^2}\left(1+\frac{1}{\log U}\right)^{\log c^2} \\ &\leq \mathtt{nnz}(AC)2^{-\log c^2}e^{\frac{\log c^2}{\log U}} &(3)\\ &\leq \mathtt{nnz}(AC)O(1)/c^2 = O(M/\log U) &(4)\end{aligned}$$

The main observation to see that we get the right subproblem size as in (4) is that for each recursion we decrease the output size by a factor $\Omega(c)$. For (3) we use $(1+1/x)^y \leq e^{y/x}$, and (4) follows from $\mathtt{nnz}(AC) \leq U^2$ and the definition of c. The analysis for the case where each split is done with the maximum negative error possible is analogous and thus omitted.

We will now argue that with access to the $\hat{z}_i$ estimates as in (1) we can always construct a split such that (I) and (II) hold. Let partitions 1 and 2 be denoted P_1 and P_2 and $\hat{z} = \sum_i \hat{z}_i$ be the estimate of the total number of outputs for the current subproblem. Create P_1 by examining rows $[A]_{k*}$ one at a time. If the estimated number of nonzeros of $P_1 \cup [A]_{k*}$ is less than $\hat{z}/2$ then add $[A]_{k*}$ to P_1. Otherwise perform dense-vector sparse-matrix multiplication $[A]_{k*}C$ using $\tilde{O}(\mathtt{nnz}(C)/B)$ I/Os [6]and emit every nonzero of that product - this eliminates the row vector $[A]_{k*}$ from matrix A as all outputs generated by row $[A]_{k*}$ has now been emitted. Because of (2) we have that the remaining rows of A can now be placed in partition P_2 and the sum of their outputs will be at most $(1+\log^{-1}U)\,\mathtt{nnz}(AC)/2$. The procedure and analysis is equivalent for the case of columns. From (4) we had that even with splits of $\mathtt{nnz}(AC)(1/2+\log(U)/2)$ nonzeros then the subproblem size is the desired $O(M/\log U)$ after all $\log c^2$ splits are done.

In terms of I/O complexity consider first the coloring of all rows in A. First we perform the size estimates of Corollary 1 in $\tilde{O}(N/B)$ such that we know where to split. Then we perform c splits and each split also emits the output entries for a specific row using dense-vector sparse-matrix multiplication, hence this split takes $c\,\mathtt{nnz}(C) = \tilde{O}\left(cN/B\right)$ I/Os. Finally for each of the c sets of rows of A we partition columns of C in the same manner, first by invoking c size estimations taking $\tilde{O}(N/B)$ due to the sum of the nonzeros in the c subproblems being at most N. Then for each of the c row sets we perform c splits and output a column from C. This step takes time $\tilde{O}(cN/B)$ and hence in total we use $\tilde{O}(cN/B + 2N/B) = \tilde{O}\left(\frac{N\sqrt{\mathtt{nnz}(AC)}}{B\sqrt{M}}\right)$. ∎

3.5 I/O Complexity Analysis

Next, we will use Lemma 3 for the algorithm that shows part (b) of Theorem 1.

We summarize the steps taken and their cost in the external memory model.

Proof. (Theorem 1, part (b)) The algorithm first estimates $\mathtt{nnz}(AC)$ with parameters $\varepsilon = 1/\log N$ and $\delta = 1/U$ which by Corollary 1 uses $\tilde{O}(N/B)$ I/Os. We then perform the coloring, outputting some entries of AC and dividing the remaining entries into c^2 balanced sets for $c = \sqrt{\frac{\texttt{nnz}(AC)\log U}{M}} + O(1)$. By Lemma 3 this uses $\tilde{O}\left(\frac{N\sqrt{\texttt{nnz}(AC)}}{B\sqrt{M}}\right)$ I/Os. Finally we invoke the compressed matrix multiplication algorithm from Lemma 2 on each subproblem. This is possible since each subproblem has at most $\gamma M/\log U$ nonzeros entries in the output. The total cost of this is $O(cN/B)$ I/Os, since each nonzero entry in A and C is part of at most c products, and the cost of each product is simply the cost of scanning the input. ■

4 Lower Bound

Our lower bound generalizes that of Hong and Kung [14] on the I/O complexity of dense matrix multiplication. We extend the technique of [14] while taking inspiration from lower bounds in [13,21,20]. The closest previous work is the lower bound in [20] on the I/O complexity of triangle enumeration in a graph, but new considerations are needed due to the fact that cancellations can occur.

Like the lower bound of Hong and Kung [14], our lower bound holds in a *semiring model* where:

– A memory block holds up to B matrix entries (from the semiring), and internal memory can hold M/B memory blocks.
– Semiring elements can be multiplied and added, resulting in new semiring elements.
– No other operations on semiring elements (e.g. division, subtraction, or equality test) are allowed.

The model allows us to store sparse matrices by listing just non-zero matrix entries and their coordinates. We note that our algorithm respects the constraints of the semiring model with one small exception: It uses equality checks among semiring elements. We require the algorithm to work for every semiring, and in particular over fields of infinite size such as the real numbers, and for arbitrary values of nonzero entries in A and C. Since only addition and multiplication are allowed, we can consider each output value as a polynomial over nonzero entries of the input matrices. By the Schwartz-Zippel theorem [17, Theorem 7.2] we know that two polynomials agree on all inputs if and only if they are identical. Since we are working in the semiring model, the only way to get the term $A_{i,k}C_{k,j}$ in an output polynomial is to directly multiply these input entries. That means that to compute an output entry $[AC]_{i,j}$ we need to compute a polynomial that is identical to the sum of elementary products $\sum_k A_{i,k}C_{k,j}$. It is possible that the computation of this polynomial involves other nonzero terms, but these are required to cancel out.

We now argue that for every N and Z there exist matrices A and C with $\texttt{nnz}(A)+\texttt{nnz}(C) = \Theta(N)$ and $\texttt{nnz}(AC) = \Theta(Z)$, for which every execution of an external memory algorithm in the semiring model must use $\Omega\left(\frac{N}{B}\min\left(\sqrt{\frac{Z}{M}}, \frac{N}{M}\right)\right)$

I/Os. Our lower bound holds for the *best possible execution*, i.e., even if the algorithm has been tailored to the structure of the input matrices.

The hard instance for the lower bound is a dense matrix product, which maximizes the number of elementary products. In particular, since we ignore constant factors we may assume that $\sqrt{Z}$ and $N/\sqrt{Z}$ are integers. Let A be a $(\sqrt{Z})$-by-$(N/\sqrt{Z})$ dense matrix, and let C be a $(N/\sqrt{Z})$-by-$(\sqrt{Z})$ dense matrix. Without loss of generality, every semiring element that is stored during the computation is either: 1) An input entry, or 2) Part of a sum that will eventually be emitted as the value of a unique nonzero element $[AC]_{i,j}$.

This is because these are the only values that can be used to compute an output entry (making use of the fact that additive and multiplicative inverses cannot be computed). This implies that every output entry can be traced through the computation, and it is possible to pinpoint the time in the execution where an elementary product is computed and stored in internal memory.

We use the following lemma from [13]:

Lemma 4. *In space M the number of elementary products that can be computed and stored is at most $M^{3/2}$.*

Following [20], observe that any execution of an I/O efficient algorithm can be split into *phases* of M/B I/Os. By doubling the memory size to $2M$ we find an equivalent execution where every read I/O happens at the beginning of the phase (before any processing takes place), and every write I/O happens at the end of the phase. For every phase we can therefore identify the set of at most $2M$ input and output entries that involved in the phase.

If all values needed for emitting a particular output entry are present in a phase there may not be any storage location that can be associated with it. We first account for such *direct* outputs: Each direct output requires two vectors of length $N/\sqrt{Z}$ to be stored in main memory. In each phase we can store at most $M\sqrt{Z}/N$ such vectors, resulting in at most M^2Z/N^2 output pairs. So the number of phases needed to emit, say, $Z/2$ outputs would be at least $(N/M)^2$, using $N^2/(MB)$ I/Os. This means that to output a substantial portion of AC in this way we need at least this number of I/Os.

Next, we focus on output entries for which an elementary product is written to disk in some phase. By Lemma 4 the number of elementary products computed and stored is at most $(2M)^{3/2}$. If the total number of elementary products is p then we need at least $p/(2M)^{3/2}$ phases of M/B I/Os each. Considering $Z/2$ output entries in our hard instance, these contain $N\sqrt{Z}/2$ elementary products.

Since $Z/2$ outputs are needed either in the direct or the indirect way, the number of I/Os needed becomes the minimum of the two lower bounds we get Theorem 2.

References

1. Aggarwal, A., Vitter, J.S.: The Input/Output complexity of sorting and related problems. Commun. ACM 31(9), 1116–1127 (1988)
2. Alon, N., Yuster, R., Zwick, U.: Color-coding. J. ACM (4), 844–856

3. Alon, N., Yuster, R., Zwick, U.: Finding and counting given length cycles. Algorithmica 17(3), 209–223 (1997)
4. Amossen, R.R., Campagna, A., Pagh, R.: Better size estimation for sparse matrix products. APPROX/RANDOM 2010, 406–419 (2010)
5. Amossen, R.R., Pagh, R.: Faster join-projects and sparse matrix multiplications. In: ICDT 2009, pp. 121–126. ACM (2009)
6. Bender, M., Brodal, G., Fagerberg, R., Jacob, R., Vicari, E.: Optimal sparse matrix dense vector multiplication in the I/O-model. TCS 47(4), 934–962 (2010)
7. Boyer, R.S., Moore, J.S.: MJRTY - A fast majority vote algorithm. Technical Report AI81-32 (February 1, 1981)
8. Cohen, E.: Estimating the size of the transitive closure in linear time. In: SFCS 1994, pp. 190–200. IEEE Computer Society, Washington, DC (1994)
9. Cohen, E.: Structure prediction and computation of sparse matrix products. Journal of Combinatorial Optimization 2(4), 307–332 (1998)
10. Demaine, E.D.: Cache-oblivious algorithms and data structures. In: Lecture Notes from the EEF Summer School on Massive Data Sets, June 27-July 1 (2002)
11. Flajolet, P., Martin, G.N.: Probabilistic Counting Algorithms for Data Base Applications. Journal of Computer and System Sciences
12. Greiner, G., Jacob, R.: The I/O complexity of sparse matrix dense matrix multiplication. In: López-Ortiz, A. (ed.) LATIN 2010. LNCS, vol. 6034, pp. 143–156. Springer, Heidelberg (2010)
13. Irony, D., Toledo, S., Tiskin, A.: Communication lower bounds for distributed-memory matrix multiplication. JPDC 64(9), 1017–1026 (2004)
14. Jia-Wei, H., Kung, H.T.: I/O complexity: The red-blue pebble game. In: STOC 1981, pp. 326–333. ACM (1981)
15. Kane, D.M., Nelson, J., Woodruff, D.P.: An optimal algorithm for the distinct elements problem. In: PODS 2010, pp. 41–52. ACM (2010)
16. McGregor, A.: Algorithms for Signals, book draft (2013)
17. Motwani, R., Raghavan, P.: Randomized Algorithms. Cambridge University Press, New York (1995)
18. Mulmuley, K., Vazirani, U., Vazirani, V.: Matching is as easy as matrix inversion. Combinatorica 7(1), 105–113 (1987)
19. Pagh, R.: Compressed matrix multiplication. ACM Trans. Comput. Theory 5(3), 9:1–9:17 (2013)
20. Pagh, R., Silvestri, F.: The input/output complexity of triangle enumeration. arXiv preprint arXiv:1312.0723 (2013)
21. Pietracaprina, A., Pucci, G., Riondato, M., Silvestri, F., Upfal, E.: Space-round tradeoffs for mapreduce computations. In: ICS, pp. 235–244. ACM (2012)
22. Rabin, M.O., Vazirani, V.V.: Maximum matchings in general graphs through randomization. J. Algorithms 10(4), 557–567 (1989)
23. Strassen, V.: Gaussian elimination is not optimal. Numerische Mathematik 13(4), 354–356 (1969)
24. van Dongen, S.: Graph Clustering by Flow Simulation. PhD thesis, University of Utrecht (2000)
25. Williams, R., Yu, H.: Finding orthogonal vectors in discrete structures. In: SODA 2014, ch. 135, pp. 1867–1877 (2014)
26. Williams, V.V.: Multiplying matrices faster than coppersmith-winograd. In: STOC 2012, pp. 887–898. ACM, New York (2012)

Faster FPTASes for Counting and Random Generation of Knapsack Solutions

Romeo Rizzi[1] and Alexandru I. Tomescu[2]

[1] Department of Computer Science, University of Verona, Italy
[2] Helsinki Institute for Information Technology HIIT,
Department of Computer Science, University of Helsinki, Finland
romeo.rizzi@univr.it, tomescu@cs.helsinki.fi

Abstract. We give faster and simpler fully polynomial-time approximation schemes (FPTASes) for the #P-complete problem of counting 0/1 Knapsack solutions, and for its random generation counterpart. Our method is based on dynamic programming and discretization of large numbers through floating-point arithmetic. We improve both deterministic counting FPTASes in (Gopalan et al., FOCS 2011), (Štefankovič et al., SIAM J. Comput. 2012) and the randomized counting and random generation algorithms in (Dyer, STOC 2003). We also improve the complexity of the problem of counting 0/1 Knapsack solutions in an arc-weighted DAG.

1 Introduction

In the #P-complete problem of counting 0/1 Knapsack solutions, the input consists of a sequence of n nonnegative integer weights $w_1, \ldots, w_n$ and an integer C, and we have to find the number of subsets (subsets of indices) whose weights add up to at most C. In its extension to a DAG with nonnegative integer arc weights, we are given two vertices s and t, and a capacity C, and we are asked to count how many paths exist from s to t of total weight at most C.

The 0/1 Knapsack counting problem is the classical example of a problem where the difficulty lies entirely in the size of the numbers of solutions. After initial efforts, including a randomized subexponential time algorithm [3] (based on near-uniform sampling of feasible solutions by a random walk), and a fully polynomial-time randomized approximation scheme[1] (FPRAS) [8] (based on a rapidly mixing Markov chain), this problem was shown to admit a dynamic programming solution [2].

The solution in [2] uses a linear discretization scheme of the large numbers, which gives an $O(n)$-factor approximation algorithm. This leads to a randomized sampling algorithm generating a solution with probability at least $1 - e^{-1}$, using $O(n^2)$ arithmetic operations, once a supporting table is computed with $O(n^3)$

[1] A *fully polynomial-time (randomized) approximation scheme*, FPTAS (FPRAS), is an algorithm that estimates the exact solution within relative error $1 \pm \varepsilon$, in time polynomial in the input size and in $1/\varepsilon$.

arithmetic operations. However, these operations are on $O(n)$-bit numbers (of possible solutions), and translate to a randomized algorithm generating ν uniform samples in time $O(n^4 + \nu n^3)$ with probability at least $1 - e^{-\Omega(n)}$. This can be refined using rejection sampling into an FPRAS of complexity $O(n^4 + n^3\varepsilon^{-2})$.[2]

Recently, two deterministic FPTASes for this problem appeared. One is [4] (see also the combined paper [5]), of complexity $O(n^3\varepsilon^{-1}\log(n/\varepsilon)\log C)$. It uses read-once branching programs and insight from [6] to approximate the solution space; it can be seen as a derandomized version of the result of [2]. Another FPTAS is [10] (see also the combined paper [5]), of complexity $O(n^3\varepsilon^{-1}\log(n/\varepsilon))$. It is based on dynamic programming and geometric approximation of the numbers.

The problem of counting 0/1 Knapsack solutions has been extended in [7] to a DAG with nonnegative arc weights, in connection to various applications in biological sequence analysis (see the references in [7]). Given two vertices s and t, we have to count the number of s,t-paths of weight at most C. This is clearly a generalization of counting 0/1 Knapsack solutions, since given an instance $w_1, \ldots, w_n$ and C it suffices to construct the DAG having $\{v_0, \ldots, v_n\}$ as vertex set, $s = v_0$, $t = v_n$, and for each $i \in \{1, \ldots, n\}$, there are two parallel arcs from v_{i-1} to v_i, with weights 0 and w_i, respectively. In [7], the technique of [10] is extended to this problem, and an FPTAS running in time $O(mn^3\log(n)\varepsilon^{-1}\log(n/\varepsilon))$ is obtained (inaccurately, the factor $\log(n/\varepsilon)$ is missing from their stated complexity bound). Our results are the following ones:

Theorem 1. *Let $w_1, \ldots, w_n$ and C be an input to the 0/1 Knapsack counting problem, and let Z be the number of solutions. For any $0 < \varepsilon \le 1$ we can deterministically compute a floating-point number Z' with a $\lceil \log n \rceil$-bit exponent and a $\lceil \log n + \log(1/\varepsilon) + 1 \rceil$-bit mantissa satisfying $(1-\varepsilon)Z \le Z' \le Z$, in time*

$$O(n^3\varepsilon^{-1}\lceil \log(1/\varepsilon)/\log n \rceil),$$

assuming unit cost additions and comparisons on numbers with $O(\log C)$ bits.

Theorem 2. *Let $w_1, \ldots, w_n$ and C be an input to the 0/1 Knapsack problem. For any $0 < \varepsilon \le 1$, we can generate a solution with a probability different from the uniform one by a relative factor $(1-\varepsilon)^{\pm 1}$, in expected time*

$$O(n \log(n/\varepsilon)).$$

This assumes data structures occupying $O(n^4\varepsilon^{-1}\lceil \log(1/\varepsilon)/\log n \rceil W)$ bits (W is the word size) and computable in $O(n^4\varepsilon^{-1}\lceil \log(1/\varepsilon)/\log n \rceil M(\lceil \log(1/\varepsilon)/\log n \rceil))$ time, where $M(x)$ denotes the multiplicative slowdown of multiplying two x-bit numbers.[3]

Our supporting data structures are computed slower than in [2]; however, the time needed for generating one solution is smaller by orders of magnitude. The next result offers a generalization of Thm. 1 since, as explained above, a 0/1 Knapsack instance can be embedded into a DAG with $m = O(n)$ arcs.

[2] We assume, like in [3,5,10], that additions of $O(\log C)$-bit numbers take unit time.

[3] With the Schönhage-Strassen method [9], two x-bit numbers can be multiplied in time $x \log x \log\log x$, thus $M(x) = \log x \log\log x$.

Theorem 3. *Let G be a DAG with n vertices, m arcs with nonnegative weights, and let s and t be two of its vertices. For any C and any $0 < \varepsilon \leq 1$, we can deterministically compute an $1 - \varepsilon$ approximation of the number of s,t-paths of weight at most C in G, in time*

$$O\left(mn^2 \log\left(\frac{m}{n}\right) \varepsilon^{-1} \lceil \log(1/\varepsilon) / \log n \rceil\right),$$

assuming unit cost additions and comparisons on numbers with $O(\log C)$ bits.

2 Previous Work and Outline of the Proofs

The solution in [10] is based on the decomposition $\tau(i, a) :=$ the smallest capacity c such that there exist at least a solutions to the 0/1 Knapsack problem with weights $w_1, \ldots, w_i$ and capacity c. The second parameter of τ is then approximated according to a geometric progression of ratio $Q = 1 + \varepsilon/(n+1)$. The approximated table is computed by dynamic programming using the recurrence:

$$T[i,j] = \min_{\alpha \in [0,1]} \max \begin{cases} T[i-1, \lfloor j + \log_Q \alpha \rfloor], \\ T[i-1, \lfloor j + \log_Q (1-\alpha) \rfloor] + w_i. \end{cases} \tag{1}$$

Thanks to the geometric discretization, the second parameter of T takes $O(n^2\varepsilon^{-1})$ values. Finding the minimum over $\alpha \in [0,1]$ is reducible to two binary searches in row $i-1$ of T, due to its monotonicity. The computation of $\lfloor \log_Q \alpha \rfloor$ and $\lfloor \log_Q (1-\alpha) \rfloor$ is then shown to be doable in time $O(\log(n/\varepsilon))$.

We consider instead the classic problem decomposition $s(i,c) :=$ the number of 0/1 Knapsack solutions that use a subset of the items $w_1, \ldots, w_i$, and their weights sum up to at most $c \leq C$. Notice that s and τ are dual, in the sense

$$\tau(i,a) = \min\{c \,:\, s(i,c) \geq a\} \text{ and } s(i,c) = \max\{a \,:\, \tau(i,a) \leq c\}. \tag{2}$$

Table s can be computed by a dynamic programming algorithm obtained from the recurrence

$$s(i,c) = s(i-1,c) + s(i-1, c-w_i). \tag{3}$$

We also approximate the number of solutions, which are now the values of s. However, we approximate them using binary floating-point numbers. The difference with respect to a standard computer implementation is that we need as many bits for the exponent as to represent them exactly, and as many bits for the mantissa as to guarantee the required approximation. For counting 0/1 Knapsack solutions, we need $\lceil \log n \rceil$ bits for the exponent and $\lceil \log(n/\varepsilon) \rceil + 1$ bits for the mantissa. The main advantage of such an approximation scheme for s is that it avoids computing values such as $\lfloor \log_Q \alpha \rfloor$ (and the associated complexity analysis), and requires a much simpler approximation analysis.

Second, we are able to avoid a minimization as in (1), and thus avoid binary search. The idea is to store each row of s as a list. We prune entries with the same value in each list (leading to $O(n^2/\varepsilon)$ different entries in each list), and compute a list by two linear scans of the previous one.

Third, having the table s explicitly, we can implement a random generation algorithm which works by tracing back probabilistically a random solution from $s(n, C)$. At each step i, it throws a dice with two faces of sizes $s(i-1, c)$ and $s(i-1, c-w_i)$, where c is the capacity available for the remaining first i items. In order to guarantee that the sampling distribution differs from the uniform one by a factor $(1-\varepsilon)^{\pm 1}$, we need another $\lceil \log n \rceil$ bit for the mantissa of our approximated floating-point numbers, as this algorithm makes n choices. Since we represent the table s as a collection of lists, we need to keep, for every entry of a list, back-pointers to the corresponding two entries in the previous list. Each such a pointer occupies W bits, where W is the word size.

Notice that the possibility of doing random generation presented itself also in [10]. First, one needs to decrease Q to $Q = 1 + \varepsilon/(n(n+1))$ (as we do by increasing the length of the mantissa). Thanks to equations (2), one could employ the same probabilistic trace back, by using the approximated table T as black box, and decoding each necessary value of s from T. This can be done in time $O(\log(n/\varepsilon))$ by doing binary search in the corresponding row of T, which adds another factor $\log(n/\varepsilon)$ to the construction time. However, this can be avoided by similarly storing back-pointers from each entry of T to the corresponding two entries in the previous row of T, which are obtained when having found the minimum over $\alpha \in [0, 1]$. Otherwise put, the two faces of the approximated dice will have sizes $\lfloor j + \log_Q \alpha \rfloor$ and $\lfloor j + \log_Q (1-\alpha) \rfloor$, where α minimizes (1).

A further issue is rolling this dice in (expected) time proportional to the number of bits of the two faces of the approximated dice. In our case, since the two faces are floating-point numbers, we can easily solve this by generating a random floating-point number x as follows. We generate a sequence of bits until seeing the first bit equal to '1'. The expected number of bits until this happens is 2, thus this charges only a tiny $O(1)$ term to the expected value of the running time of rolling one dice. At this point, we know the exponent of x, and it is sufficient to continue generating only the remaining bits of the mantissa of x, and check whether x is smaller than the ratio between the approximations of $s(i-1, c-w_i)$ and $s(i, c)$. Moreover, our improvement and simplification obtained by Thm. 1 preserves itself in this random generation algorithm.

Recurrence relation (3) can be extended to a DAG, and thus we can analogously obtain a counting FPTAS. Another improvement with respect to [7] lies in organizing the computation in sequences of $O(n \log(\frac{m}{n}))$ successive additions, so that we need floating-point numbers with only $\lceil \log(n \log(\frac{m}{n})/\varepsilon) \rceil + 1$ bits for the mantissa, and $\lceil \log n \rceil$ bits for the exponent.

3 Approximation by Floating-Point Numbers

In this paper, floating-point arithmetic with base 2 is sufficient, as it also has the advantage of being immediately implementable on a computer for small enough instances. Floating-point arithmetic, and the inherent accuracy analysis issues, have a long history in numerical computation. Another recent application of floating-point arithmetic to approximate counting problems was in [1] in connection with uniform random generation of decomposable structures by partial approximate counts. Moreover, observe that, conceptually, floating-point arithmetic can be seen as an effective combination of the geometric discretization of [10], through the exponent, and of the linear discretization of [3], through the mantissa.

Throughout this paper, we assume that the problem instances consist of n objects (0/1 Knapsack instances with n objects, DAGs with n vertices). Let $c \geq 1$ be such that the maximum numerical value of a particular counting problem is $2^{n^c} - 1$ (that is, it can be represented with n^c bits). Any number $x \in \{0, \dots, 2^{n^c} - 1\}$ can be written as

$$x = x_1 2^{p-1} + x_2 2^{p-2} + \dots + x_{p-1} 2^1 + x_p 2^0 = 2^p \left(x_1 2^{-1} + x_2 2^{-2} + \dots + x_p 2^{-p} \right),$$

where $1 \leq p \leq n^c$, $x_1 = 1$, and $x_i \in \{0, 1\}$, for $i \in \{2, \dots, p\}$. Under floating-point arithmetic terminology, p is called the *exponent* of x, and the binary string $x_1 x_2 \dots x_p$ is called its *mantissa.*

We will approximate x as a floating-point number which has $c \log n$ bits dedicated to store its exponent p exactly, but only t bits dedicated to store the first t bits of its mantissa; that is, we approximate x by the number

$$\langle x \rangle_{c \log n, t} := 2^p \left(x_1 2^{-1} + x_2 2^{-2} + \dots + x_t 2^{-t} \right).$$

We will often drop the subscript $c \log n, t$ when this will be clear from the context. For sure, we will choose $t \geq c \log n$, since the contrary cannot help.

For every $0 \leq x < 2^{n^c}$, it holds that

$$(1 - 2^{1-t}) x \leq \langle x \rangle_{c \log n, t} \leq x. \tag{4}$$

Let $\underline{x}$ and $\underline{y}$ be two floating-point numbers with $c \log n$ bits for the exponent and t bits for the mantissa. We denote the sum $\langle \underline{x} + \underline{y} \rangle$ by $\underline{x} \oplus \underline{y}$. We assume that we can compute $\underline{x} \oplus \underline{y}$ with a bit complexity of $O(c \log n + t) = O(t)$; if additions on $O(\log n)$-bit numbers take unit time, then we assume we can compute $\underline{x} \oplus \underline{y}$ with a word complexity of $O(t / \log n)$.

If $x, y \in \{0, \dots, 2^{n^c} - 1\}$ are such that $x + y \in \{0, \dots, 2^{n^c} - 1\}$, and $\underline{x}, \underline{y}$ are two floating-point numbers with $c \log n$ bits for the exponent and t bits for the mantissa such that

$$(1 - 2^{1-t})^i x \leq \underline{x} \leq x, \text{ and } (1 - 2^{1-t})^j y \leq \underline{y} \leq y,$$

for some integers $i, j \geq 0$, then by (4) the following inequality holds

$$(1 - 2^{1-t})^{1 + \max(i,j)} (x + y) \leq \underline{x} \oplus \underline{y} \leq x + y, \tag{5}$$

For each particular problem, we will choose t as a function of n and of the error factor ε, $0 < \varepsilon \leq 1$. For the problem of counting 0/1 Knapsack solutions, $c = 1$ and $t(n, \varepsilon) = 1 + \lceil \log(n/\varepsilon) \rceil$, while for its extension on a DAG, $c = 1$ and $t(n, \varepsilon) = 1 + \lceil \log(n \log(\frac{m}{n})/\varepsilon) \rceil$. For the random generation of 0/1 Knapsack solutions, $c = 1$ and $t(n, \varepsilon) = 1 + \lceil \log(n^2/\varepsilon) \rceil$.

Algorithm 1. ApproximateCount($w_1, \ldots, w_n, C$)
An FPTAS for counting 0/1 Knapsack solutions.

```
1 Notation: $\underline{s}(i, c) := \max\{r : [c', \underline{r}] \in list(i), c' \leq c\}$;
2 insert the pair $[0, 1]$ into $list(0)$;
3 for $i = 1$ to $n$ do
4     construct the bimotonotic $list'(i)$ containing, for each $[c, \underline{r}]$ in $list(i - 1)$, the
      two pairs:
5         • $[c, \underline{r} \oplus \underline{s}(i - 1, c - w_i)]$;
6         • $[c + w_i, \underline{s}(i - 1, c + w_i) \oplus \underline{r}]$;
7     obtain $list(i)$ by scanning $list'(i)$ and dropping a pair if the previous one
      has the same second component;
8 return $\underline{s}(n, C)$.
```

4 Counting 0/1 Knapsack Solutions

The classic pseudo-polynomial algorithm for counting 0/1 Knapsack solutions defines $s(i, c)$ as the number of Knapsack solutions that use a subset of the items $\{1, \ldots, i\}$, of weight at most $c \in \{0, \ldots, C\}$, and computes these values $s(i, c)$ by dynamic programming, using the recurrence

$$s(i, c) = s(i - 1, c) + s(i - 1, c - w_i), \tag{6}$$

where $s(0, c) = 1$ for any $c \geq 0$, and $s(i, c) = 0$, for any $i \in \{1, \ldots, n\}$ and $c < 0$. Indeed, we either use only a subset of items from $\{1, \ldots, i - 1\}$ whose weights sum up to c, or use item i of weight w_i and a subset of items from $\{1, \ldots, i - 1\}$ whose weights sum up to $c - w_i$. This DP algorithm executes nC additions on n-bit numbers and its complexity is $O(C\,n^2)$. When $C \leq n$, this is $O(n^3)$, whence $n \leq C$ will be assumed in the following. We will assume, like in [10], that additions and comparisons on numbers with $O(\log C)$ bits have unit cost, which implies the same on $O(\log n)$-bit numbers.

We use relation (6) to count, but our numbers, for any $0 < \varepsilon \leq 1$, are approximate floating-point numbers with $\log n$ bits for the exponent, and $1 + \lceil \log(n/\varepsilon) \rceil$ bits for the mantissa (we can assume for simplicity that a solution using all n objects has cost greater than C, so that $s(i, c) < 2^n$ for all $i \in \{1, \ldots, n\}$, $c \in \{0, \ldots, C\}$). By the above assumption, we have that additions and comparisons of these floating-point numbers on $O(\log(n/\varepsilon))$ bits take time $O(\lceil \log(n/\varepsilon)/\log n \rceil) = O(\lceil \log(1/\varepsilon)/\log n \rceil)$.

For every $i \in \{0, \ldots, n\}$ we keep a list, $list(i)$, whose entries are pairs of the form $[c, \underline{r}]$, where c is a capacity in $\{0, \ldots, C\}$ and $\underline{r}$ is an approximate floating-point number of solutions. We will refer to the set of first components of the pairs in $list(i)$ as the *capacities in* $list(i)$. Having $list(i)$, for every $c \in \{0, \ldots, C\}$ we define $\underline{s}(i, c) := \max\{\underline{r} : [c', \underline{r}] \in list(i), c' \leq c\}$, where the maximum of an empty set is taken to be 0.

The first list, $list(0)$, consists of the single pair $[0, 1]$. After this initialization, while computing $list(i)$ from $list(i-1)$, we maintain the following two invariants:

(I$_1$) *list*(i) is strictly increasing on both components;
(I$_2$) $(1-\varepsilon/n)^i\, s(i,c) \le \underline{s}(i,c) \le s(i,c)$, for every $c \in \{0,\ldots,C\}$.

Note that Property (I$_1$) implies that the length of *list*(i) is at most the total number of floating-point numbers that can be represented with $\lceil \log n\rceil + \lceil \log(n/\varepsilon)\rceil + 1$ bits, that is $O(n^2/\varepsilon)$.

We obtain *list*(i) by first building the bimonotonic (i.e., nondecreasing on both components) list *list*$'(i)$ which, for every capacity c in *list*$(i-1)$, contains the following two pairs:

$$[c, \underline{s}(i-1,c) \oplus \underline{s}(i-1,c-w_i)] \text{ and } [c+w_i, \underline{s}(i-1,c+w_i) \oplus \underline{s}(i-1,c)]. \quad (7)$$

It may turn out that *list*$'(i)$ contains distinct pairs having the same second component. Therefore, in order to assure Property (I$_1$), we obtain *list*(i) by pruning away from *list*$'(i)$ those pairs $[c_2, \underline{r}]$ when another pair $[c_1, \underline{r}]$ with $c_1 \le c_2$ is present. We summarize this procedure as Algorithm 1. Lemma 1 below shows that we can efficiently construct *list*$'(i)$; the idea of the proof is to do two linear scans of *list*(i), each with two pointers.

Lemma 1. *We can compute list$'(i)$ and list(i) from list$(i-1)$ in time* $O(n^2\varepsilon^{-1}\lceil \log(1/\varepsilon)/\log n\rceil)$.

Proof. At a generic step $i \in \{1,\ldots,n\}$, we compute *list*$'(i)$ as follows. We construct two auxiliary lists of pairs, *back*(i) and *forw*(i). For every capacity c in *list*$(i-1)$, the list *back*(i) will contain the pairs $[c, \underline{s}(i-1,c) \oplus \underline{s}(i-1,c-w_i)]$, and the list *forw*$(i)$ will contain the pairs $[c+w_i, \underline{s}(i-1,c+w_i) \oplus \underline{s}(i-1,c)]$. List *list*$'(i)$ is now obtained by merging in a unique sorted list the lists *back*(i) and *forw*(i).

In order to compute *forw*(i), proceed as follows (the computation of *back*(i) is entirely analogous). Keep two pointers *left* and *right* in *list*$(i-1)$. Pointer *left* is initially set to the first pair in *list*$(i-1)$, say $[c, \underline{r}]$. Pointer *right* is also set to the first pair in *list*$(i-1)$, but starts scanning *list*$(i-1)$ until reaching a pair $[c_1, \underline{r}_1]$, such that $c_1 + w_i \ge c$ and either $[c_1, \underline{r}_1]$ is the last pair in *list*$(i-1)$, or $[c_1, \underline{r}_1]$ is immediately followed by a pair $[c_2, \underline{r}_2]$ with the property $c + w_i < c_2$. Append the pair $[c + w_i, \underline{r}_1 \oplus \underline{r}]$ at the end of *forw*(i), and advance pointer *left* to the next pair in *list*$(i-1)$; repeat the above procedure, by advancing pointer *right* to the corresponding pair, and inserting a new resulting pair in *forw*(i). This is repeated until pointer *left* reaches the end of *list*$(i-1)$.

Observe that list *forw*(i) is bimonotonic, by the fact that Property (I$_1$) holds for *list*$(i-1)$. By analogy, this is true also for *back*(i). Therefore, we can merge them and call *list*$'(i)$ the resulting list. In order to prune the bimonotonic *list*$'(i)$ to obtain *list*(i), we do a scan with two pointers, dropping a pair if the previous one has the same second component. Thus Property (I$_1$) holds for *list*(i).

Since we assume that additions and comparisons on $O(\log C)$-bit numbers take unit time, that floating-point additions and comparisons take $O(\lceil \log(1/\varepsilon)/\log n\rceil)$ time, and the length of *list*$(i-1)$ is $O(n^2/\varepsilon)$, the construction of *list*(i) takes time $O(n^2\varepsilon^{-1}\lceil \log(1/\varepsilon)/\log n\rceil)$. □

Lemma 2. *Property* (I_2) *holds for* $list(i)$*, that is, for every* $i \in \{0, \dots, n\}$ *and every* $c \in \{0, \dots, C\}$*,* $(1 - \varepsilon/n)^i\, s(i, c) \le \underline{s}(i, c) \le s(i, c)$ *holds.*

Proof. The claim is clear for $i = 0$. For an arbitrary capacity $c \in \{0, \dots, C\}$, let $[c_1, \underline{r}_1]$ in $list(i)$ be such that $\underline{s}(i, c) = \underline{r}_1$. From the definition of $\underline{s}$, we get $\underline{s}(i, c) = \underline{s}(i, c_1)$; from the fact that the pairs in $list(i)$ are of the form (7), we have

$$\underline{s}(i, c) = \underline{s}(i, c_1) = \underline{s}(i - 1, c_1) \oplus \underline{s}(i - 1, c_1 - w_i). \tag{8}$$

Since the capacities in $list(i-1)$ are a subset of the capacities in $list'(i)$, and the fact that we have pruned the pairs in $list'(i)$ by keeping the smallest capacity for every approximate number of solutions corresponding to that capacity, it holds that

$$\underline{s}(i - 1, c_1) = \underline{s}(i - 1, c). \tag{9}$$

Moreover, observe that there is no capacity c_2 in $list(i - 1)$ such that $c_1 - w_i < c_2 < c - w_i$. Indeed, for assuming the contrary, $c_2 + w_i$ would be a capacity in $list'(i)$, by (7). Since we have chosen c_1 as the largest capacity in $list(i)$ smaller than c, and $c_1 < c_2 + w_i < c$ holds, this implies that $c_2 + w_i$ was pruned when passing from $list'(i)$ to $list(i)$; thus, the two pairs of $list'(i)$ having c_1 and $c_2 + w_i$ as first components have equal second components. By (8) and the bimonotonicity of $list(i - 1)$, this entails that also the two pairs of $list(i - 1)$ having $c_1 - w_i$ and c_2 as first components must have equal second components. This contradicts the fact that $list(i - 1)$ satisfies Property (I_1).

Therefore, it also holds that

$$\underline{s}(i - 1, c_1 - w_i) = \underline{s}(i - 1, c - w_i). \tag{10}$$

Plugging equations (9) and (10) into (8) we obtain

$$\underline{s}(i, c) = \underline{s}(i - 1, c) \oplus \underline{s}(i - 1, c - w_i). \tag{11}$$

From (6), the fact that Property (I_2) holds for $list(i - 1)$, and from (5), we get that $(1 - \varepsilon/n)^i s(i, c) \le \underline{s}(i, c) \le s(i, c)$, which shows that Property (I_2) holds also for $list(i)$. □

By standard techniques, for all natural numbers $n \ge 1$ and all $0 < \varepsilon \le 1$, the following hold:

$$1 - \varepsilon \le \left(1 - \frac{\varepsilon}{n}\right)^n, \text{ and } \left(1 - \frac{\varepsilon}{n}\right)^{-n} \le (1 - \varepsilon)^{-1}. \tag{12}$$

From Lemma 2, the fact that Property (I_2) holds, and (12), we obtain Thm. 1.

5 Random Generation of 0/1 Knapsack Solutions

For the random generation problem, we increase the length of the mantissa of the floating-point numbers up to $\lceil \log(n^2/\varepsilon) \rceil + 1$ bits.

Let $\underline{f}(i,c) := \underline{s}(i-1, c-w_i) \oslash \underline{s}(i,c)$ ('$\oslash$' denotes the floating-point division).[4] It is important to remark here that the number of solutions including object i is at most the number of solutions not including object i. Indeed, to every solution S containing object i we can associate a different solution not containing object i, namely $S \setminus \{i\}$. It follows that $\underline{f}(i,c) \leq \frac{1}{2}$ so that $\underline{f}(i,c)$ is conveniently bounded away from 1.

For clarity, assume for now that each $\underline{s}(i,c)$ and $\underline{f}(i,c)$ are available. We repeat the following procedure, for every i from n downto 1, and starting with $c = C$. With probability $\underline{f}(i,c)$ we include w_i in the solution, and move to entry $(i-1, c-w_i)$; with complementary probability we do not include w_i, and move to entry $(i-1, c)$. We next show how to implement this simple procedure so that it samples in $O(n \log(n/\varepsilon))$ expected time a Knapsack solution with probability different from the uniform one by a factor $(1-\varepsilon)^{\pm 1}$. The next lemma shows how to take each of the n subsequent choices.

Lemma 3. *For any $i \in \{1, \ldots, n\}$, $c \in \{0, \ldots, C\}$, in expected time $O(\log(n/\varepsilon))$ we can generate $B \in \{0,1\}$ with*

$$\left(1 - \frac{\varepsilon}{n^2}\right)^i \frac{s(i-1, c-w_i)}{s(i,c)} \leq \Pr(B = 0) \leq \left(1 - \frac{\varepsilon}{n^2}\right)^{-i} \frac{s(i-1, c-w_i)}{s(i,c)}.$$

Proof. By Property (I_2), we get $\left(1 - \varepsilon/n^2\right)^i s(i,c) \leq \underline{s}(i,c) \leq s(i,c)$. Together with (4), this implies

$$\left(1 - \frac{\varepsilon}{n^2}\right)^i \frac{s(i-1, c-w_i)}{s(i,c)} \leq \underline{f}(i,c) \leq \left(1 - \frac{\varepsilon}{n^2}\right)^{-i} \frac{s(i-1, c-w_i)}{s(i,c)}. \tag{13}$$

Thus, in order to generate B with the desired probability, it is enough to generate uniformly at random a number $x \in [0,1)$ and set $B = 0$ iff $x < \underline{f}(i,c)$.

This can be implemented in expected time $O(\log(n/\varepsilon))$ as follows. We start generating a random sequence of bits (starting with the most significant one of x) until seeing the first bit equal to '1' (the first bit of the mantissa of x). At this point, we know the exponent of x. Since $\underline{f}(i,c)$ has a mantissa of $\lceil \log(n^2/\varepsilon) \rceil + 1$ bits, in order to decide whether $x < \underline{f}(i,c)$, it is enough to generate other $\lceil \log(n^2/\varepsilon) \rceil$ bits for the mantissa of x. Call $\underline{x}$ the resulting floating-point number, and set $B = 0$ iff $\underline{x} < \underline{f}(i,c)$.

The exponent of x can be computed by starting with the exponent equal to 0, and for every bit of x equal to 0, subtracting 1 from it. Since the expected number of bits until seeing the first bit of x equal to '1' is 2, the expected time for generating $\underline{x}$ is $O(\log(n/\varepsilon))$. □

By Lemma 3, the probability X of generating a 0/1 Knapsack solution satisfies the following relation, which by (12) gives our $(1-\varepsilon)^{\pm 1}$ approximation:

$$\left(1 - \frac{\varepsilon}{n^2}\right)^{n^2} \frac{1}{s(n,C)} \leq X \leq \left(1 - \frac{\varepsilon}{n^2}\right)^{-n^2} \frac{1}{s(n,C)}.$$

Algorithm 2. ApproximateRandomGeneration($w_1, \ldots, w_n, C$)
Random generation of 0/1 Knapsack solutions.

```
1  compute $list(1), \ldots, list(n)$ with Algorithm 1, by attaching the two
   back-pointers to each element;
2  $c := C$;
3  $current$ := the last element of $list(n)$;
4  $solution := \emptyset$;
5  for $i := n$ downto 1 do
6      generate $B \in \{0, 1\}$ with Lemma 3;
7      if $B = 0$ then
8          $solution := solution \cup \{w_i\}$;
9          $c := c - w_i$;
10         update $current$ by following its back-pointer corresponding to choosing
           $w_i$ in the solution;
11     else
12         update $current$ by following its back-pointer corresponding to not
           choosing $w_i$ in the solution;
13 return $solution$.
```

We show now how to implement this random generation procedure efficiently, using the lists constructed in Sec. 4. See the resulting procedure in Algorithm 2. The idea is that for every element approximating an entry $s(i, c)$, we attach one pointer to the element of $list(i - 1)$ approximating $s(i - 1, c)$, and one pointer to the element of $list(i - 1)$ approximating $s(i - 1, c - w_i)$.

We do this by extending the construction of $forw(i)$ inside Lemma 1, as follows. Assume, like in Lemma 1, that pointer $left$ is on a pair $[c, \underline{r}]$ of $list(i-1)$. Assume also that pointer $right$ reached a pair $[c_1, \underline{r}_1]$, and that we need to append the pair $[c + w_i, \underline{r}_1 \oplus \underline{r}]$ (the approximation of $s(i, c + w_i)$) at the end of $forw(i)$. We now attach to it two back-pointers: one to element $left$ (the approximation of $s(i - 1, c + w_i)$) and one to element $right$ (the approximation of $s(i - 1, c)$). Similarly when computing list $back(i)$. The trace back in the random generation procedure starts in the last element of $list(n)$, and follows the back-pointers corresponding to whether the current element is included or not in the solution.

The time needed to construct this collection of extended lists is the same as before, the only difference being that the floating-point numbers have mantissas of $\lceil \log(n^2/\varepsilon) \rceil + 1$ bits, leading to a complexity of $O(n^4 \varepsilon^{-1} \lceil \log(1/\varepsilon)/\log n \rceil)$. Since we also need to store the back-pointers, the memory needed to store all the lists is $O(n^4 \varepsilon^{-1} \lceil \log(1/\varepsilon)/\log n \rceil W)$ bits, where W is the word size.

We can pre-compute each $\underline{f}(i, c)$ needed in Lemma 3 using one of the two back-pointers of every element of a list, and doing the floating-point division with the Newton-Raphson division method, which reduces a division to

[4] For simplicity, in all subsequent considerations, we ignore the technical issue that we need to use one extra bit for indicating that the exponent is less than zero.

a multiplication algorithm [9]. Thus, each division can be computed in time $O(\lceil\log(1/\varepsilon)/\log n\rceil M(\lceil\log(1/\varepsilon)/\log n\rceil)$ time, where $M(x) = \log x \log\log x$ [9] (assuming again operations on $O(\log n)$ bits to have unit cost). Generating one Knapsack solution takes expected time $O(n\log(n/\varepsilon))$, by Lemma 3. This completes the proof of Thm. 2.

6 Counting Knapsack Solutions on a DAG

Onwards, we briefly sketch the details on applying this method to the Knapsack problem on a DAG. We can assume that all vertices of the DAG D (with n vertices and m arcs) are reachable from s, and all vertices reach t. We transform D into an equivalent DAG D' in which every vertex has at most two in-coming arcs, and D' has $O(m)$ vertices and arcs, and the maximum path length (i.e., number of arcs in the path) is $O(n\log(\frac{m}{n}))$. This can be achieved as follows. For every node v_i of D, if its in-degree $d^-(v_i) > 2$, we construct a complete binary tree on top of the in-neighbors $v_{i_1}, \dots, v_{i_{d^-(v_i)}}$ of v_i, where v_i is its root; this tree has $O(d^-(v_i))$ vertices and edges, and depth $\log(d^-(v_i))$. The vertices and edges of this tree are added to D, the arcs from $v_{i_1}, \dots, v_{i_{d^-(v_i)}}$ to v_i are removed, and all edges of the tree are directed towards v_i. The weights of the new arcs out-going from $v_{i_1}, \dots, v_{i_{d^-(v_i)}}$ are set to be the weights of their former arcs towards v_i; all other new arcs have weight 0.

Say that D' has n' vertices and let $s = v_1, v_2, \dots, v_{n'} = t$ be a topological ordering of them. We now denote by $s(i,c)$ the number of paths that end in v_i and their total weight is at most $c \in \{0,\dots,C\}$. If for every node v_i, its its in-neighborhood is $\{v_{i_1}, v_{i_{d^-(v_i)}}\}$, and the weights of the arcs entering v_i are $w_{i_1}, w_{i_{d^-(v_i)}}$, respectively, relation (6) generalizes to:

$$s(i,c) = \begin{cases} s(i_1, c - w_{i_1}), & \text{if } d^-(v_i) = 1, \\ s(i_1, c - w_{i_1}) + s(i_2, c - w_{i_2}), & \text{if } d^-(v_i) = 2. \end{cases} \tag{14}$$

The solution is obtained as $s(n', C)$. As before, we use (14) to count, keeping at each step approximate floating-point numbers. These numbers still have $\lceil\log n\rceil$ bits for the exponent, but, since the maximum path length in D' is $O(n\log(\frac{m}{n}))$, the length of their mantissa will be $1 + \lceil\log(n\log(\frac{m}{n})/\varepsilon)\rceil$ bits. Additions and comparisons of these floating-point numbers still take the same time as before, namely $O(\lceil\log(1/\varepsilon)/\log n\rceil)$.

As before, for every $i \in \{1,\dots,n'\}$, we keep a list, $\mathit{list}(i)$, of pairs [capacity, approximate number of solutions], now of length at most $O(n^2\log(\frac{m}{n})\varepsilon^{-1})$. Analogously, $\mathit{list}(1)$ consists of the single pair $[0,1]$, and while computing $\mathit{list}(i)$ from lists $\mathit{list}(i_1)$, or $\mathit{list}(i_1)$ and $\mathit{list}(i_2)$ (doable now in time $O(n^2\log(\frac{m}{n})\varepsilon^{-1}\lceil\log(1/\varepsilon)/\log n\rceil))$, we maintain the following two invariants, where $\ell(i)$ denotes the length of the longest path from s to v_i:

(I_1) $\mathit{list}(i)$ is strictly increasing on both components;

(I_2) $\left(1 - \varepsilon/(n\log(\frac{m}{n}))\right)^{\ell(i)} s(i,c) \le \underline{s}(i,c) \le s(i,c)$, for every $c \in \{0,\dots,C\}$.

From these considerations, Thm. 3 immediately follows.

7 Conclusions

Our approach can be applied to combinatorial decompositions of various other problems, with the required math for bounding the run-time in terms of ε embodied in the technical floating-point arithmetic layer. It is a conceptual tool that can guide and inspire the design of new algorithms. In this new scenario, the length of the mantissa becomes a resource, and minimizing its consumption leads one to reduce the number of subsequent approximation phases in processing the data flow. This view indeed supported us in gaining an extra n factor in Thm. 3. Moreover, the algorithms inspired by this framework require very little ad-hoc analysis, thanks to the reusable layer of floating-point arithmetic.

Acknowledgements. This work was partially supported by the Academy of Finland under grant 250345 (CoECGR), and by the European Science Foundation, activity "Games for Design and Verification". We thank Djamal Belazzougui and Daniel Valenzuela for discussions on data structures for large numbers, and Stephan Wagner for remarks on decomposable structures.

References

1. Denise, A., Zimmermann, P.: Uniform Random Generation of Decomposable Structures Using Floating-Point Arithmetic. Theor. Comput. Sci. 218(2), 233–248 (1999)
2. Dyer, M.E.: Approximate counting by dynamic programming. In: Larmore, L.L., Goemans, M.X. (eds.) STOC, pp. 693–699. ACM (2003)
3. Dyer, M.E., Frieze, A.M., Kannan, R., Kapoor, A., Perkovic, L., Vazirani, U.V.: A Mildly Exponential Time Algorithm for Approximating the Number of Solutions to a Multidimensional Knapsack Problem. Combinatorics, Probability & Computing 2, 271–284 (1993)
4. Gopalan, P., Klivans, A., Meka, R.: Polynomial-Time Approximation Schemes for Knapsack and Related Counting Problems using Branching Programs. CoRR abs/1008.3187 (2010)
5. Gopalan, P., Klivans, A., Meka, R., Stefankovic, D., Vempala, S., Vigoda, E.: An FPTAS for #Knapsack and Related Counting Problems. In: Ostrovsky, R. (ed.) FOCS, pp. 817–826. IEEE (2011)
6. Meka, R., Zuckerman, D.: Pseudorandom generators for polynomial threshold functions. In: Schulman, L.J. (ed.) STOC, pp. 427–436. ACM (2010)
7. Mihalák, M., Šrámek, R., Widmayer, P.: Counting approximately-shortest paths in directed acyclic graphs. In: Kaklamanis, C., Pruhs, K. (eds.) WAOA 2013. LNCS, vol. 8447, pp. 156–167. Springer, Heidelberg (2014), http://arxiv.org/abs/1304.6707v2
8. Morris, B., Sinclair, A.: Random Walks on Truncated Cubes and Sampling 0-1 Knapsack Solutions. SIAM J. Comput. 34(1), 195–226 (2004)
9. Schönhage, A., Strassen, V.: Schnelle multiplikation großer zahlen. Computing 7(3-4), 281–292 (1971)
10. Štefankovič, D., Vempala, S., Vigoda, E.: A Deterministic Polynomial-Time Approximation Scheme for Counting Knapsack Solutions. SIAM J. Comput. 41(2), 356–366 (2012)

Improved Guarantees for Tree Cut Sparsifiers

Harald Räcke and Chintan Shah

Institut für Informatik, Technische Universität München

Abstract. Harrelson, Hildrum and Rao [11] construct for a given graph a single tree that acts as a flow sparsifier, i.e., it can approximate multicommodity flows in G up to an $\mathcal{O}(\log^2 n \log\log n)$ factor. Many applications that use these trees do not actually require a flow sparsifier but would already work with just having a cut sparsifier. We show how to construct a cut sparsifier that is a single tree and has quality $\mathcal{O}(\log^{1.5} n \log\log n)$.

In addition we show a close connection of this problem to the Mincut Linear Arrangement Problem which shows that improving the guarantee to $o(\log^{1.5} n)$ might be difficult.

1 Introduction

There exist many different results that aim to approximate the cut-structure of a graph $G = (V, E, \mathrm{cap}_G)$ by a different graph $H = (V_H, E_H, \mathrm{cap}_H)$, where H may either be smaller than G or may have a simpler structure. Formally, we say H is a *cut sparsifier* for G with quality q if $V \subseteq V_H$ and for every cut $U \subseteq V$ we have $\mathrm{cap}_G(U, V \setminus U) \leq \mathrm{mincut}_H(U, V \setminus U) \leq q \cdot \mathrm{cap}_G(U, V \setminus U)$. Benczur and Karger [4] e.g. show how to compute a $(1+\epsilon)$-quality cut sparsifier H that has the same vertex set as G and has a small edge-set.

A related notion is the concept of a *flow sparsifier* which is a graph H s.t. any feasible multicommodity flow in G can be routed in H with congestion at most 1, and any feasible flow in H (between vertices from V) can be supported in G with congestion at most q.

In the context of designing oblivious routing schemes, Räcke [17] developed tree flow sparsifiers, where the graph H is a single tree and the leaf nodes of the tree correspond to the vertex set V of G. He showed the existence of a tree flow sparsifier with quality $\mathcal{O}(\log^3 n)$, which was later improved by Harrelson, Hildrum and Rao [11] to $\mathcal{O}(\log^2 n \log\log n)$. These tree flow sparsifiers have subsequently been used to obtain approximate solutions for a variety of cut-related problems that seem very hard on general graphs but that are efficiently solvable on trees. Examples include: Minimum Bisection, Simultaneous Source Location, Online Multicut, k-multicut etc. (see e.g. [1,3,6,9,12,13,18]).

Instead of considering a *single tree* to approximate the cut-structure of a graph G, Räcke [18] considered a convex combination of trees and obtained a flow sparsifier with quality $\mathcal{O}(\log n)$. While quite a few problems like e.g. Minimum Bisection or Generalized Sparsest Cut can be approximated using a convex combination of trees, there are important problems that seem to require a single tree. In particular, if the objective function is not linear in the total capacity of cut edges, an approximation by a convex combination of trees does not work. (see e.g. [19,12,1] for such problems).

Interestingly, most results that use the trees of Harrelson et al. do not actually require a flow sparsifier but a cut sparsifier would work as well. Therefore in this paper we develop improved guarantees for cut sparsifiers that are single trees. One particular problem that benefits from the result of this paper is the Min-Max graph partitioning problem, where the goal is to partition a graph into k balanced components such that the number of edges incident to a single component is minimized. Stotz [19] analyzes this problem in the scenario where the balance constraint on the components is very tight (components should have size at most $(1+\epsilon)n/k$). He obtains a $(1+\epsilon, \mathcal{O}(q))$-bicriteria approximation, where q is the quality of a single-tree *cut sparsifier*.

A more precise statement of the result of Harrelson et al. is that they compute an $\mathcal{O}(\alpha \log n \log\log n)$-quality flow sparsifier that is a single tree, where α is the sparsest cut gap of the graph G. We show that we can efficiently compute a single tree that is an $\mathcal{O}(\beta \log n \log\log n)$-quality cut sparsifier where β is the approximation guarantee of a sparsest cut algorithm for product multicommodity flows It is known that for general graphs $\beta = \mathcal{O}(\sqrt{\log n})$ [2], while α may be $\Omega(\log n)$. If we only care about an existential result we can use an exact algorithm for sparsest cut and show that there exists an $\mathcal{O}(\log n \log\log n)$-quality cut sparsifier that is a single tree for arbitrary undirected graphs. There is a lower bound of $\Omega(\log n)$ for a tree *flow sparsifier* if G is a grid. Since the maxflow mincut gap of a grid graph is constant, this lower bound also applies to *cut sparsifiers*, which shows that our existential result is nearly tight.

Concerning the question whether our polynomial time construction can be improved to yield a cut sparsifier of better quality we show a close relationship of this problem to the mincut linear arrangement problem. Hence, it is likely that an improvement of the construction actually yields an improvement for mincut linear arrangement which seems very challenging.

1.1 Further Work

Vertex sparsifiers [16,14,5,8,15,10] are closely related to the sparsifiers that we consider in this paper. In this scenario we are given a large capacitated graph $G = (V, E, \mathrm{cap}_G)$ together with a set of *terminals* $T \subseteq V$. The goal is to generate a cut sparsifier $H = (V_H, E_H, \mathrm{cap}_H)$ just for the terminal set T. Hence, it is required that for all $U \subseteq T$: $\mathrm{mincut}_G(U, T \setminus U) \leq q\,\mathrm{mincut}_H(U, T \setminus U) \leq \mathrm{mincut}_G(U, T \setminus U)$.

One usually aims for the size of H to be independent of $n = |V|$ but only depend on $k = |T|$. In this context one can construct flow sparsifiers with quality $\mathcal{O}(\frac{\log k}{\log\log k})$ [5,15] and one can also get flow sparsifiers that are a convex combination of trees and have quality $\mathcal{O}(\log k)$ [10].

The idea of using ARV [2] to obtain better bounds when only cut-based guarantees are required appears frequently. For example, Chekuri et al. [7] develop *well-linked decompositions* to tackle edge disjoint path and related routing problems. They differentiate between *flow well linked* and *cut well linked* and obtain better bounds when just cut-well-linked decompositions are required.

1.2 Our Results

Theorem 1. *There is a polynomial time algorithm that for a weighted graph $G = (V, E, \mathrm{cap}_G)$ computes a tree cut sparsifier of quality $\mathcal{O}(\beta \log n \log\log n)$.*

Before sketching the techniques we use, we introduce some notation. For two sets $X, Y \subseteq V$ we use $\mathrm{cap}_G(X, Y) = \sum_{x\in X} \sum_{y\in Y} \mathrm{cap}_G((x, y))$, where we define $\mathrm{cap}_G((x, y)) = 0$ if $(x, y) \notin E$. Note that this counts an edge e twice if both end-points of e are in X and Y. For a set $S \subseteq V$ we will use $\mathcal{S} = \{S_0, \dots, S_\ell\}$ to denote a partition of S into subsets. $w_{\mathcal{S}}(x)$ is a function that for a vertex $x \in S$ counts the weight of edges incident to x that leave the sub-cluster of S in which x is contained (i.e., it counts edges that go into other sub-clusters or leave S). If the partition $\mathcal{S}$ is the trivial partition $\mathcal{S} = \{S\}$ we also write $w_S(x)$ instead of $w_{\{S\}}(x)$. This counts the capacity of border-edges of S incident to x. For two disjoint sets X and Y we use $E(X, Y)$ to denote the set of edges that connect nodes in X to nodes in Y.

A hierarchical decomposition of a graph G is a recursive partitioning of the vertex set into smaller and smaller components such that on the final level of the recursion the components just contain single vertices of G. Such a recursive decomposition corresponds in a natural way to a *decomposition tree* T in which leaf nodes correspond to nodes in G, the root corresponds to the whole vertex set, and internal vertices correspond to subsets generated during the partitioning.

We will identify a node in T with the component/cluster S of vertices in G that the node corresponds to. For such a cluster S we use $\mathcal{S}$ to denote its partition into child-clusters in T. Harrelson et al. [11] have constructed a hierarchical decomposition such that an all-to-all flow between edges connecting different sub-clusters can be routed with low congestion. We will not route a flow between these edges but we will embed an expander between them according to the following definition.

Definition 1. *Consider a cluster S together with its set $\mathcal{S}$ of child-clusters $\mathcal{S} = \{S_0, \dots, S_\ell\}$. We say a weighted graph $H_S = (S, E_H, \mathrm{cap}_H)$ is an* $\mathcal{S}$-expander *if* $\min_{U\subseteq S}\{\mathrm{cap}_H(U, S \setminus U)/\min\{w_{\mathcal{S}}(U), w_{\mathcal{S}}(S \setminus U)\}\} \geq 1$.

Theorem 1 follows from the following lemma.

Lemma 1. *There exists a polynomial time algorithm to find a hierarchical decomposition tree T, together with an $\mathcal{S}$-expander H_S for every cluster S such that all graphs H_S can be concurrently routed in G with congestion $\mathcal{O}(\beta \log n \log\log n)$.*

2 Constructing the Hierarchical Partition

We want to construct a hierarchical decomposition tree T, such that any cluster S in T has good expansion w.r.t. $\mathcal{S}$, its partition into child clusters. We demonstrate this by embedding a graph H_S which is an $\mathcal{S}$-expander into $G[S]$ with low congestion.

It is natural to construct T by providing a partitioning routine that on input of a cluster S outputs a set of sub-clusters $\mathcal{S} = \{S_0, \ldots, S_\ell\}$ of S, such that S has good expansion w.r.t. $\mathcal{S}$, and then to apply this routine recursively. However, for some cluster S, there may not exist any partition $\mathcal{S}$ for which S expands well.

Consider e.g. a cluster S that has a cut U for which $\mathrm{cap}(U, S \setminus U) \ll \min\{w_S(U), w_S(S \setminus U)\}$. Regardless of the chosen partition $\mathcal{S}$, routing any $\mathcal{S}$-expander will incur a large congestion. We call such a cut U a *bottleneck cut.* Harrelson, Hildrum and Rao [11] introduced the concept of a bad child event, whereby a cluster S that does not have a good partition, signals this to the caller by returning a bottleneck cut.

The partitioning of a cluster S is done in two phases – the *merge phase* and the *refinement phase.* In the merge phase, we try to find a partition $\mathcal{S}$ together with an $\mathcal{S}$-expander H_S such that H_S can be embedded in S with low congestion. If we succeed in finding such a partition, say $\mathcal{S}_{\mathrm{merge}}$, we proceed to the refinement phase. Otherwise, we find a bottleneck cut U that certifies that there is no good partition, and return this cut to the caller.

During the refinement phase for a cluster S, we call the partitioning routine for each child-cluster $S_i \in \mathcal{S}_{\mathrm{merge}}$. Whenever some S_i signals a bad child event and returns a bottleneck cut U, we find a related cut U' and *refine* the current partition by replacing S_i with $S_i \cap U'$ and $S_i \setminus U'$. Then we call the partitioning routine again on $S_i \cap U'$ and $S_i \setminus U'$. This process is repeated until all calls of the partitioning routine for sub-clusters succeeded. We use $\mathcal{S}_{\mathrm{final}}$ to denote the partition of S at the end of the refinement phase.

For the detailed description of our decomposition we require the following lemma, which follows from a β-approximation algorithm for Sparsest Cut.

Theorem 2. *Given a cluster S and a partition $\mathcal{S} = \{S_0, \ldots, S_\ell\}$ of S into sub-clusters, there is a polynomial time algorithm that finds an $\mathcal{S}$-expander H_S, and a parameter γ such that*

- *H_S embeds into $G[S]$ with congestion β/γ, and*
- *there is a cut $U \subseteq S$ in $G[S]$ with $|U| \leq |S|/2$, s.t. $\frac{\mathrm{cap}_G(U, S\setminus U)}{\min\{w_{\mathcal{S}}(U), w_{\mathcal{S}}(S\setminus U)\}} = \gamma$.*

U is the cut returned by a β-approximation for Sparsest Cut, where the demands are given by $\mathrm{dem}(u, v) = w_{\mathcal{S}}(u) w_{\mathcal{S}}(v)$, and γ is its sparsity. This means the sparsest cut in $G[S]$ has sparsity at least γ/β. Scaling capacities in $G[S]$ by β/γ gives us an $\mathcal{S}$-expander that can be embedded into $G[S]$ with congestion β/γ.

2.1 The Merge Phase

During the merge phase of a cluster S, we always maintain a working partition $\mathcal{S}$ of S. Starting with $\mathcal{S} = \{\{v\} \mid v \in S\}$, we perform the following steps:

1. Using the algorithm from Theorem 2, we find an $\mathcal{S}$-expander H_S, and an upper bound β/γ on the congestion for embedding H_S into $G[S]$. If $1/\gamma \leq$

$f(S)$, we set $\mathcal{S}_{\text{merge}} = \mathcal{S}$ and proceed to the refinement phase. Here, $f(S) = 8\log(|P|/|S|)\log\log n$, where P is the parent of S.

2. Otherwise, Theorem 2 returns a cut U with

$$\text{cap}_G(U, S \setminus U)/w_{\mathcal{S}}(U) = \gamma \leq 1/f(S) \ . \tag{1}$$

where $|U| \leq |S|/2$. If, in addition,

$$\text{cap}_G(U, S \setminus U)/w_S(U) \leq 2/f(S) \ , \tag{2}$$

we have a bottleneck cut. We signal this to the caller and return U. Otherwise, we remove vertices in U from other sets of $\mathcal{S}$[1], and *merge* them into one set U that we add to $\mathcal{S}$. Then we go to Step 1.

The following lemma shows that this process terminates.

Lemma 2. *An execution of the merge phase takes polynomial time.*

Proof. In a merge step $w_{\mathcal{S}}(S)$ changes as follows. We add edges between U and $S \setminus U$ which increases $w_{\mathcal{S}}(S)$ by $2\,\text{cap}_G(U, S \setminus U)$. We remove edges that are incident to nodes in U and connect different sub-clusters S_i in the current partition (but do not leave S). The weight of the latter set of edges is

$$\begin{aligned} w_{\mathcal{S}}(U) - w_S(U) &\geq f(S) \cdot \text{cap}(U, S \setminus U) - f(S) \cdot \text{cap}(U, S \setminus U)/2 \\ &= f(S) \cdot \text{cap}(U, S \setminus U)/2 \\ &= 4 \cdot \text{cap}(U, S \setminus U) \cdot \log\log n \cdot \log(|P|/|S|) > 2 \cdot \text{cap}(U, S \setminus U) \ . \end{aligned}$$

Here the first inequality is due to the fact that whenever we perform a merge step Equation 1 holds, but Equation 2 does not. The last inequality holds for $n \geq 4$ and uses the fact that $|P| \geq 2|S|$.

Hence, $w_{\mathcal{S}}(S)$ strictly decreases every time we perform a merge step. Initially, $w_{\mathcal{S}}(S)$ is a polynomial number, and since $w_{\mathcal{S}}(S)$ cannot be 0, the merge phase takes polynomial time. □

2.2 The Refinement Phase

In the refinement phase we refine the partition $\mathcal{S}_{\text{merge}}$ obtained at the end of the merge phase by partitioning a cluster S_i into $S_i \cap U'$ and $S_i \setminus U'$ whenever the call of the partitioning routine for S_i signals a bad child event and returns a bottleneck cluster U. We first motivate the choice of the cut U' that we make within S_i by relating it to our overall goal.

In the end we want to embed an $\mathcal{S}_{\text{final}}$-expander H_S into $G[S]$, where $\mathcal{S}_{\text{final}}$ is our final partition of S. After the merge phase we are able to embed an $\mathcal{S}_{\text{merge}}$-expander with congestion at most $\beta/\gamma \leq \beta f(S)$. One way to embed an $\mathcal{S}_{\text{final}}$-expander (with slightly larger congestion) is to route flow from the additional edges separated in $\mathcal{S}_{\text{final}}$ to the edges that were already separated in

[1] If sets in the partition become empty we delete them.

$\mathcal{S}_{\text{merge}}$. Formally, we say that we route from a set of edges X to a distribution $\mu(\cdot)$ over nodes, if a node v injects a flow equal to its weighted degree in X and receives a flow of $2\mu(v)\sum_{e\in X}\text{cap}_G(e)$. The following lemma shows that given a bottleneck cut $U \subseteq S_i$ (e.g. returned by a bad child event) we can find another bottleneck cut $U' \subseteq U$ such that we route from the edges between U' and $S_i \setminus U'$ to the distribution $w_{S_i}(\cdot)/w_{S_i}(U')$ among border nodes in U'.

Lemma 3. *Given a bottleneck cut U for S_i with $\text{cap}_G(U, S_i \setminus U)/w_{S_i}(U) \leq 2/f(S_i)$ we can find in polynomial time a cut $U' \subseteq U$ s.t.*

- $\text{cap}_G(U', S_i \setminus U')/w_{S_i}(U') \leq 2/f(S_i)$
- *we can route from the edges $E(U', S_i \setminus U')$ to the distribution $w_{S_i}(\cdot)/w_{S_i}(U')$ among nodes in U' while incurring congestion at most 2 on edges in $G[U'] \cup E(U', S_i \setminus U')$.*

Proof. We show how to find a cut U' s.t. the vertices *inside* U' can route the desired demand with congestion 1 inside $G[U']$. To route the complete flow we have to add the flow originating at nodes in $S_i \setminus U'$. This flow is first routed across the cut U' (resulting in load 1 on edges in $E(U', S_i \setminus U')$) and then to the distribution $w_{S_i}(\cdot)/w_{S_i}(U')$ by using the same flow as before a second time.

We construct an s–t flow network from $G[U]$ by adding a source s, a sink t, edges (s, u) with capacity $\text{cap}(u, S_i \setminus U)$ for $u \in U$, and edges (u, t) with capacity $\text{cap}(U, S_i \setminus U) \cdot w_{S_i}(u)/w_{S_i}(U)$ for $u \in U$. Observe that $\text{cap}(\{s\}, U) = \text{cap}(\{t\}, U) = \text{cap}(U, S_i \setminus U)$ and that sending a flow from s to t in this network corresponds to sending a flow from the $\text{cap}(U, S_i \setminus U)$ units of capacity generated by U to the boundary edges of S_i that are incident to U.

We compute the minimum cut in this network. If $\{t\}$ is a mincut, then $U' = U$ satisfies the first condition of the lemma by Equation 2, and the second condition since the mincut, and, hence, the maxflow has value $\text{cap}(U, S_i \setminus U)$.

Otherwise, let U' be the set of vertices in U, which are on the same side of the mincut as t. Observe that $U' \subsetneq U$, since the mincut has value strictly less than $\text{cap}(U, S_i \setminus U)$ and so $\{s\}$ is not a mincut. The capacity of the mincut is

$$\begin{aligned}
&\text{cap}(\{s\} \cup U \setminus U', U' \cup \{t\}) \\
&\qquad = \text{cap}(s, U') + \text{cap}(U \setminus U', U') + \text{cap}(U \setminus U', t) \\
&\qquad = \text{cap}(S_i \setminus U', U') + \text{cap}(U \setminus U', t) \\
&\qquad = \text{cap}(U', S_i \setminus U') + \text{cap}(U, S_i \setminus U) \cdot \frac{w_{S_i}(U \setminus U')}{w_{S_i}(U)} < \text{cap}(U, S_i \setminus U) \ .
\end{aligned}$$

This gives

$$\text{cap}(U', S_i \setminus U') < (w_{S_i}(U')/w_{S_i}(U)) \cdot \text{cap}(U, S_i \setminus U)$$

and, hence, $\text{cap}(U', S_i \setminus U')/w_{S_i}(U') < \text{cap}(U, S_i \setminus U)/w_{S_i}(U)$. This means U' is also a bottleneck cluster. Summing up we have shown that if $\{t\}$ is a mincut in the flow network the lemma follows for $U' = U$ and otherwise we find another bottleneck cut U' with strictly smaller cardinality. We can repeat the previous step for this new bottleneck cut. At some point this process stops and we obtain a U' that satisfies the lemma. $\square$

The description how to obtain the cut U' from cut U completes the description of the refinement phase.

2.3 Embedding an $\mathcal{S}$-Expander

The goal of this section is to prove the following technical version of Lemma 1.

Lemma 4. *There exists a polynomial time algorithm to find a hierarchical decomposition tree T, together with an $\mathcal{S}$-expander H_S for every cluster S such that the following holds: Every graph H_S can be routed in $G[S]$ in such a way that an edge $e \in S_i$ has load $\mathcal{O}(\beta \log\log n \cdot \log(|P|/|S_i|))$ while edges between clusters S_i have load $\mathcal{O}(\beta \log\log n \cdot \log(|P|))$. Here P is the parent cluster of S.*

Lemma 1 is an easy consequence of Lemma 4 and is proved in the full version.

For proving Lemma 4 we have to show how to embed an $\mathcal{S}_{\text{final}}$-expander into a cluster S. Recall that we can embed an $\mathcal{S}_{\text{merge}}$-expander with congestion at most $\beta \cdot f(S)$ into $G[S]$. We first show how we can route from the additional edges separating clusters in $\mathcal{S}_{\text{final}}$ to the edges separating clusters in $\mathcal{S}_{\text{merge}}$ with small congestion.

More concretely, consider a cluster $C \in \mathcal{S}_{\text{merge}}$. During the refinement phase this cluster may be partitioned several times. We can represent the refinement process by a binary tree T_C. The root of T_C corresponds to C, and the two children of a cluster $X \in T_C$ correspond to the two subsets into which X is refined. We show that we can route from the additional edges in the refinement of C to the border edges of C.

Lemma 5. *Let $C \in \mathcal{S}_{\text{merge}}$ and let $\mathcal{C}$ be the final refinement of C. We can route a flow inside $G[C]$ such that a node x injects $w_{\mathcal{C}}(x)$ units of flow and receives at most $2w_C(x)$ units. In addition*

- *The load on an edge contained in a sub-cluster $D \in \mathcal{C}$ is at most $4\log(|C|/|D|)$.*
- *The load on an inter-cluster edge in $\mathcal{C}$ is at most $4\log(|C|)$.*

Proof. For the following analysis we will view the flow as being routed between edges; for simplicity we also assume that all edges have capacity 1.

Each border edge generated in the refinement process for C appears at some point in T_C namely when a cluster X is split into X_L and X_R (wlog, assume that $|X_L| \leq |X_R|$). An edge from $G[C]$ between X_L and X_R will then give rise to a border-edge of X_L and to a border-edge of X_R. We say that these border edges are *generated* at X. They will also be border-edges for sub-clusters of X_L or X_R further down in the tree, i.e., the same edge may exist at different levels in the tree and we treat these edges as different in the following analysis.

The edges incident to leaf vertices in T_C represent all weight of $w_{\mathcal{C}}(C)$. We show how to send flow from these edges on the leaf level to the edges incident to the root cluster (edges contributing to $w_C(C)$) with low congestion.

Reversing the direction of T_C we can view it as an iterative merging process where starting from the leafs, clusters are successively merged until only the root-cluster C is left. Initially, every border-edge incident to a leaf cluster carries one unit of flow. Whenever two clusters X_L and X_R are merged into X we send flow that is currently sitting at the border edges generated by X to the border-edges of X as dictated by Lemma 3. The flow at other border-edges of X_L and X_R is just send to the corresponding copy of the border-edge on the next level. In this way we successively send the flow upwards in the tree until in the end it resides at border edges of C.

We define the *left depth* of a cluster X in T_C as the number of left-branches on a path from the root-cluster to X. Let $\Phi(j)$ denote an upper bound on the amount of flow routed to a border edge incident to a cluster of left depth at least j. We show by induction over the merge steps that

$$\Phi(j) \leq \prod_{i=j}^{\lceil \log |C| \rceil} \left(1 + \frac{1}{i \log \log n}\right) . \tag{3}$$

The left depth of a cluster can be at most $\lceil \log(|C|) \rceil + 1$. For the base case, $\Phi(j) = 1$. Equation 3 holds as on the right hand side, we get an empty product which is one.

Let X be a cluster with left depth j, and assume that Equation 3 holds for all descendant clusters of X. Let X_L and X_R denote the child-clusters of X with $|X_L| \leq |X_R|$. Note that cluster X_L has left depth at least $j+1$ and cluster X_R has left depth at least j. A border edge from X that is incident to a node from X_R only gets the flow from its downward copy, which means it only gets flow $\Phi(j)$ which is ok because X_R fulfills Equation 3. A border-edge e of X that is incident to a node in X_L gets at most flow

$$\Phi(j) \leq \Phi(j+1) + (2/f(X))(\Phi(j+1) + \Phi(j)) \leq \Phi(j+1) + (4/f(X))\Phi(j) .$$

Here $\Phi(j+1)$ comes from its downward copy, and the remaining terms come from the distribution of flow due to Lemma 3. Note that one portion of the flow sent to e comes from border-edges of X_L (left depth $\geq j+1$) and another from border edges of X_R (left depth $\geq j$). By Lemma 3, $\text{cap}(X_L, X_R) \leq 2w_X(X_L)/f(X)$, and since the flow is distributed evenly the equation follows. Rearranging gives

$$\Phi(j) \leq \Phi(j+1)/(1 - 4/f(X)) \leq \Phi(j+1) \cdot (1 + 8/f(X)) \tag{4}$$

as long as $4/f(X) \leq 1/2$, which holds for sufficiently large n.

Since a left child has size at most half of its parent in T_C, and since there are $j-1$ left edges on the path from C to X, $|X| \leq |C|/2^{j-1}$. Since $|C| \leq |S|/2, |X| \leq |S|/2^j$. So, $\log(|S|/|X|) \geq j$. Hence, $f(X) \geq 8j \log \log n$. From Equation 4 we get

$$\Phi(j) \leq \Phi(j+1) \cdot \left(1 + \frac{8}{8j \log \log n}\right) \leq \prod_{i=j}^{\lceil \log |C| \rceil} \left(1 + \frac{1}{i \log \log n}\right) ,$$

where the last inequality follows by plugging in the induction hypothesis for $\Phi(j+1)$. So, each border edge of C receives at most $\Phi(1)$ units of flow. Lemma 10 in [11] gives $\Phi(1) \leq 2$.

An edge is congested by a flow in Lemma 3 when it is on the small side of the cut, or when it is contained in the cut. In the first case, let D be the leaf node in T_C containing both endpoints of the edge. Since the left depth of D is at most $\log(|C|/|D|)$, the edge can be congested at most $\log(|C|/|D|)$ times. The second case happens at most once, and in that case, the edge is congested at most $\log|C|$ times, which is an upper bound on the height of T_C. $\square$

Proof (of Lemma 4). At the end of the merge phase for a cluster S, we have an $\mathcal{S}_{\text{merge}}$-expander H_S embedded into $G[S]$. Using Lemma 5, we embed a flow (represented by a graph H_{new}) that routes from the additional edges cut in the refinement phase to the border edges of the clusters being refined.

Let for $U \subseteq S$, $Z(U) = w_{\mathcal{S}_{\text{final}}}(U) - w_{\mathcal{S}_{\text{merge}}}(U)$ denote the weight added to U by the refinement. We wish to show that $H_S \cup H_{\text{new}}$ expands well, i.e. $(\text{cap}_{H_S}(U, S \setminus U) + \text{cap}_{H_{\text{new}}}(U, S \setminus U))/\min\{w_{\mathcal{S}_{\text{final}}}(U), w_{\mathcal{S}_{\text{final}}}(S \setminus U)\} \geq \frac{1}{8}$.

We differentiate two cases. First suppose that either $Z(U) \geq w_{\mathcal{S}_{\text{merge}}}(U)\}$ or $Z(S \setminus U) \geq w_{\mathcal{S}_{\text{merge}}}(S \setminus U)\}$ and assume w.l.o.g. that this holds for U. In the flow represented by H_{new} at most $2w_{\mathcal{S}_{\text{merge}}}(U)$ units can be absorbed within U. Hence, at least $Z(U) - 2w_{\mathcal{S}_{\text{merge}}}(U) \geq Z(U)/4 + w_{\mathcal{S}_{\text{merge}}}/4 = w_{\mathcal{S}_{\text{final}}}/4$ units have to leave U. For the other case, we have $Z(U) \leq 4w_{\mathcal{S}_{\text{merge}}}(U)$, where w.l.o.g. U fulfills $w_{\mathcal{S}_{\text{merge}}}(U) \leq w_{\mathcal{S}_{\text{merge}}}(S \setminus U)$. Then, since H_S is an $\mathcal{S}_{\text{merge}}$-expander we have $\text{cap}_{H_S}(U, S \setminus U) \geq w_{\mathcal{S}_{\text{merge}}}(U) \geq Z(U)/8 + w_{\mathcal{S}_{\text{merge}}}(U)/8 = w_{\mathcal{S}_{\text{final}}}/8$.

Now, we consider the congestion incurred in $G[S]$ to embed $H_S \cup H_{\text{new}}$. An edge e in cluster $S_i \in \mathcal{S}_{\text{merge}}$ in the merge phase, and cluster $S_j \in \mathcal{S}_{\text{final}}$ after the refinement phase (if S_i is not a bad cluster, $S_i = S_j$) has load at most $8\beta \log\log n \log(|P|/|S|)$ from the merge phase and load at most $4\log(|S_i|/|S_j|)$ from the refinement phase. Consequently, the total load on edge e is at most $8\beta \log\log n \log(|P|/|S_j|)$. A similar argument holds for inter-cluster edges.

As a final step, we observe that instead of embedding a 1/8 expander we embed a 1 expander by scaling H_S and H_{new} by a factor of 8. $\square$

3 Constructing a Cut Sparsifier

Theorem 3. *Given a graph G, there is a polynomial time algorithm to compute a cut sparsifier for G that is a single tree and has quality $\mathcal{O}(\beta \log n \log\log n)$.*

Given a decomposition tree T according to Lemma 1, we assign capacities to the edges of T as follows. The capacity of an edge in T connecting a cluster S to a parent cluster P is equal to $\text{cap}(S, V \setminus S)$, i.e., the capacity of all edges leaving cluster S in G. In the following we show that the tree defined in this way is a cut sparsifier for G. Observe that the leaf nodes of T correspond to nodes in V.

Fix a cut (B, W) in G. Here, $B \subseteq V$ is an arbitrary subset of vertices and $W := V \setminus B$ is its complement. We will refer to B as the set of *black vertices*, and to W as the set of *white vertices*. In order to compare cuts in T and in G we have to define a cut in T that separates the black leaf nodes from the white leaf nodes. For this we extend the sets B and W also to inner vertices of T. We color a vertex in T, that corresponds to cluster S, *white* if $w_{\mathcal{S}}(S \cap W) \geq w_{\mathcal{S}}(S \cap B)$, otherwise we color it *black*. This coloring induces a cut in T that separates the black nodes from the white nodes, and in particular the black leaf nodes from the white leaf nodes. We use $\mathrm{cap}_T(B,W)$ to denote the capacity of this cut. Similarly, $\mathrm{cap}_G(B,W)$ and $\mathrm{cap}_H(B,W)$ denotes the capacity of the cut (B,W), measured in G and H, respectively. Here, H is the graph formed by the union of all $\mathcal{S}$-expanders H_S that we obtained from the construction of T.

The fact that T is a cut sparsifier follows from the following three lemmas (some proofs omitted).

Lemma 6. *Any cut in T that separates the black leaf nodes from the white leaf nodes has capacity at least* $\mathrm{cap}_G(B,W)$.

Lemma 7. $\mathrm{cap}_T(B,W) \leq 2\,\mathrm{cap}_H(B,W)$.

Proof. Fix a cluster S and assume w.l.o.g. that $w_{\mathcal{S}}(S \cap W) \geq w_{\mathcal{S}}(S \cap B)$, i.e., the tree node corresponding to S is colored white. Let $S_0, \ldots, S_\ell$ denote the child-clusters of S, and let $S_0, \ldots, S_k$, $k < \ell$, be the black child-clusters of S. In the tree the edges from S to each of the S_i's, $0 \leq i \leq k$ are cut. The capacity of these edges is $\sum_{i=0}^{k} w_{S_i}(S_i) = \sum_{i=0}^{k} w_{S_i}(S_i \cap W) + \sum_{i=0}^{k} w_{S_i}(S_i \cap B)$. We can estimate the second term on the r.h.s. of this equation by $\sum_{i=0}^{k} w_{S_i}(S_i \cap B) \leq \sum_{i=0}^{\ell} w_{S_i}(S_i \cap B) = w_{\mathcal{S}}(S \cap B) \leq \mathrm{cap}_{H_S}(S \cap B, S \cap W)$. Here the equality comes from the definition $w_{S_i}(\cdot)$ and $w_{\mathcal{S}}(\cdot)$ and the final inequality follows from the expansion properties of H_S and the fact that $w_{\mathcal{S}}(S \cap B) \leq w_{\mathcal{S}}(S)/2$. For the first term we use $\sum_{i=0}^{k} w_{S_i}(S_i \cap W) \leq \sum_{i=0}^{k} w_{\mathcal{S}}(S_i \cap W) \leq \sum_{i=0}^{k} \mathrm{cap}_{H_{S_i}}(S_i \cap W, S_i \cap B)$ where the second inequality holds because of the expansion properties of H_{S_i} and the fact that $w_{\mathcal{S}_i}(S_i \cap W) \leq w_{\mathcal{S}_i}(S_i)/2$.

Hence, we can amortize the cut-edges leading from S to child-clusters of S against cut edges in H_S and cut edges in H_{S_i}'s. Doing this for all clusters S in the decomposition tree gives that we amortize against every edge in H at most twice which yields the lemma. □

Lemma 8. $\mathrm{cap}_H(B,W) \leq \mathcal{O}(\beta \log\log\log n) \cdot \mathrm{cap}_G(B,W)$.

4 Mincut Linear Arrangement

In this section we relate our results to the Mincut Linear Arrangement Problem and give some indication that a substantial improvement in the approximation guarantee of our construction would also lead to an improved approximation guarantee for this problem.

In the Mincut Linear Arrangement Problem we are given a weighted graph $G = (V, E, \mathrm{cap}_G)$ and the task is to map the vertices of G to the vertices of a linear array A. If the endpoints of an edge $(x,y) \in E_G$ are mapped to u and v in A all edges in A between u and v increase their load by $\mathrm{cap}_G((x,y))$. The goal is to minimize the maximum load of an edge in A. The Mincut Linear Arrangement problem has an $\mathcal{O}(\beta \cdot \log n)$-approximation algorithm were β is the approximation guarantee of a sparsest cut algorithm used as a sub-routine. The following lemma shows that our construction gives rise to an $\mathcal{O}(\log^{1.5} n \log\log n)$ approximation for Mincut Linear Arrangement.

Lemma 9. *Suppose we are given a graph G together with a decomposition tree T. Let for a cluster S in T, $\mathcal{S}$ denote the sub-cluster of S used in T and let $\mathcal{D}_S$ denote some other sub-clustering with $w_{\mathcal{D}_S}(S) \geq w_{\mathcal{S}}(S)/2$.*

Further, assume that a $\mathcal{D}_S$-expander H_S can be embedded into G with congestion C_S and that for every root-to-leaf path $S_0, \ldots, S_\ell$ in T $\sum_i C_{S_i} \leq X$. Then we can solve the Minimum Cut Linear Arrangement Problem on G with approximation guarantee $6X$.

Note that the above lemma does not require that H_S can be routed in $G[S]$, nor does it require that all $H_S's$ can be *concurrently* routed in G with small congestion. Only the H_{S_i} that lie on some root-to-leaf path can concurrently be routed in G. However, in contrast to Lemma 4 it requires that H_S can be routed with a uniform congestion (i.e., the congestion on an edge e does not depend on the cluster S_i in which e is contained on the next level).

The conditions for Lemma 9 follow from our decomposition with $X = \mathcal{O}(\beta \log n \log\log n)$ if we choose $\mathcal{D}_S = \mathcal{S}_{\mathrm{merge}}$ for any cluster S, where $\mathcal{S}_{\mathrm{merge}}$ is the partitioning of S after the merge phase.

Proof. We perform a DFS-traversal of T and arrange the leaf nodes (i.e., nodes in G) in the order of appearance into the linear array A. In the following we show that this linear arrangement gives a $6X$-approximation.

We assign the edges of G in a one-to-one fashion to clusters of T as follows. We say an edge e of G *is separated by* cluster S if both end-points of e are in cluster S but lie in different child-clusters of S. Note that the total weight of edges that are separated by a cluster S is at most $w_{\mathcal{S}}(S)$.

Depending on the mapping of G to the linear array A we also assign edges of A to clusters (albeit not in a one-to-one fashion). We say an edge $e \in A$ *belongs to* cluster S if the vertices from G that are mapped to the endpoints of e both lie in cluster S. Note that by this definition the clusters that an edge $e \in A$ belongs to, lie all on a root-to-leaf path in T.

An edge $e = (u, v) \in G$ induces load on an edge $e' = (x, y) \in A$ if, e.g., u is mapped to the left of x and v is mapped to the right of y (or vice versa). Since, the nodes in some cluster S are mapped to a continuous region of the linear array we can make the following observation.

Claim. An edge $e = (u, v) \in G$ that induces load on an edge $e' \in A$ must be separated by a cluster S to which e' belongs.

Corollary 1. *The load of an edge $e \in A$ is at most $\sum_{i=0}^{\ell} w_{\mathcal{S}_i}(S_i)$, where $S_0, \ldots, S_\ell$ are the clusters to which e belongs.*

The following lemma gives a lower bound (proof omitted).

Lemma 10. *Let* OPT *denote the cost of an optimum solution to the Minimum Cut Linear Arrangement Problem. Then for every cluster S we have* $\mathsf{OPT} \geq w_{\mathcal{S}}(S)/12C_S$.

We can now combine the upper bound in Corollary 1 and the lower bound from Lemma 10. According to Corollary 1 the load of an edge $e \in A$ is at most $\sum_{i=0}^{\ell} w_{\mathcal{S}_i}(S_i) = \sum_{i=0}^{\ell} C_{S_i} \cdot w_{\mathcal{S}_i}(S_i) \cdot C_{S_i}^{-1} \leq 12 \sum_{i=0}^{\ell} C_{S_i} \cdot \mathsf{OPT} \leq 12X \cdot \mathsf{OPT}$, where the first inequality is due to Lemma 10 and the last inequality follows since $S_0, \ldots, S_\ell$ are on a root-to-leaf path in T.

References

1. Andreev, K., Garrod, C., Golovin, D., Maggs, B., Meyerson, A.: Simultaneous source location. ACM Transactions on Algorithms 6(1) (2009)
2. Arora, S., Rao, S., Vazirani, U.: Expander flows, geometric embeddings, and graph partitionings. In: Proc. of the 36th STOC, pp. 222–231 (2004)
3. Bansal, N., Feige, U., Krauthgamer, R., Makarychev, K., Nagarajan, V., Naor, J.S., Schwartz, R.: Min-max graph partitioning and small set expansion. In: Proc. of the 52nd FOCS, pp. 17–26 (2011)
4. Benczur, A.A., Karger, D.R.: Approximate s-t min-cuts in $\tilde{O}(n^2)$ time. In: Proc. of the 28th STOC, pp. 47–55 (1996)
5. Charikar, M., Leighton, F.T., Li, S., Moitra, A.: Vertex sparsifiers and abstract rounding algorithms. In: Proc. of the 51st FOCS, pp. 265–274 (2010)
6. Chekuri, C., Khanna, S., Shepherd, F.B.: The all-or-nothing multicommodity flow problem. In: Proc. of the 36th STOC, pp. 156–165 (2004)
7. Chekuri, C., Khanna, S., Shepherd, F.B.: Multicommodity flow, well-linked terminals, and routing problems. In: Proc. of the 37th STOC, pp. 183–192 (2005)
8. Chuzhoy, J.: On vertex sparsifiers with steiner nodes. In: Proc. of the 44th STOC, pp. 673–688 (2012)
9. Engelberg, R., Könemann, J., Leonardi, S., Naor, J.S.: Cut problems in graphs with a budget constraint. Journal of Discrete Algorithms 5(2), 262–279 (2007)
10. Englert, M., Gupta, A., Krauthgamer, R., Räcke, H., Talgam-Cohen, I., Talwar, K.: Vertex sparsifiers: New results from old techniques. In: Serna, M., Shaltiel, R., Jansen, K., Rolim, J. (eds.) APPROX 2010. LNCS, vol. 6302, pp. 152–165. Springer, Heidelberg (2010)
11. Harrelson, C., Hildrum, K., Rao, S.: A polynomial-time tree decomposition to minimize congestion. In: Proc. of the 15th SPAA, pp. 34–43 (2003)
12. Khandekar, R., Kortsarz, G., Mirrokni, V.: Advantage of overlapping clusters for minimizing conductance. In: Fernández-Baca, D. (ed.) LATIN 2012. LNCS, vol. 7256, pp. 494–505. Springer, Heidelberg (2012)
13. Könemann, J., Parekh, O., Segev, D.: A unified approach to approximating partial covering problems. Algorithmica 59(4), 489–509 (2011)
14. Leighton, F.T., Moitra, A.: Extensions and limits to vertex sparsification. In: Proc. of the 42nd STOC, pp. 47–56 (2010)
15. Makarychev, K., Makarychev, Y.: Metric extension operators, vertex sparsifiers and Lipschitz extendability. In: Proc. of the 51st FOCS, pp. 255–264 (2010)
16. Moitra, A.: Approximation algorithms for multicommodity-type problems with guarantees independent of the graph size. In: Proc. of the 50th FOCS, pp. 3–12 (2009)
17. Räcke, H.: Minimizing congestion in general networks. In: Proc. of the 43rd FOCS, pp. 43–52 (2002)
18. Räcke, H.: Optimal hierarchical decompositions for congestion minimization in networks. In: Proc. of the 40th STOC, pp. 255–264 (2008)
19. Stotz, R.: Approximation algorithms for scheduling processes in distributed systems. Master's thesis, Institut für Informatik, Technische Universität München (2014)

Representative Families: A Unified Tradeoff-Based Approach

Hadas Shachnai and Meirav Zehavi

Department of Computer Science, Technion, Haifa 32000, Israel
{hadas,meizeh}@cs.technion.ac.il

Abstract. Let $M = (E, \mathcal{I})$ be a matroid, and let $\mathcal{S}$ be a family of subsets of size p of E. A subfamily $\widehat{\mathcal{S}} \subseteq \mathcal{S}$ represents $\mathcal{S}$ if for every pair of sets $X \in \mathcal{S}$ and $Y \subseteq E \setminus X$ such that $X \cup Y \in \mathcal{I}$, there is a set $\widehat{X} \in \widehat{\mathcal{S}}$ disjoint from Y such that $\widehat{X} \cup Y \in \mathcal{I}$. Fomin et al. (*Proc. ACM-SIAM Symposium on Discrete Algorithms, 2014*) introduced a powerful technique for fast computation of representative families for uniform matroids. In this paper, we show that this technique leads to a unified approach for substantially improving the running times of parameterized algorithms for some classic problems. This includes, among others, k-Partial Cover, k-Internal Out-Branching, and Long Directed Cycle. Our approach exploits an interesting tradeoff between running time and the size of the representative families.

1 Introduction

Matroid theory connects such disparate branches of combinatorial theory and algebra as graph theory, combinatorial optimization, linear algebra, and algorithm theory. Marx [20] was the first to apply matroids to design fixed-parameter tractable algorithms, using the notion of representative families as a main tool. Representative families for set systems were introduced by Monien [21].

Let E be a universe of n elements, and $\mathcal{I}$ a family of subsets of size at most k of E, for some $k \in \mathbb{N}$, i.e., $\mathcal{I} \subseteq \{S \subseteq E : |S| \leq k\}$. Then, $U_{n,k} = (E, \mathcal{I})$ is called a *uniform matroid*. Consider such a matroid and a family $\mathcal{S}$ of p-subsets of E, i.e., sets of size p. A subfamily $\widehat{\mathcal{S}} \subseteq \mathcal{S}$ *represents* $\mathcal{S}$ if for every pair of sets $X \in \mathcal{S}$ and $Y \subseteq E \setminus X$ such that $X \cup Y \in \mathcal{I}$ (i.e., $|Y| \leq (k-p)$), there is a set $\widehat{X} \in \widehat{\mathcal{S}}$ disjoint from Y. In other words, if a set Y can be extended to a set of size at most k by adding a subset from S, then it can also be extended to a set of the same size by adding a subset from $\widehat{\mathcal{S}}$.

The *Two Families Theorem* of Bollobás [2] implies that for any uniform matroid $U_{n,k} = (E, \mathcal{I})$ and a family $\mathcal{S}$ of p-subsets of E, for some $1 \leq p \leq k$, there is a subfamily $\widehat{\mathcal{S}} \subseteq \mathcal{S}$ of size $\binom{k}{p}$ that represents $\mathcal{S}$. For more general matroids, the generalization of Lovász for this theorem, given in [18], implies a similar result, and algorithms based on this generalization are given in [20] and [11].

A parameterized algorithm with parameter k has running time $O^*(f(k))$ for some function f, where O^* hides factors polynomial in the input size. A fast computation of representative families for uniform matroids plays a central role in obtaining better running times for such algorithms. Plenty parameterized algorithms are based dynamic programming, where after each stage, the algorithm computes a family $\mathcal{S}$ of sets that are partial solutions. At that point we can compute a subfamily $\widehat{\mathcal{S}} \subseteq \mathcal{S}$ that represents $\mathcal{S}$. Then, each reference to $\mathcal{S}$ can be replaced by a reference to $\widehat{\mathcal{S}}$. The representative family $\widehat{\mathcal{S}}$ contains "enough" sets from $\mathcal{S}$; therefore, such replacement preserves the correctness of the algorithm. Thus, if we can efficiently compute representative families that are small enough, we can substantially improve the running time of the algorithm.

For uniform matroids, Monien [21] computed representative families of size $\sum_{i=0}^{k-p} p^i$ in time $O(|\mathcal{S}|p(k-p)\sum_{i=0}^{k-p} p^i)$, and Marx [19] computed representative families of size $\binom{k}{p}$ in time $O(|\mathcal{S}|^2 p^{k-p})$. Recently, Fomin et al. [11] introduced a powerful technique which enables to compute representative families of size $\binom{k}{p}2^{o(k)}\log n$ in time $O(|\mathcal{S}|(k/(k-p))^{k-p}2^{o(k)}\log n)$, thus significantly improving the previous results.

In this paper, we show that the technique of [11] leads to a unified tradeoff-based approach for substantially improving the running time of parameterized algorithms for some classic problems. In particular, we demonstrate the applicability of our approach for the following problems (among others).

k-Partial Cover (k-PC): Given a universe U, a family $\mathcal{S}$ of subsets of U and a parameter $k \in \mathbb{N}$, find the smallest number of sets in $\mathcal{S}$ whose union contains at least k elements.

k-Internal Out-Branching (k-IOB): Given a *directed* graph $G = (V, E)$ and a parameter $k \in \mathbb{N}$, decide if G has an *out-branching* (i.e., a spanning tree having exactly one node of in-degree 0) with at least k nodes of out-degree ≥ 1.

1.1 Prior Work

The k-PC problem generalizes the well-known k-Dominating Set (k-DS) problem, defined as follows. Given a graph $G = (V, E)$ and a parameter $k \in \mathbb{N}$, find the smallest size of a set $U \subseteq V$ such that the number of nodes in the closed neighborhood of U is at least k. If k-PC can be solved in time $t(|U|, |\mathcal{S}|, k)$, then k-DS can be solved in time $t(|V|, |V|, k)$ (see, e.g., [3]). Note that the special cases of k-PC and k-DS in which $k = n$, are the classical NP-complete Set Cover and Dominating Set problems [12], respectively. Table 1 presents a summary of known parameterized algorithms for k-PC and k-DS. We note that the parameterized complexity of k-PC and k-DS has been studied also with respect to other parameters and for more restricted inputs (see, e.g., [3,10,28]).

The k-IOB problem is of interest in database systems [6]. A special case of k-IOB, called k-Internal Spanning Tree (k-IST), asks if a given *undirected* graph $G = (V, E)$ has a spanning tree with at least k internal nodes. An interesting application of k-IST, for connecting cities with water pipes, is given in [25]. The k-IST problem is NP-complete, since it generalizes the Hamiltonian Path problem [13]; thus, k-IOB is also NP-complete. Table 2 presents a summary of known parameterized algorithms for k-IOB and k-IST. More details on k-IOB, k-IST and variants of these problems can be found in the excellent surveys of [22,26].

Table 1. Known parameterized algorithms for k-PC and k-DS

Reference	Deterministic\Randomized	Variant	Running Time
Bonnet et al. [3]	*det*	k-PC	$O^*(4^k k^{2k})$
Bläser [1]	*rand*	k-PC	$O^*(5.437^k)$
Kneis et al. [16]	*det*	k-DS	$O^*((16+\epsilon)^k)$
	rand	k-DS	$O^*((4+\epsilon)^k)$
Chen et al. [4]	*det*	k-DS	$O^*(5.437^k)$
Kneis [15]	*det*	k-DS	$O^*((4+\epsilon)^k)$
Koutis et al. [17]	*rand*	k-DS	$O^*(2^k)$
This paper	**det**	**k-PC**	$\mathbf{O^*(2.619^k)}$

Table 2. Known parameterized algorithms for k-IOB and k-IST

Reference	Deterministic\Randomized	Variant	Running Time
Priesto et al. [24]	*det*	k-IST	$O^*(2^{O(k \log k)})$
Gutin al. [14]	*det*	k-IOB	$O^*(2^{O(k \log k)})$
Cohen et al. [5]	*det*	k-IOB	$O^*(55.8^k)$
	rand	k-IOB	$O^*(49.4^k)$
Fomin et al. [8]	*det*	k-IOB	$O^*(16^{k+o(k)})$
Fomin et al. [7]	*det*	k-IST	$O^*(8^k)$
Zehavi [29]	*rand*	k-IOB	$O^*(4^k)$
This paper	**det**	**k-IOB**	$\mathbf{O^*(6.855^k)}$

1.2 Our Results

Given a uniform matroid $U_{n,k} = (E, \mathcal{I})$ and a family $\mathcal{S}$ of p-subsets of E, we compute a subfamily $\widehat{\mathcal{S}} \subseteq \mathcal{S}$ of size $\frac{(ck)^k}{p^p(ck-p)^{k-p}} 2^{o(k)} \log n$ which represents $\mathcal{S}$, in time $O(|\mathcal{S}|((ck)/(ck-p))^{k-p} 2^{o(k)} \log n)$, for any fixed $c \geq 1$. For $c = 1$, we have the result of Fomin et al. [11]. As c grows larger, the size of $\widehat{\mathcal{S}}$ increases, with a corresponding decrease in computation time. This enables to obtain better running times for the algorithms for Long Directed Cycle, Weighted k-Path and Weighted k-Tree, as given in [11].

In particular, we use this approach to develop *deterministic* algorithms solving k-PC and k-IOB in times $O^*(2.619^k)$ and $O^*(6.855^k)$, respectively. We thus significantly improve the algorithm with the best known $O^*(5.437^k)$ running time for k-PC [1], and the deterministic algorithm with the best known $O^*(16^{k+o(k)})$ running time for k-IOB [8]. This also improves the running times of the best known deterministic algorithms for k-DS and k-IST (see Tables 1 and 2).

Independently of our work, Fomin et al. [9] have recently obtained a tradeoff similar to the one we show in Section 3.

Technical Contribution: Our unified approach exploits an interesting tradeoff between running time and the size of the representative families. This tradeoff is made precise by using, along with the scheme of [11], a parameter $c \geq 1$, which enables a more careful selection of elements to the sets.

Indeed, towards computing a representative family $\widehat{\mathcal{S}}$, we seek a family $\mathcal{F} \subseteq 2^E$ that satisfies the following condition. For every pair of sets $X \in \mathcal{S}$, and $Y \subseteq E \backslash X$ such that $X \cup Y \in \mathcal{I}$, there is a set $F \in \mathcal{F}$ such that $X \subseteq F$, and $Y \cap F = \emptyset$. Then, we compute $\widehat{\mathcal{S}}$ by iterating over all $S \in \mathcal{S}$ and $F \in \mathcal{F}$ such that $S \subseteq F$. The time complexity of this iterative process is the dominant factor in the overall running time. Thus, we seek a small family $\mathcal{F}$, such that for any $S \in \mathcal{S}$, the expected number of sets in $\mathcal{F}$ containing S is small. In constructing each set $F \in \mathcal{F}$, we insert each element $e \in E$ to F with probability $p/(ck)$. For $c = 1$, this is the approach proposed in [11]. When we take a larger value for c, we need to construct a larger family $\mathcal{F}$. Yet, since elements in E are inserted to sets in $\mathcal{F}$ with a smaller probability, we get that for any $S \in \mathcal{S}$, the expected number of sets in $\mathcal{F}$ containing S is smaller.

Organization: Section 2 gives some definitions and notation. Section 3 presents a tradeoff between running time and the size of the representative families. Using this computation, we derive in Sections 4 and 5 our main results, which are fast parameterized algorithms for k-PC and k-IOB. Finally, Section 6 shows the improvements in running times resulting from our tradeoff-based approach for three previous applications of representative families of [11].

Due to space constraints, some of the results are omitted. We give the full details in [27].

2 Preliminaries

We now define the weighted version of representative families.

Definition 1. *Given a matroid $U_{n,k} = (E, \mathcal{I})$, a family $\mathcal{S}$ of p-subsets of E, and a function $w : \mathcal{S} \to \mathbb{R}$, we say that a subfamily $\widehat{\mathcal{S}} \subseteq \mathcal{S}$* max (min) represents *$\mathcal{S}$ if for every pair of sets $X \in \mathcal{S}$, and $Y \subseteq E \setminus X$ such that $X \cup Y \in \mathcal{I}$, there is a set $\widehat{X} \in \widehat{\mathcal{S}}$ disjoint from Y such that $w(\widehat{X}) \geq w(X)$ ($w(\widehat{X}) \leq w(X)$).*

The special case where $w(S) = 0$, for all $S \in \mathcal{S}$, is the unweighted version of Definition 1.

Notation: Given a set U and a nonnegative integer t, let $\binom{U}{t} = \{U' \subseteq U : |U'| = t\}$. Also, recall that an out-tree T is a directed tree having exactly one node of in-degree 0, called *the root*. We denote by V_T, E_T, $i(T)$ and $\ell(T)$ the node set, edge set, number of internal nodes (i.e., nodes of out-degree ≥ 1) and number of leaves (i.e., nodes of out-degree 0), respectively.

3 A Tradeoff-Based Approach

In this section we sketch the beginning of the proof of the following theorem, which contains our main contribution. We note that, to fully prove this theorem, we essentially follow and redo the proof of Theorem 6 in [11], taking into account our tradeoff-related parameter c.

Theorem 1. *Given a parameter $c \geq 1$, a uniform matroid $U_{n,k} = (E, \mathcal{I})$, a family $\mathcal{S}$ of p-subsets of E, and a function $w : \mathcal{S} \to \mathbb{R}$, a family $\widehat{\mathcal{S}} \subseteq \mathcal{S}$ of size $\frac{(ck)^k}{p^p(ck-p)^{k-p}} 2^{o(k)} \log n$ that max (min) represents $\mathcal{S}$ can be found in time $O(|\mathcal{S}|(ck/(ck-p))^{k-p} 2^{o(k)} \log n + |\mathcal{S}| \log |\mathcal{S}|)$.*

Roughly speaking, the proof of Theorem 1 is structured as follows. We first argue that we can focus on finding a certain data structure to compute representative families. Then, we construct such a data structure that is not as efficient as required (first randomly, and then deterministically). Finally, we show how to improve the "efficiency" of this data structure (this is made precise below).

Proof. Clearly, we may assume that $|\mathcal{S}| \geq \frac{(ck)^k}{p^p(ck-p)^{k-p}} 2^{o(k)} \log n$. Recall that our computation of representative families requires finding initially a family $\mathcal{F} \subseteq 2^E$ that satisfies the following condition. For every pair of sets $X \in \mathcal{S}$, and $Y \subseteq E \setminus X$ such that $X \cup Y \in \mathcal{I}$, there is a set $F \in \mathcal{F}$ such that $X \subseteq F$, and $Y \cap F = \emptyset$. An (n, k, p)-*separator* is a data structure containing such a family $\mathcal{F}$, which, given a set $S \in \binom{E}{p}$, outputs the subfamily of sets in $\mathcal{F}$ that contain S, i.e., $\chi(S) = \{F \in \mathcal{F} : S \subseteq F\}$.

To derive a fast computation, we need an efficient (n, k, p)-separator, where efficiency is measured by the following parameters: $\zeta = \zeta(n, k, p)$, the number of sets in the family $\mathcal{F}$; $\tau_I = \tau_I(n, k, p)$, the time required to compute the family $\mathcal{F}$; $\Delta = \Delta(n, k, p)$, the maximum size of $\chi(S)$, for any $S \in \binom{E}{p}$; and $\tau_Q = \tau_Q(n, k, p)$, an upper bound for the time required to output $\chi(S)$, for any $S \in \binom{E}{p}$.

Given such a separator, a subfamily $\widehat{\mathcal{S}} \subseteq \mathcal{S}$ of size ζ that max (min) represents $\mathcal{S}$ can be constructed in time $O(\tau_I + |\mathcal{S}|\tau_Q + |\mathcal{S}| \log |\mathcal{S}|)$ as follows. First, compute $\mathcal{F}$, and $\chi(S)$ for all $S \in \mathcal{S}$. Then, order $\mathcal{S} = \{S_1, \ldots, S_{|\mathcal{S}|}\}$, such that $w(S_{i-1}) \geq w(S_i)$ ($w(S_{i-1}) \leq w(S_i)$), for all $2 \leq i \leq |\mathcal{S}|$. Finally, return all $S_i \in \mathcal{S}$ for which there is a set $F \in \mathcal{F}$ containing S_i but no S_j, for $1 \leq j < i$. Formally, return $\widehat{\mathcal{S}} = \{S_i \in \mathcal{S} : \chi(S_i) \setminus (\bigcup_{1 \leq j < i} \chi(S_j)) \neq \emptyset\}$. The correctness of this construction is proved in [11]. Thus, to prove the theorem it suffices to find an (n, k, p)-separator with parameters:

- $\zeta^* \leq \frac{(ck)^k}{p^p(ck-p)^{k-p}} 2^{o(k)} \log n$. [Separator size]
- $\tau_I^* \leq \frac{(ck)^k}{p^p(ck-p)^{k-p}} 2^{o(k)} n \log n$. [Initialization time]
- $\tau_Q^* \leq (ck/(ck-p))^{k-p} 2^{o(k)} \log n$. [Query time]

We start by giving an (n, k, p)-separator, that we call Separator 1, with the following parameters, which are worse than required:

– $\zeta^1 = O(\frac{(ck)^k}{p^p(ck-p)^{k-p}} k^{O(1)} \log n)$. [Separator size]
– $\tau_I^1 = O(\binom{2^n}{\zeta^1} n^{O(k)})$. [Initialization time]
– $\Delta^1 = O((ck/(ck-p))^{k-p} k^{O(1)} \log n)$. [Query size]
– $\tau_Q^1 = O(\frac{(ck)^k}{p^p(ck-p)^{k-p}} n^{O(1)})$. [Query time]

First, we give a randomized algorithm which constructs, with positive probability, an (n,k,p)-separator having the desired ζ^1 and Δ^1 parameters. We then show how to deterministically construct an (n,k,p)-separator having all the desired parameter values. Let $t = \frac{(ck)^k}{p^p(ck-p)^{k-p}}(k+1)\ln n$, and construct the family $\mathcal{F} = \{F_1,\ldots,F_t\}$ as follows. For each $i \in \{1,\ldots,t\}$ and element $e \in E$, insert e to F_i with probability $p/(ck)$. The construction of different sets in $\mathcal{F}$, as well as the insertion of different elements into each set in $\mathcal{F}$, are independent. Clearly, $\zeta^1 = t$ is within the required bound.

For fixed sets $X \in \binom{E}{p}$, $Y \in \binom{E\setminus X}{k-p}$ and $F \in \mathcal{F}$, the probability that $X \subseteq F$ and $Y \cap F = \emptyset$ is $(\frac{p}{ck})^p(1-\frac{p}{ck})^{k-p} = \frac{p^p(ck-p)^{k-p}}{(ck)^k} = (k+1)\ln n/t$. Thus, the probability that *no* set $F \in \mathcal{F}$ satisfies $X \subseteq F$ and $Y \cap F = \emptyset$ is $(1-(k+1)\ln n/t)^t \le e^{-(k+1)\ln n} = n^{-k-1}$. There are at most n^k choices for $X \in \binom{E}{p}$ and $Y \in \binom{E\setminus X}{k-p}$; thus, applying the union bound, the probability that there exist $X \in \binom{E}{p}$ and $Y \in \binom{E\setminus X}{k-p}$, such that no set $F \in \mathcal{F}$ satisfies $X \subseteq F$ and $Y \cap F = \emptyset$, is at most $n^{-k-1} \cdot n^k = 1/n$.

For any sets $S \in \binom{E}{p}$ and $F \in \mathcal{F}$, the probability that $S \subseteq F$ is $(p/(ck))^p$. Therefore, $|\chi(S)|$, the number of sets in $\mathcal{F}$ containing S, is a sum of t i.i.d. Bernoulli random variables with parameter $(p/(ck))^p$. Then, the expected value of $|\chi(S)|$ is $E[|\chi(S)|] = t(\frac{p}{ck})^p = (\frac{ck}{ck-p})^{k-p}(k+1)\ln n$. Applying standard Chernoff bounds, we have that the probability that $|\chi(S)| \ge 6E[|\chi(S)|]$ is upper bounded by $2^{-6E[|\chi(S)|]} \le n^{-k-1}$. There are $\binom{n}{p}$ choices for $S \in \binom{E}{p}$. Thus, by the union bound, the probability that $\Delta^1 > 6 \cdot [((ck)/(ck-p))^{k-p}(k+1)\ln n]$ is upper bounded by $1/n$.

So far, we have given a randomized algorithm that constructs an (n,k,p)-separator having the desired ζ^1 and Δ^1 parameters with probability at least $1-2/n > 0$. To deterministically construct $\mathcal{F}$ in time bounded by τ_I^1, we iterate over all families of t subsets of E (there are $\binom{2^n}{\zeta^1}$ such families), where for each family $\mathcal{F}$, we test in time $n^{O(k)}$ whether Δ^1 is within the required bound, and whether for any pair of sets $X \in \binom{E}{p}$ and $Y \in \binom{E\setminus X}{k-p}$, there is a set $F \in \mathcal{F}$ such that $X \subseteq F$ and $Y \cap F = \emptyset$. Then, given a set $S \in \binom{E}{p}$, we can deterministically compute $\chi(S)$ within the stated bound for τ_Q^1, by iterating over $\mathcal{F}$ and inserting each set that contains S.

We next repeatedly apply Lemmas 4.4 and 4.5 of [11] to Separator 1, constructing intermediate separators different than those in [11] (as we start with a different Separator 1). This process eventually leads to an (n, k, p)-separator having the desired parameters ζ^*, τ^*_I and τ^*_Q. □

Given a parameter c and assuming that w is irrelevant, let $\mathsf{RepAlg}(E, k, \mathcal{S})$ be the algorithm developed in Theorem 1.

4 An Algorithm for *k*-Partial Cover

We now apply our scheme, RepAlg, to obtain a faster parameterized algorithm for k-PC. Let $m = |\mathcal{S}|$ be the number of sets in $\mathcal{S}$. The main idea of the algorithm is to iterate over the sets in $\mathcal{S}$ in some arbitrary order $S_1, S_2, \ldots, S_m$, such that when we reach a set S_i, we have already computed representative families for families of "partial solutions" that include only elements from the sets $S_1, \ldots, S_{i-1}$. Then, we try to extend the partial solutions by adding *uncovered* elements from S_i. The key observation, that leads to our improved running time, is that we cannot simply add "many" elements from S_i at once, but rather add these elements one-by-one; thus, we can compute new representative families after adding each element, which are then used when adding the next element.

The Algorithm: We now describe PCAlg, our algorithm for k-PC (see the pseudocode below). The first step solves the simple case where k elements can be covered with one set. Then, algorithm PCAlg generates a matrix M, where each entry $\mathrm{M}[i, j, \ell]$ holds a family that represents $Sol_{i,j,\ell}$. The sets in $Sol_{i,j,\ell}$ are those of exactly j elements, which can be covered by ℓ sets among $\{S_1, \ldots, S_i\}$, i.e., $Sol_{i,j,\ell} = \{S \subseteq (\bigcup \mathcal{S}') : \mathcal{S}' \subseteq \{S_1, \ldots, S_i\}, |S| = j, |\mathcal{S}'| = \ell\}$.

PCAlg iterates over all triples (i, j, ℓ), where $i \in \{1, \ldots, m\}, j \in \{1, \ldots, k\}$ and $\ell \in \{1, \ldots, \min\{i, k\}\}$. In each iteration, corresponding to a triple (i, j, ℓ), PCAlg computes $\mathrm{M}[i, j, \ell]$ by using $\mathrm{M}[i-1, j', \ell-1]$, for all $1 \leq j' \leq j$, and $\mathrm{M}[i-1, j, \ell]$. In other words, PCAlg computes a family that represents $Sol_{i,j,\ell}$ by using families that represent $Sol_{i-1,j',\ell-1}$, for all $1 \leq j' \leq j$, and $Sol_{i-1,j,\ell}$. In particular, algorithm PCAlg adds elements in S_i one-by-one to sets in $\mathrm{M}[i-1, j', \ell-1]$, for all $1 \leq j' \leq j$. After adding an element, PCAlg computes (in Step 7) new representative families, to be used when adding the next element. Let $S_i = \{s_1, \ldots, s_r\}$. Then, PCAlg computes a family $\mathcal{A}_{r',j'}$, for all $1 \leq r' \leq r$ and $0 \leq j' \leq j$, that represents the family of sets of exactly j' elements that can be covered by $\{s_1, \ldots, s_{r'}\}$ and $\ell - 1$ sets among $\{S_1, \ldots, S_{i-1}\}$. The family $\mathcal{A}_{r',j'}$ is computed by calling RepAlg with the *family parameter* containing the union of $\mathcal{A}_{r'-1,j'}$ and the family of sets obtained by adding $s_{r'}$ to sets in $\mathcal{A}_{r'-1,j'-1}$.

Suppose the solution is ℓ^*. Then, using representative families guarantees that each entry $\mathrm{M}[i, j, \ell]$ holds "enough" sets from $Sol_{i,j,\ell}$, such that when the algorithm terminates, $\mathrm{M}[m, k, \ell^*] \neq \emptyset$. Moreover, using representative families guarantees that each entry $\mathrm{M}[i, j, \ell]$ does not hold "too many" sets from $Sol_{i,j,\ell}$, thereby yielding an improved running time.

Correctness and Running Time: We first state a lemma referring to Steps 5–8 in PCAlg. In this lemma, we use the following notation. For all

Algorithm 1. $\mathsf{PCAlg}(U, k, \mathcal{S} = \{S_1, \ldots, S_m\})$

```
1: if there is $S \in \mathcal{S}$ s.t. $|S| \geq k$ then return 1. end if
2: let M be a matrix that has an entry $[i, j, \ell]$ for all $0 \leq i \leq m, 1 \leq j \leq k$ and
   $0 \leq \ell \leq k$, initialized to $\emptyset$.
3: for $i = 1, \ldots, m,\ j = 1, \ldots, k,\ \ell = 1, \ldots, \min\{i, k\}$ do
4:     let $S_i = \{s_1, \ldots, s_r\}$.
5:     $\mathcal{A}_{0,0} \Leftarrow \{\emptyset\}$, and for $j' = 1, \ldots, j$ do $\mathcal{A}_{0,j'} \Leftarrow \mathrm{M}[i-1, j', \ell-1]$. end for
6:     for $r' = 1, \ldots, r,\ j' = 0, \ldots j$ do
7:         $\mathcal{A}_{r',j'} \Leftarrow \mathsf{RepAlg}(U, k, [\mathcal{A}_{r'-1,j'} \cup \{S \cup \{s_{r'}\} : j' \geq 1,\ S \in \mathcal{A}_{r'-1,j'-1},\ s_{r'} \notin S\}])$.
8:     end for
9:     $\mathrm{M}[i, j, \ell] \Leftarrow \mathsf{RepAlg}(U, k, \mathrm{M}[i-1, j, \ell] \cup \mathcal{A}_{r,j})$.
10: end for
11: return the smallest $\ell$ such that $\mathrm{M}[m, k, \ell] \neq \emptyset$.
```

$0 \leq i \leq m, 1 \leq j \leq k$ and $0 \leq \ell \leq k$, let $\mathcal{A}^*_{i,j,\ell}$ denote the family of sets containing j elements, constructed by adding elements from S_i to sets in $(\bigcup_{1 \leq j' \leq j} \mathrm{M}[i-1, j', \ell-1]) \cup \{\emptyset\}$, i.e., $\mathcal{A}^*_{i,j,\ell} = \{S \cup S'_i : S \in (\bigcup_{1 \leq j' \leq j} \mathrm{M}[i-1, j', \ell-1]) \cup \{\emptyset\}, S'_i \subseteq S_i, |S \cup S'_i| = j\}$.

Lemma 2. *Consider an iteration of Step 3 in* PCAlg*, corresponding to some values i, j and ℓ. Then, the family $\mathcal{A}_{r,j}$ represents the family $\mathcal{A}^*_{i,j,\ell}$.*

We use Lemma 2 in proving the next lemma, showing the correctness of PCAlg.

Lemma 3. *For all $0 \leq i \leq m$, $1 \leq j \leq k$ and $0 \leq \ell \leq k$, $M[i, j, \ell]$ represents $Sol_{i,j,\ell}$.*

We summarize in the next result.

Theorem 4. PCAlg *solves k-PC in time $O(2.619^k |\mathcal{S}| \log^2 |U|)$.*

Proof. Lemma 3 and Step 11 imply that PCAlg solves k-PC. Also, Lemmas 2 and 3, and the way RepAlg proceeds, imply that PCAlg runs in time

$$O(2^{o(k)} |\mathcal{S}| \log^2 |U| \cdot \max_{0 \leq t \leq k} \left\{ \frac{(ck)^k}{t^t (ck-t)^{k-t}} (\frac{ck}{ck-t})^{k-t} \right\})$$

Choosing $c = 1.447$, the maximum is obtained at $t = \alpha k$, where $\alpha \cong 0.55277$. Thus, PCAlg runs in time $O(2.61804^k |\mathcal{S}| \log^2 |U|)$.[1] □

5 An Algorithm for k-Internal Out-Branching

We show below how to use our scheme, RepAlg, to obtain a faster parameterized algorithm for k-IOB. We first define an auxiliary problem called (k,t)-Tree, which requires finding a tree on a "small" number of nodes, rather than a spanning tree. Given a directed graph $G = (V, E)$, a node $r \in V$, and nonnegative integers k and t, the (k,t)-Tree problem asks if G contains an out-tree T rooted at r, such that $i(T) = k$ and $\ell(T) = t$. The following lemma implies that we can focus on solving (k,t)-Tree.

[1] Choosing $c = 1$, PCAlg runs in time $O^*(2.851^k)$.

Lemma 5 ([29]). *If* (k,t)-TREE *can be solved in time* $\tau(G,k,t)$, *then* k-IOB *can be solved in time* $O(|V|(|E| + \sum_{1\leq t\leq k} \tau(G,k,t)))$.

We next solve (k,t)-TREE. Our solution technique is based on iterating over all pairs of nodes $v,u \in V$, and all values $0 \leq i \leq k-1$ and $0 \leq \ell \leq t$. When we reach such v,u,i and ℓ, we have already computed, for all $v',u' \in V$, $0 \leq i' \leq i$, and $0 \leq \ell' \leq \ell$ satisfying $i'+\ell' < i+\ell$, representative families of "partial solutions". Such a partial solution is a set of nodes of an out-tree of G that is rooted at v', includes u' as a leaf (unless $v'=u'$) and consists of i' internal nodes (excluding v') and ℓ' leaves (excluding u'). We then try to "connect" out-trees represented by partial solutions in a manner that results in a *legal* out-tree—i.e., an out-tree of G that is rooted at v, includes u as a leaf (unless $v=u$) and consists of i internal nodes (excluding v) and ℓ leaves (excluding u). In constructing a set of such legal out-trees, we add families of "small" partial solutions one-by-one, so we can compute new representative families after adding each family, and then use them when adding the next one—this is a crucial point in obtaining our improved running time. The construction itself is quite technical. On a high level, it consists of iterating over some trees that indicate which families of partial solutions should be currently used, and in which order they should be added. We briefly note that this iterative process is based on a tool called *guiding trees*, introduced in [23] (based on [11]).

Some Definitions: Let $d \geq 2$ be a constant. Given nodes $v,u \in V$, $0 \leq i \leq k-1$ and $0 \leq \ell \leq t$, let $\mathcal{T}_{v,u,i,\ell}$ be the set of out-trees of G rooted at v, having exactly i internal nodes and ℓ leaves, excluding v and u, where $v=u$ or u is a leaf. Also, let $Sol_{v,u,i,\ell} = \{V_T \setminus \{v,u\} : T \in \mathcal{T}_{v,u,i,\ell}\}$. Given nodes $v,u \in V$, let $\mathcal{C}_{v,u}$ be the set of trees C rooted at v, where $v=u$ or u is a leaf, $V_C \subseteq V$, and $3 \leq |V_C| \leq 4d$. Given a node v of a rooted tree T, let $f_T(v)$ be the father of v in T.

The Algorithm: We now describe TreeAlg, our algorithm for (k,t)-TREE (see the pseudocode below). TreeAlg first generates a matrix M, where each entry $\mathrm{M}[v,u,i,\ell]$ holds a family that represents $Sol_{v,u,i,\ell}$. TreeAlg iterates over all $i \in \{0,\ldots,k-1\}$, $\ell \in \{0,\ldots,t\}$ such that $1 \leq i+\ell$, and $v,u \in V$. Next, consider some iteration, corresponding to such i,ℓ,v and u.

The goal in each iteration is to compute $\mathrm{M}[v,u,i,\ell]$, by using entries that are already computed. TreeAlg generates a matrix N, where each entry $\mathrm{N}[C]$ holds a family that represents the subfamily of $Sol_{v,u,i,\ell}$ including the node set (excluding v and u) of each out-tree $T \in \mathcal{T}_{v,u,i,\ell}$ *complying* with the rooted tree C as follows (see Fig. 1): (1) for any two nodes $v',u' \in V_C$, v' is an ancestor of u' in C iff v' is an ancestor of u' in T, (2) the leaves in C are leaves in T, and (3) in the forest obtained by removing V_C from T, each tree has at most two neighbors in T from V_C and, unless this neighborhood includes only v, the tree contains at most $(k+t)/d$ nodes. Roughly speaking, each entry $\mathrm{N}[C]$ is easier to compute than the entry $\mathrm{M}[v,u,i,\ell]$, since C "guides" us through the computation as follows. The rooted tree C implies which entries in M are relevant to $\mathrm{N}[C]$, in which order they should be used, and, in particular, it ensures that these are only entries of the form $\mathrm{M}[v',u',i',\ell']$, where $i'+\ell' \leq (k+t)/d$. This bound on $i'+\ell'$ ensures that the families for which we compute representative families are "small", thereby reducing the running time of calls to RepAlg. Next, consider an iteration corresponding to some $C \in \mathcal{C}_{v,u}$.

The current goal is to compute N[C], using the guidance of C. To this end, TreeAlg generates a matrix L, where each entry L$[j, i', \ell']$ holds a family that represents the family of node sets, excluding nodes in V_C, of trees in $\mathcal{P}_{v,u,C,j,i',\ell'}$, which is defined as follows. The set $\mathcal{P}_{v,u,C,j,i',\ell'}$ includes each subtree P' of G complying with the subtree P of C induced by $\{w_1, \ldots, w_j\}$, demanding only from leaves in P that are leaves in C to be leaves in P', such that: (1) $V_{P'} \cap (V_C \setminus V_P) = \emptyset$, and (2) the number of internal nodes (leaves) in P', excluding those in V_P, is i' (ℓ'). Informally, we consider such a subtree P' as a stage towards computing an out-tree $T \in \mathcal{T}_{v,u,i,\ell}$ that complies with C. Indeed, $\mathcal{P}_{v,u,C,|V_C|,i^*,\ell^*}$ is the set of out-trees in $\mathcal{T}_{v,u,i,\ell}$ that comply with C.[2] Roughly speaking, the matrix L is computed by using dynamic programming and RepAlg (in Steps 8–12) as follows. Each entry in L is computed by adding node sets of certain "small" trees to node sets of trees computed at a previous stage, and then calling RepAlg to compute a representative family for the result.

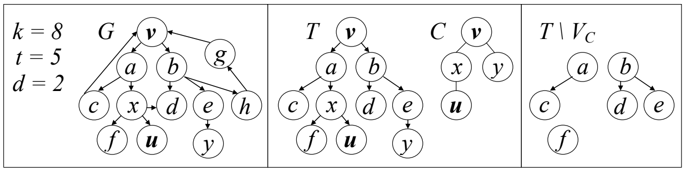


Fig. 1. An out-tree $T \in \mathcal{T}_{v,u,4,4}$, complying with the rooted tree $C \in \mathcal{C}_{v,u}$

Correctness and Running Time: The following lemma implies the correctness of TreeAlg.

Lemma 6. *For all* $v, u \in V$, $0 \leq i < k$ *and* $0 \leq \ell \leq t$, $M[v,u,i,\ell]$ *represents* $Sol_{v,u,i,\ell}$.

For $c = 1.447$ and a large enough constant d, we obtain the following result.

Lemma 7. TreeAlg *solves* (k, t)-Tree *in time* $O(2.61804^{k+t}|V|^{O(1)})$.

Finally, Lemmas 5 and 7 imply the following theorem.[3]

Theorem 8. k-IOB *can be solved in time* $O(6.85414^k|V|^{O(1)})$.

6 Improving Known Applications

Fomin et al. [11] proved that Long Directed Cycle, Weighted k-Path and Weighted k-Tree can be solved in times $O(8^k|E|\log^2|V|)$, $O(2.851^k|V|\log^2|V|)$

[2] Note that i^* (ℓ^*), defined in Step 6, is the number of internal nodes (leaves) in an out-tree $T \in \mathcal{T}_{v,u,i,\ell}$, excluding those in V_C.

[3] Choosing $c = 1$, we solve k-IOB in time $O^*(8.125^k)$.

Algorithm 2. $\mathsf{TreeAlg}(G=(V,E),r,k,t)$

```
1: let M be a matrix that has an entry $[v,u,i,\ell]$ for all $v,u \in V$, $0 \leq i \leq k-1$ and $0 \leq \ell \leq t$, which is initialized to $\emptyset$.
2: $\mathrm{M}[v,u,0,0] \Leftarrow \{\emptyset\}$ for all $v,u \in V$ s.t. $(v,u) \in E$ or $v=u$.
3: for $i=0,\dots,k-1$, $\ell=0,\dots,t$ s.t. $1 \leq i+\ell$, all $v,u \in V$ do
4:    let N be a matrix that has an entry $[C]$ for all $C \in \mathcal{C}_{v,u}$.
5:    for all $C \in \mathcal{C}_{v,u}$ do
6:       let $w_1,\dots,w_{|V_C|}$ be a preorder on $V_C$, where $w_1=v$, and let $i^*=i+1-i(C)$ and $\ell^*=\ell+|\{u\}\setminus\{v\}|-\ell(C)$.
7:       let L be a matrix that has an entry $[j,i',\ell']$ for all $1 \leq j \leq |V_C|$, $0 \leq i' \leq i^*$ and $0 \leq \ell' \leq \ell^*$, which is initialized to $\emptyset$.
8:       $\mathrm{L}[1,i',\ell'] \Leftarrow \{U \in \mathrm{M}[v,v,i',\ell'] : U \cap V_C = \emptyset\}$ for all $0 \leq i' \leq i^*$ and $0 \leq \ell' \leq \ell^*$.
9:       for $j=2,\dots,|V_C|$, $i'=0,\dots,i^*$, $\ell'=0,\dots,\ell^*$ do
10:         let $\mathcal{A}$ be the family of all sets $U \cup W$ such that $U \cap (W \cup V_C)=\emptyset$, and there are $0 \leq i'' \leq i'$ and $0 \leq \ell'' \leq \ell'$ satisfying $i''+\ell'' \leq \frac{k+t}{d}$ for which
            (1) $U \in \mathrm{M}[f_C(w_j),w_j,i'',\ell'']\}$ and $W \in \mathrm{L}[j-1,i'-i'',\ell'-\ell'']$; or
            (2) $w_j$ is not a leaf in $C$, $\ell'' \geq 1$, $U \in \mathrm{M}[w_j,w_j,i'',\ell'']\}$ and $W \in \mathrm{L}[j,i'-i'',\ell'-\ell'']$.
11:         $\mathrm{L}[j,i',\ell'] \Leftarrow \mathsf{RepAlg}(V,k+t,\mathcal{A})$.
12:      end for
13:      $\mathrm{N}[C] \Leftarrow \{U \cup (V_C \setminus \{v,u\}) : U \in \mathrm{L}[|V_C|,i^*,\ell^*]\}$.
14:   end for
15:   $\mathrm{M}[v,u,i,\ell] \Leftarrow \mathsf{RepAlg}(V,k+t,\bigcup_{C \in \mathcal{C}_{v,u}} \mathrm{N}[C])$.
16: end for
17: accept iff $\mathrm{M}[r,r,k-1,t] \neq \emptyset$.
```

and $O(2.851^k|V|^{O(1)})$, respectively. By replacing their computation of representative families with our scheme, RepAlg, we solve these problems in times $O(6.75^k|E|\log^2|V|)$, $O(2.619^k|V|\log^2|V|)$ and $O(2.619^k|V|^{O(1)})$, respectively.

Acknowledgment. We thank the anonymous referees for valuable comments.

References

1. Bläser, M.: Computing small partial coverings. Inf. Proc. Let. 85(6), 327–331 (2003)
2. Bollobás, B.: On generalized graphs. Acta Math. Aca. Sci. Hun. 16, 447–452 (1965)
3. Bonnet, E., Paschos, V.T., Sikora, F.: Multiparameterizations for max k-set cover and related satisfiability problems. CoRR abs/1309.4718 (2013)
4. Chen, S., Chen, Z.: Faster deterministic algorithms for packing, matching and t-dominating set problems. CoRR abs/1306.3602 (2013)
5. Cohen, N., Fomin, F.V., Gutin, G., Kim, E.J., Saurabh, S., Yeo, A.: Algorithm for finding k-vertex out-trees and its application to k-internal out-branching problem. J. Comput. Syst. Sci. 76(7), 650–662 (2010)
6. Demers, A., Downing, A.: Minimum leaf spanning tree. US Patent no. 6,105,018 (August 2013)
7. Fomin, F.V., Gaspers, S., Saurabh, S., Thomassé, S.: A linear vertex kernel for maximum internal spanning tree. J. Comput. Syst. Sci. 79(1), 1–6 (2013)

8. Fomin, F.V., Grandoni, F., Lokshtanov, D., Saurabh, S.: Sharp separation and applications to exact and parameterized algorithms. Algorithmica 63(3), 692–706 (2012)
9. Fomin, F.V., Lokshtanov, D., Panolan, F., Saurabh, S.: Representative sets of product families. CoRR abs/1402.3909 (2014)
10. Fomin, F.V., Lokshtanov, D., Raman, V., Saurabh, S.: Subexponential algorithms for partial cover problems. Inf. Proc. Let. 111(16), 814–818 (2011)
11. Fomin, F.V., Lokshtanov, D., Saurabh, S.: Efficient computation of representative sets with applications in parameterized and exact agorithms. In: SODA (see also: CoRR abs/1304.4626), pp. 142–151 (2014)
12. Garey, M.R., Johnson, D.S.: Computers and intractability: a guide to the theory of NP-completeness. W.H. Freeman, New York (1979)
13. Garey, M.R., Johnson, D.S., Stockmeyer, L.: Some simplified NP-complete problems. In: STOC, pp. 47–63 (1974)
14. Gutin, G., Razgon, I., Kim, E.J.: Minimum leaf out-branching and related problems. Theor. Comput. Sci. 410(45), 4571–4579 (2009)
15. Kneis, J.: Intuitive algorithms. RWTH Aachen University, pp. 1–167 (2009)
16. Kneis, J., Mölle, D., Rossmanith, P.: Partial vs. complete domination: t-dominating set. In: SOFSEM, pp. 367–376 (2007)
17. Koutis, I., Williams, R.: Limits and applications of group algebras for parameterized problems. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009, Part I. LNCS, vol. 5555, pp. 653–664. Springer, Heidelberg (2009)
18. Lovász, L.: Flats in matroids and geometric graphs. In: BCC, pp. 45–86 (1977)
19. Marx, D.: Parameterized coloring problems on chordal graphs. Theor. Comput. Sci. 351, 407–424 (2006)
20. Marx, D.: A parameterized view on matroid optimization problems. Theor. Comput. Sci. 410, 4471–4479 (2009)
21. Monien, B.: How to find long paths efficiently. Annals Disc. Math. 25, 239–254 (1985)
22. Ozeki, K., Yamashita, T.: Spanning trees: A survey. Graphs and Combinatorics 27(1), 1–26 (2011)
23. Pinter, R.Y., Shachnai, H., Zehavi, M.: Deterministic parameterized algorithms for the graph motif problem. In: MFCS (to appear, 2014)
24. Prieto, E., Sloper, C.: Reducing to independent set structure – the case of k-internal spanning tree. Nord. J. Comput. 12(3), 308–318 (2005)
25. Raible, D., Fernau, H., Gaspers, D., Liedloff, M.: Exact and parameterized algorithms for max internal spanning tree. Algorithmica 65(1), 95–128 (2013)
26. Salamon, G.: A survey on algorithms for the maximum internal spanning tree and related problems. Electronic Notes in Disc. Math. 36, 1209–1216 (2010)
27. Shachnai, H., Zehavi, M.: Representative families: a unified tradeoff-based approach. CoRR abs/1402.3547 (2014)
28. Skowron, P., Faliszewski, P.: Approximating the MaxCover problem with bounded frequencies in FPT time. CoRR abs/1309.4405 (2013)
29. Zehavi, M.: Algorithms for k-internal out-branching. In: Gutin, G., Szeider, S. (eds.) IPEC 2013. LNCS, vol. 8246, pp. 361–373. Springer, Heidelberg (2013)

A Branch and Price Procedure for the Container Premarshalling Problem

Martijn van Brink and Ruben van der Zwaan

Maastricht University, Maastricht, The Netherlands
m.vanbrink@maastrichtuniversity.nl, grjzwaan@gmail.com

Abstract. During the loading phase of a vessel, only the containers that are on top of their stack are directly accessible. If the container that needs to be loaded next is not the top container, extra moves have to be performed, resulting in an increased loading time. One way to resolve this issue is via a procedure called premarshalling. The goal of premarshalling is to reshuffle the containers into a desired lay-out prior to the arrival of the vessel, in the minimum number of moves possible. This paper presents an exact algorithm based on branch and bound, that is evaluated on a large set of instances. The complexity of the premarshalling problem is also considered, and this paper shows that the problem at hand is NP-hard, even in the natural case of stacks with fixed height.

1 Introduction

Enormous volumes of goods are shipped yearly all over the world in standardized containers. These containers typically require multiple modes of transportation to reach their destination. At container terminals, containers are transshipped between ships, trucks, and trains. This transshipment generally does not occur immediately upon delivery of a container, therefore containers are temporarily stored in an area called the container yard. The container yard consists of a set of blocks, which in turn consist of a set of bays. Each bay contains a number of rows, called stacks, with a certain height.

One main indicator of the efficiency of a container terminal is the berthing time of a vessel, which consists primarily of the time needed to store and load containers. During the storage, information on pick-up time and destination of the containers is often inaccurate or even unknown. This makes it difficult to obtain a storage sequence that permits an efficient loading sequence. Hence, during the loading phase it can occur that the container that needs to be retrieved next, is not on top of the stack. In this case, the containers on top of this container need to be rehandled, i.e., relocated within the container yard, before the desired container can be retrieved. These rehandle operations greatly increase the time needed to remove the container from the yard.

One way to resolve this issue is to reshuffle the containers prior to the arrival of the vessel. This operation is called *remarshalling*, and the goal is to find a sequence of rehandles, also called moves, of minimum length that reorganizes the stacks such that no container that is needed early is below a container that is

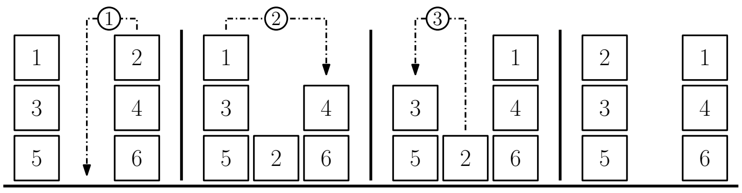


Fig. 1. Small example of three moves swapping the position of containers 1 and 2

needed late. This results in no rehandles during the loading phase, thus reducing the berthing time. The only valid move is to pick up the top container of one stack and put it on top of another stack, see Figure 1 for an illustration.

Two types of remarshalling operations can be identified, called *intra-block remarshalling* (hereafter called remarshalling) and *intra-bay premarshalling* (hereafter called premarshalling). The containers are reshuffled between bays for the former, and within a bay for the latter. The premarshalling variant is primarily applicable to container yards that use rail mounted gantry cranes. It is usually not allowed to move a container to another bay, as this operation is extremely time consuming [9].

Another main difference between remarshalling and premarshalling lies in the number of cranes that are used. For premarshalling typically only a single crane is used, while for remarshalling several cranes are often used simultaneously [3].

In this paper we focus on the premarshalling problem, and we follow the same assumptions as Bortfeldt and Forster [2]: (a) a single crane is used for rehandling containers, and (b) the time needed to move a container from one stack to another does not depend on the distance between the two stacks. This last assumption follows from the fact that the time needed to position the crane over a stack is negligible compared to the time needed to pick up or drop off a container. As a consequence, we are only interested in the number of rehandling operations.

We assume that for each container a priority level is given, and the goal is to transform the initial lay-out into a desired target lay-out in the minimum number of rehandles. The main problem studied is called PRIORITY STACKING: we accept all lay-outs in which no container with a lower priority is placed on top of a container with a higher priority. In a variant, called CONFIGURATION STACKING, we restrict the target lay-out to a single pre-specified lay-out. The main motivation to also look at CONFIGURATION STACKING is that giving a concrete target lay-out might give guidance to the algorithm and yield a faster computation time.

Related Literature. The operations at container terminals are well studied in the literature. Steenken, Voß, and Stahlbock [13] and Stahlbock and Voß [12] describe the most important processes and operations at container terminals and give an overview of methods to optimize these operations. Vis and De Koster [16] give a classification of the different decision problems that occur at container terminals, and give an overview of relevant literature. Lehnfeld and Knust [11] develop a classification scheme for loading, unloading and premarshalling problems that appear in several applications. This scheme is applied to existing literature.

While there is a vast amount of work on the logistics of container terminals, the number of publications on the premarshalling problem is limited. We are only aware of one paper that thoroughly investigates an exact algorithm. Lee and Hsu [10] develop a mixed integer linear program based on a multicommodity network flow formulation that solves both Priority Stacking and Configuration Stacking to optimality. However, this formulation can only be reasonably applied to very small instances, and the running time heavily depends on the choice of the number of time points. For larger instances the authors provide a heuristic that iteratively applies the exact approach on small parts of the instance. Integer multicommodity network flow is a generalization of edge-disjoint paths which cannot be approximated better than $\Omega(\sqrt{n})$, which immediately implies that the integrality gap of this formulation is at least that [7].

The remaining literature on the premarshalling problem is about the design of fast heuristics for Priority Stacking. Lee and Chao [9] describe a heuristic that minimizes the weighted sum of the mis-overlay index, which can be seen as a measure for the number of rehandles during the loading phase, and the number of rehandles during the premarshalling phase. Caserta and Voß [5] develop a heuristic based on the corridor method, where the basic idea is to use an exact method for limited portions of the entire solution space. Bortfeldt and Forster [2] describe a refined heuristic tree search procedure that looks at move sequences rather than individual moves. This heuristic is reported to be faster than the heuristic by Caserta and Voß. Huang and Lin [8] describe two heuristics that iteratively improve the lay-out of the yard. The second heuristic is applied to a special case of Priority Stacking, where all containers in a stack should be of the same priority level. Expósito-Izquierdo, Melián-Batista, and Morena-Vega [6] describe a heuristic that considers the container from lowest to highest priority. The considered container is moved to a position where it is not above a container with higher priority. The authors also provide an instance generator and describe an A^* search algorithm that provides the optimal solution for smaller instances.

Caserta, Schwarze, and Voß [3] give an overview of recent developments on three so-called post-stacking problems. Besides the remarshalling and premarshalling problem, the authors also consider the (intra-bay) blocks relocation problem. In addition to the premarshalling problem, containers need to be removed from the bay in a certain order that minimizes the number of rehandles. It was proven that this problem is NP-hard, but with arbitrarily high stacks [4]. This proof also works as a proof that Priority Stacking is NP-hard with arbitrarily high stacks. To the best of the authors knowledge there are no results about the natural case when stack heights are bounded by a constant. Typical stack heights are between 2 and 8 containers, while currently used equipment can handle a stack height of at most 10 containers [12], [16].

Our Contributions. We develop a fast exact algorithm based on column generation for the premarshalling problem and evaluate it extensively. To the best of our knowledge, we are the first to extensively experiment with an exact algorithm. Expósito-Izquierdo, Melián-Batista, and Morena-Vega [6] describe an A^* search algorithm, but its results are only used as a benchmark for their heuristic. Lee and Hsu [10] also design an exact algorithm, but only evaluate it on two instances. Our algorithm is evaluated on 960 instances, with roughly 70% of the instances solved within one second. We also see that our method exhibits a low integrality gap. Finally, we consider the complexity of Priority Stacking and Configuration Stacking. Current NP-hardness proofs require a stack height that depends on the number of containers. We strengthen this results by showing that both problems are already NP-hard for all fixed heights at least six, which resembles the real-life situation.

Organization. In Section 2 we introduce notation and formally describe the premarshalling problem. In Section 3 we describe an ILP formulation and an oracle for finding variables in a column generation approach. This is then used in Section 4 to design a branch and price algorithm, whose experimental performance is analyzed in Section 5. In Section 6 we consider the complexity of both premarshalling variants. Finally, some conclusions are drawn in Section 7.

2 Preliminaries

Let $[n]$ denote the set of integers from 1 up to n, i.e., $[n] := \{1, \ldots, n\}$. The premarshalling problem is defined as follows. Given are m stacks of maximum height h and n containers, each container labeled with a priority ℓ from $[k]$ $(2 \leq k \leq n)$. In line with the definitions used in the literature, a lower priority number indicates a higher priority level, i.e., containers with priority 1 are needed first, and containers with priority k last. The *lay-out* of a stack i with $j \leq h$ containers is denoted as an ordered set of priorities $X_i := \{x_1, \ldots, x_j\}$, where the first element is the priority of the bottom container and the last element is the priority of the top container. Notice that containers with the same priority are indistinguishable, therefore we will abbreviate "move container with priority ℓ" to "move container ℓ".

The goal is to transform the initial lay-out to a target lay-out by performing the minimum number of moves, while adhering to the maximum stack height. A move is defined as picking up the top-most container of one stack and placing it on top of another stack. For Priority Stacking the set of target lay-outs consists of all lay-outs such that all stacks are sorted in non-increasing order when viewed from the bottom, i.e., for a stack i with $X_i := \{x_1, \ldots, x_j\}$, we have that $x_{p+1} \leq x_p$ for $p = 1, \ldots, j-1$. For Configuration Stacking there is only one target lay-out, which is specified beforehand.

3 Formulation as an ILP

In this section we describe the linear program model that we use for both Priority Stacking and Configuration Stacking. Let us first introduce some notation. Let T denote the number of time points. As only a single move is allowed per time point, T can also be viewed as the maximum possible number of moves. Let the tuple (ℓ, t) denote a move of container ℓ at time $t \in [T]$, and consider stack s. Let L_s^{add} and L_s^{rem} contain moves (ℓ, t) such that container ℓ is respectively added to, or removed from, stack s at time t, and let $L_s := (L_s^{\mathsf{add}}, L_s^{\mathsf{rem}})$. The set L_s is *feasible* if (a) at every time t there is at most one move, i.e., either a container is added, a container is removed, or no move is performed; (b) for all moves $(\ell, t) \in L_s^{\mathsf{add}}$ and $(\ell, t) \in L_s^{\mathsf{rem}}$ we have that at time t stack s contains at least one free spot, or container ℓ is the top container of stack s, respectively; (c) the lay-out obtained by executing the moves is a target lay-out. Hence, a feasible L_s can be viewed as a sequence of moves that transforms stack s into a target lay-out, where L_s^{add} and L_s^{rem} consist of the moves where a container is added or removed, respectively. Furthermore, let $\#\mathsf{moves}(L_s)$ denote the number of containers added in L_s, i.e., $\#\mathsf{moves}(L_s) := |L_s^{\mathsf{add}}|$. Finally, let $\mathcal{L}_s$ denote the set of sequences that transform stack s into a target lay-out, i.e., $\mathcal{L}_s := \{L_s : L_s \text{ is feasible}\}$. For an example of a feasible set of sequences, see Figure 2.

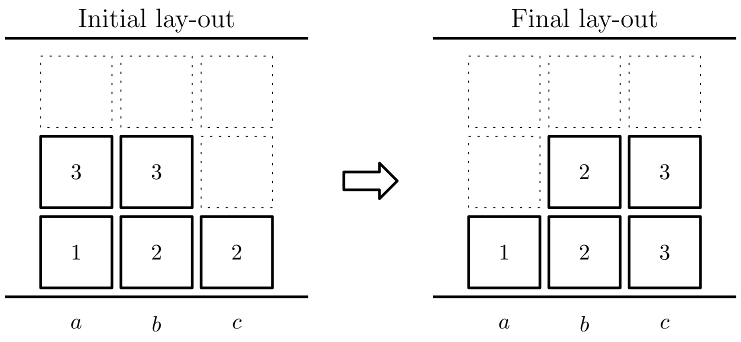


Fig. 2. For ease of reference, the stacks are called a, b, and c. The final lay-out can be obtained by moving container 2 from stack c to stack a, 3 from b to c, 2 from a to b, and 3 from a to c. Let L_a, L_b, and L_c denote the sequence for stack a, b, and c, respectively. For these moves we get $L_a^{\mathsf{add}} = \{(2,1)\}$, $L_a^{\mathsf{rem}} = \{(2,3),(3,4)\}$, $L_b^{\mathsf{add}} = \{(2,3)\}$, $L_b^{\mathsf{rem}} = \{(3,2)\}$, $L_c^{\mathsf{add}} = \{(3,2),(3,4)\}$, and $L_c^{\mathsf{rem}} = \{(2,1)\}$.

We consider the following ILP, where $x_{s,L} \in \{0,1\}$ for $L \in \mathcal{L}_s$ has value 1 if and only if stack s is transformed according to sequence L. Let $\mathsf{add}(L,\ell,t)$ and $\mathsf{rem}(L,\ell,t)$ be equal to 1 if and only if $(\ell,t) \in L^{\mathsf{add}}$ and $(\ell,t) \in L^{\mathsf{rem}}$, respectively.

$$\begin{array}{llll}
\min & \sum_{s\in[m]} \sum_{L\in\mathcal{L}_s} \#\mathsf{moves}(L)\; x_{s,L} & & \text{(Integer Linear Program)} \\
\text{s.t.} & \sum_{s\in[m]} \sum_{L\in\mathcal{L}_s} (\mathsf{add}(L,\ell,t) - \mathsf{rem}(L,\ell,t))\, x_{s,L} \geq 0 & \forall \ell \in [k], t \in [T] & \text{(C1)} \\
 & \sum_{s\in[m]} \sum_{L\in\mathcal{L}_s} \sum_{\ell\in[k]} \mathsf{add}(L,\ell,t) x_{s,L} \leq 1 & \forall t \in [T] & \text{(C2)} \\
 & \sum_{L\in\mathcal{L}_s} x_{s,L} = 1 & \forall s \in [m] & \text{(C3)} \\
 & x_{s,L} \in \{0,1\} & \forall s \in [m], L \in \mathcal{L}_s. &
\end{array}$$

The variables themselves already ensure that only sequences of moves are chosen that *for every stack individually* are feasible. The remaining constraints ensure that the local solutions together form a valid global solution. Constraint (C1) ensures that at time t the number of containers of priority ℓ that are added is at least the number of containers of priority ℓ that are removed. Constraint (C2) enforces that at any time t at most one container can be added. Note that this implies that at most one container is moved per time point. Constraint (C3) makes sure that exactly one sequence is selected for each stack. By relaxing the requirement that the variables are either 0 or 1 we obtain the LP relaxation.

Solving the Subproblem. The problem of finding variables with negative reduced costs is almost equivalent to finding a maximum weight independent set in a circle graph, which can be solved in polynomial time by dynamic programming [1], [14]. A circle graph is an intersection graph of chords of a a circle, two vertices/chords are adjacent if and only if they intersect. Circle graphs can be equivalently defined as the overlap graph of a set of intervals. The additional constraint that we impose is that in the solution there are never more than h intersecting intervals, corresponding to the height constraint. However, the algorithm to find the maximum weight independent set by dynamic programming can be easily adapted to take this constraint into account.

We will shortly describe how the problem of finding a sequence of moves with negative reduced costs for a fixed stack s can be cast as finding a non-overlapping set of (labeled) intervals. For a more detailed description we refer the reader to the full version of this paper [15]. For ease of exposition ignore the conditions on the initial and target lay-out of a stack and assume that all endpoints of the intervals are distinct. Let an interval $[a,b]$ with label ℓ mean that a container with priority ℓ is added to stack s at time a and removed at time b. Having two overlapping intervals $[a,b]$ and $[c,d]$ such that $a < c < b < d$ is interpreted as putting a container down at time a, putting another container on top at time c

and removing the first container at time b *while the second container is still there.* Clearly this is infeasible. However, if $a < c < d < b$, then the second container would be put on top of the first, just as before, but *it would be removed before the first is removed.* Therefore, given a non-overlapping set of intervals whose endpoints are distinct, we have found a feasible sequence of moves.

4 Branch and Price Algorithm

The premarshalling problem is solved by iteratively running the algorithm with an increasing number of time points, starting from some lower bound, until the optimal solution is found. See Algorithm 1 for an overview of the procedure.

Lower Bound. We say that a container is *wrongly* placed if it is positioned on top of a container that (a) has a higher priority, or (b) is itself wrongly placed. Clearly, all wrongly placed containers need to be moved to obtain a target layout. However, if all stacks contain a wrongly placed container, then moving one does not reduce the number of wrongly placed containers. This number can only be reduced if at least one stack does not contain wrongly placed containers. The minimum number of moves required for this is equal to the lowest number of wrongly placed containers over all stacks. As lower bound on the number of moves we take the number of wrongly placed containers plus the minimum number of wrongly placed containers over all stacks.

Solving Nodes. For each node in a tree, T time points are available to move containers. For solving a single node, we start with a model that contains (if any) previously generated sequences, and for each stack a dummy sequence with cost $T+1$, that performs no moves. These dummy sequences ensure that a feasible solution for the LP relaxation always exists. This initial model is solved, and as long as there are sequences with negative reduced cost, they are added and the model is resolved. Note that at each iteration at most one sequence is added per stack. If there are no more sequences with negative reduced cost, and the LP value strictly exceeds T, we discard the node. If the LP value does not exceed T, we check if the solution is integral. If it is integral, we have found the optimal solution, and we stop the solve procedure. Otherwise, we apply the branching rule and continue with the next node. Every 100 nodes the sequence pool is cleaned. All sequences that have not been used since the last cleanup, i.e., whose corresponding variable had value zero in all LP models since the last cleanup, are discarded.

Branching and Node Selection Rule. Consider an arbitrary stack s and time point t, and observe that for this combination three actions are possible. Either (a) a container is added, (b) a container is removed, or (c) no move is performed. When applying branching, one of the three actions is forced for stack s at time t. For instance, if action (a) is forced, only sequences that add a container (of any priority level) to stack s at time t are still considered. Note that at most one container is moved per time point. Hence, if adding (removing) a container is forced for stack s at time t, this action is no longer feasible for any other stack at time t.

The stack and time point on which are branched, are determined as follows. Let t be the minimum time point such that at least one of actions (a) and (b) is not forced for any stack at time t. Hence, t is the minimum time point for which the exact move is not yet fixed. Out of all the stacks for which no action is forced at time t, we take the stack s for which the sum of all $x_{s,L}$ variables, such that for sequence L a move is performed at time t, is maximized, i.e., for which $\sum_{L \in \mathcal{L}_s} \sum_{\ell \in [k]} (\mathsf{add}(L, \ell, t) + \mathsf{rem}(L, \ell, t)) x_{s,L}$ is maximized. In case of a tie, we take the stack s that was considered first. By appropriately removing intervals, or adapting their value, this branching rule does not affect the difficulty of applying the separation oracle described in Section 3.

For exploring the tree we apply a depth first search. The node for which t is maximized, is considered next. In case of a tie, we take the most recently generated node.

Algorithm 1. Branch and Price algorithm

```
procedure PREMARSHAL
  set $T$ equal to the lower bound for the number of moves
  while optimal solution not found do
    start with tree consisting of only a root node
    while exist unpruned leaf node do
      $N$ := leaf node deepest in tree, initialize LP model with dummy and "valid"
      sequences, solve LP model with column generation, update sequence pool
      if LP value $> T$ then prune node $N$
      else if solution integral then output opt. solution & prune all nodes
      else let $(s, t)$ denote the stack and time to branch on, add children
            $N_1/N_2/N_3 := N$ with Add / Remove / Nothing fixed for $(s, t)$
      every 100 nodes clean sequence pool
    $T := T + 1$
```

5 Experimental Results

In this section we evaluate the branch and price algorithm described in Section 4. We impose a time limit of one hour for each instance, and we only consider results for Priority Stacking.

5.1 Experimental Setup

The algorithm is implemented in C++ in combination with CPLEX 12.6, run on a machine with an Intel Core 2 Duo E8400 3.00 GHz processor and 4 GB RAM, and evaluated on randomly generated instances. To the best of our knowledge no library with real-life instances for the premarshalling problem exists, and randomly generated instances are also used in for instance [2], [5].

The instances depend on four input parameters: the number of different priorities (*Priorities*), the number of stacks (*Stacks*), the height of the stacks (*Height*), and the fill grade (*Fill*). For possible values for Priorities we consider [10]. To the best of our knowledge, this is the only other paper that evaluates an exact algorithm for the premarshalling problem. The authors basically consider two instances, with 3 and 6 priority levels, respectively. Therefore we consider 2 (the minimum value possible), 3 and 6 priority levels. For the other parameters, Lee and Chao [9] observe that 12 stacks with a height of 6 is already larger than most equipment can handle, and a fill grade of 75% is considered moderately high. A minimum height of 4 is observed in general. As we apply an exact method, we consider slightly lower parameter values. For Stacks we take values 3, 5, 7, and 9, for Height we consider 4 and 6, and for Fill we take either 50% or 70%, which we consider a low and average fill grade, respectively. The number of containers is determined by multiplying the number of available positions, i.e., Stacks times Height, with Fill. In case this number is fractional, it is truncated.

First, consider the case where Priorities has value 6. Consider the containers one by one, and consecutively assign them priority 1 to 6. The initial lay-out is determined by randomly picking a container and placing it on a randomly selected non-full stack, until all containers are placed. Second, consider the case where Priorities has value 2 or 3. In this case, the instances are based on the ones where Priorities has value 6. Each stack has the same number of containers, but the priorities are updated. For Priorities equal to 2, the three lowest and the three highest priorities are grouped together. For Priorities equal to 3 the lowest two, middle two, and highest two priorities are grouped together. If for any of the three values for Priorities the instance does not contain a wrongly placed container, i.e., no premarshalling operations are necessary, all three instances are discarded. This procedure is repeated until 20 instances are generated for all 48 combinations of parameter values, resulting in 960 instances.

5.2 Results

Table 1 contains an overview of the results. The first column contains the results for all instances, while columns two through five contain the results for instances with a running time less than one second, between one second and one minute, between one minute and one hour, and more than one hour, respectively. For all statistics the average is given, except for # instances and seq. memory, which indicate the total and maximum value, respectively. If a cell contains two values, then the first one indicates the average and the second one the maximum.

Over all 960 instances, the average mis-overlay is 38.3%. The mis-overlay of a lay-out is defined as the total number of wrongly placed containers (see Section 4). Note that this definition corresponds to the one used in for instance [9]. Clearly, the higher the value for Priorities, Height, and Fill, the higher the mis-overlay. For Stacks the value of mis-overlay is constant. This follows from the

Table 1. Overview of the results

statistic	all	< 1 sec	1 sec - 1 min	1 min - 1 h	> 1 h
# instances	960	680	215	50	15
mis-overlay (%)	38.3	32.3	50.7	58.9	63.4
integrality gap	1.13 | 2.80	1.09 | 2.80	1.19 | 2.43	1.32 | 2.09	1.18 | 1.85
trees solved	2.45 | 11	1.90 | 6	3.40 | 8	5.12 | 11	5.07 | 9
trees killed	0.82 | 4	0.54 | 3	1.41 | 4	1.72 | 4	2.13 | 4
trees actually	1.63 | 8	1.36 | 5	1.99 | 6	3.40 | 8	2.93 | 7
run time (sec.)	93.65	0.23	8.42	678.39	3600.84
lp time (sec.)	46.66	0.05	3.02	327.05	1850.74
gen. time (sec.)	34.51	0.10	4.60	266.16	1250.63
clear time (sec.)	11.67	0.00	0.60	79.81	472.30
nodes solved	295.7	8.4	117.0	2569.2	8301.2
nodes memory	4.78 | 56	2.83 | 20	8.67 | 39	11.30 | 56	15.50 | 39
seq. generated	14,396	271	4,240	130,542	413,156
seq. memory	59,729	1,641	19,536	56,554	59,729

way the instances are generated: the number of containers is (almost) linear in the number of stacks, and for each container a random stack is selected.

Out of all the instances, 945 (98.4%) are solved within one hour, 895 (93.2%) within one minute, and 680 (70.8%) within one second. The 15 instances that are not solved within one hour all have value 6 for Priorities, value 6 for Height, and value 70 for Fill. The number of unsolved instances is 1, 4, 4, and 6 for 3, 5, 7, and 9 stacks, respectively.

The average running time is 93.65 seconds. The main contributors are the time spend on solving the LP relaxation (46.66 seconds), generating columns (34.51 seconds), and clearing the cplex models (11.67 seconds). This leaves on average only 0.80 seconds of overhead. If we compare the solved and unsolved instances, we observe little difference in the relative amount spend on solving the LP relaxation (47.5% versus 51.4%), generating columns (40.0% versus 34.7%), and clearing the cplex model (11.5% versus 13.1%).

For the solved instances the integrality gap is on average 1.13. The integrality gap is obtained by comparing the optimal (integer) solution with the value of the LP relaxation at the root node of the tree that contains the optimal solution. This LP value is the lowest one obtained, as adding forced actions and time points respectively increases and decreases the LP value. For the 15 unsolved instances we cannot determine the integrality gap, but we can give a lower bound. For these instances the average lower bound on the integrality gap is 1.18. Although this values are biased, there does not appear to be a big difference in integrality gap between the solved and unsolved instances.

On average 2.41 trees are solved. Out of these trees, on average 0.80 are *killed immediately.* A tree is killed immediately if the LP value of the root node exceeds the number of time points. As only the root node is solved for killed trees, solving these trees generally requires little time. Hence, on average 1.61 trees are *actually* solved per instance. The average number of solved nodes is 295.7. On average

0.32 seconds are needed to solve one node. This average time is higher for the unsolved instances compared to the solved instances (0.23 versus 0.43 seconds).

The average and maximum number of nodes in memory is 4.8 and 56, respectively. The maximum number of nodes in memory is obtained by a solved instance. Hence, the memory usage with respect to the number of nodes is low and stable over time, i.e., longer running times do not lead to a huge increase in the number of nodes in the memory. Maximally 59, 729 sequences are stored in the memory. In this case, the maximum is obtained by an unsolved instance. Here again there does not appear to be a huge increase in memory usage with increasing running time. Because unused sequences are discarded, the number of sequences stored in the memory is kept at an acceptable level, at the expense of possibly generating the same sequence several times.

6 Complexity: NP-Hard for Constant Height Stacks

Because of limited space, we only state our results that both stacking problems are NP-hard for constant height stacks. For the proof we refer the reader to the full version of this paper [15]. We would like to point out that the proof for Configuration Stacking is significantly more involved and is not implied by the proof for Priority Stacking.

Theorem 1. *For every fixed $h \geq 6$,* Priority Stacking *and* Configuration Stacking *are NP-hard.*

7 Conclusion

We considered the intra-bay premarshalling problem. The objective is to transform the initial lay-out into a target lay-out in the minimum number of moves possible. We showed that the premarshalling problem is NP-hard, even for a fixed stack height of at least six. We developed an exact algorithm based on branch and price, that is evaluated on 960 randomly generated instances. For Priority Stacking, 945 are solved within one hour, 895 within one minute, and 680 with one second. Preliminary experiments show that the algorithm runs much faster for Configuration Stacking, in the full version of the paper there will be a comparison. An interesting topic for future research concerns the solution approach. Currently, either the optimal solution is found, or no solution is found at all. By for instance incrementing the number of time points T (see Section 4) with more than one, it might be possible to look for other feasible (near-optimal) solutions. Another option would be to allow multiple moves per time point. A restriction would be that each stack is involved in at most one move per time point. This decreases the number of time points, which hopefully results in a reduced running time.

References

1. Bonsma, P., Breuer, F.: Counting hexagonal patches and independent sets in circle graphs. Algorithmica 63(3), 645–671 (2012)
2. Bortfeldt, A., Forster, F.: A tree search procedure for the container pre-marshalling problem. European Journal of Operational Research 217(3), 531–540 (2012)
3. Caserta, M., Schwarze, S., Voß, S.: Container rehandling at maritime container terminals. In: Böse, J.W. (ed.) Handbook of Terminal Planning. Operations Research/Computer Science Interfaces Series, vol. 49, pp. 247–269. Springer, New York (2011)
4. Caserta, M., Schwarze, S., Voß, S.: A mathematical formulation and complexity considerations for the blocks relocation problem. European Journal of Operational Research 219(1), 96–104 (2012)
5. Caserta, M., Voß, S.: A corridor method-based algorithm for the pre-marshalling problem. In: Giacobini, M., Brabazon, A., Cagnoni, S., Di Caro, G.A., Ekárt, A., Esparcia-Alcázar, A.I., Farooq, M., Fink, A., Machado, P. (eds.) EvoWorkshops 2009. LNCS, vol. 5484, pp. 788–797. Springer, Heidelberg (2009)
6. Expósito-Izquierdo, C., Melián-Batista, B., Moreno-Vega, M.: Pre-marshalling problem: Heuristic solution method and instances generator. Expert Systems with Applications 39(9), 8337–8349 (2012)
7. Guruswami, V., Khanna, S., Rajaraman, R., Shepherd, B., Yannakakis, M.: Near-optimal hardness results and approximation algorithms for edge-disjoint paths and related problems. Journal of Computer and System Sciences 67(3), 473–496 (2003)
8. Huang, S.-H., Lin, T.-H.: Heuristic algorithms for container pre-marshalling problems. Computers & Industrial Engineering 62(1), 13–20 (2012)
9. Lee, Y., Chao, S.-L.: A neighborhood search heuristic for pre-marshalling export containers. European Journal of Operational Research 196(2), 468–475 (2009)
10. Lee, Y., Hsu, N.-Y.: An optimization model for the container pre-marshalling problem. Computers & Operations Research 34(11), 3295–3313 (2007)
11. Lehnfeld, J., Knust, S.: Loading, unloading and premarshalling of stacks in storage areas: Survey and classification. European Journal of Operational Research (to appear, 2014)
12. Stahlbock, R., Voß, S.: Operations research at container terminals: a literature update. OR Spectrum 30(1), 1–52 (2008)
13. Steenken, D., Voß, S., Stahlbock, R.: Container terminal operation and operations research - a classification and literature review. OR Spectrum 26(1), 3–49 (2004)
14. Valiente, G.: A new simple algorithm for the maximum-weight independent set problem on circle graphs. In: Ibaraki, T., Katoh, N., Ono, H. (eds.) ISAAC 2003. LNCS, vol. 2906, pp. 129–137. Springer, Heidelberg (2003)
15. van Brink, M., van der Zwaan, R.: A branch and price procedure for the container premarshalling problem (2014), `http://arxiv.org/abs/1406.7107`
16. Vis, I.F.A., de Koster, R.: Transshipment of containers at a container terminal: An overview. European Journal of Operational Research 147(1), 1–16 (2003)

Space-Efficient Randomized Algorithms for K-SUM

Joshua R. Wang

Stanford University, Stanford CA 94305, USA
joshua.wang@cs.stanford.edu

Abstract. Recent results by Dinur et al. (2012) on random SUBSET-SUM instances and by Austrin et al. (2013) on worst-case SUBSETSUM instances have improved the long-standing time-space tradeoff curve. We analyze a family of hash functions previously introduced by Dietzfelbinger (1996), and apply it to decompose arbitrary k-SUM instances into smaller ones. This allows us to extend the aforementioned tradeoff curve to the k-SUM problem, which is SUBSETSUM restricted to sets of size k. Three consequences are:

– a Las Vegas algorithm solving 3-SUM in $O(n^2)$ time and $\tilde{O}(\sqrt{n})$ space,

– a Monte Carlo algorithm solving k-SUM in $\tilde{O}(n^{k-\sqrt{2k}+1})$ time and $\tilde{O}(n)$ space for $k \geq 3$, and

– a Monte Carlo algorithm solving k-SUM in $\tilde{O}(n^{k-\delta(k-1)} + n^{k-1-\delta(\sqrt{2k}-2)})$ time and $\tilde{O}(n^{\delta})$ space, for $\delta \in [0, 1]$ and $k \geq 3$.

Keywords: k-sum, subset-sum, hashing, time-space tradeoffs.

1 Introduction

The k-SUM problem on n numbers is as follows: Given k sets $S_1, S_2, \ldots, S_k$ with at most n integers each and a target t, find $a_1, a_2, \ldots, a_k$ such that for all i, $a_i \in S_i$ and $\sum_{i=1}^{k} a_i = t$. One common variant of the problem has only a single set S from which all elements in the solution are chosen from, but the two are easily reducible to each other. The k-SUM problem can be trivially solved in $O(n^k)$ arithmetic operations by trying all possibilities, and a more sophisticated solution runs in $O(n^{\lceil k/2 \rceil} \log n)$ time (this log factor can be avoided for k odd). However, this solution also requires $O(n^{\lceil k/2 \rceil})$ space, while the trivial solution only needed $O(1)$ space. Is some trade-off between time and space possible? Schroeppel an Shamir [12] provide an algorithm for 4-SUM that runs in $\tilde{O}(n^2)$ time[1] and $\tilde{O}(n)$ space. In a survey of the time and space complexity of exact algorithms, Woeginger [13] studied the k-SUM problem and questioned whether an algorithm similar to the Schroeppel-Shamir 4-SUM algorithm can be constructed for 6-SUM.

[1] See Section 2.1 for an explanation of $\tilde{O}$ and O^* notation.

Gajentaan and Overmars [6] classified many problems from computational geometry as "3-Sum-hard" (i.e. there exists a $o(n^2)$ reduction from 3-Sum to the problem in question) in order to indirectly demonstrate their difficulty. Finding a subquadratic algorithm for any problem in this class of problems would immediately produce a subquadratic algorithm for 3-Sum. One example of such a problem is 3-POINTS-ON-LINE: Given a set of points in the plane, are there three collinear points? To reduce 3-Sum to this problem, map each $x \in S$ (using the single-set variation of 3-Sum) to the point (x, x^3), with the idea that $a_1 + a_2 + a_3 = 0$ iff the points (a_1, a_1^3), (a_2, a_2^3), and (a_3, a_3^3) are collinear.

k-Sum is also fundamentally connected to several NP-hard problems. For example, Pătraşcu and Williams [11] show that solving k-Sum over n numbers in $n^{o(k)}$ time would imply that 3-Sat with n variables can be solved in $2^{o(n)}$ time. Schroeppel and Shamir [12] showed how the SubsetSum problem can be reduced to an (exponential-sized) k-Sum problem (Recall that in SubsetSum, we are given a set S of n integers and a target t, and want to find a subset $S' \subseteq S$ such that $\sum_{a \in S'} a = t$). Therefore, more efficient k-Sum algorithms can be used to derive faster SubsetSum algorithms. Indeed, Schroeppel and Shamir use their 4-Sum algorithm to produce a $O^*(2^{0.5n})$ time and $O^*(2^{0.25n})$ space SubsetSum algorithm. They also showed that SubsetSum is solvable in time T and space S where $T \cdot S^2 = O^*(2^n)$ for $T(n) \geq \Omega^*(2^{n/2})$.

This 30-year old time-space tradeoff for SubsetSum was recently improved. In 2010, Howgrave-Graham and Joux [7] derived an algorithm for random Subset-Sum instances that runs in time $O^*(2^{0.337n})$ and memory $O^*(2^{0.256n})$. Becker, Coron, and Joux [3] then derived two algorithms for random instances, one running in time $O^*(2^{0.291n})$ and space $O^*(2^{0.291n})$ and one running in $O^*(2^{0.72n})$ time and $O^*(1)$ space. Dinur et al. [5]. presented a time-space tradeoff curve that dominates the Schroeppel-Shamir curve and matches it at its endpoints, for random instances. Austrin et al. [1] matched the Dinur curve *for worst case instances* with a randomized algorithm.

1.1 Our Results

The best known algorithm for 3-Sum takes $\tilde{O}(n^2)$ time (Baran, Demaine, and Pătraşcu [2] have found polylogarithmic improvements over $O(n^2)$ which were further improved by Gronlund and Pettie [9]), but also requires $\tilde{\Omega}(n)$ space (to hold a sorted copy of the input). Can the same running time be achieved with significantly less space? The primary difficulty here lies in the unsortedness of the input. What about for k-Sum in general? In his 2004 survey [13], Woeginger asked such questions, with the hopes of encouraging further progress on solving SubsetSum.

In this paper, we lower the space requirement for 3-Sum while maintaining the same running time, with a zero-error randomized algorithm:

Theorem 1. *The* 3-Sum *problem on* n *numbers can be solved by a Las Vegas* [2] *algorithm in time* $O(n^2)$ *and space* [3] $\tilde{O}(\sqrt{n})$.

We also investigate time-space tradeoffs for the general k-Sum problem. Given a fixed space budget S, for what T can k-Sum be solved in time T and space S? We prove the following general self-reduction for k-Sum:

Theorem 2. *Let* A *be a Las Vegas algorithm that solves* k-Sum *(*$k \geq 3$*) on* n *numbers in* $T(n)$ *time and* $S(n)$ *space where* $T(n), S(n) \in poly(n)$*, and let* $\delta \leq 1$ *be an arbitrary constant. Then there is a Las Vegas algorithm* A' *that solves* k-Sum *on* n *numbers in* $O(n^{k-\delta(k-1)} + n^{k-\delta(k-1)-1}T(n^\delta))$ *time and* $O(n^\delta + S(n^\delta))$ *space.*

When $S(n) = \tilde{O}(n)$, the reduction of Theorem 2 optimizes its space usage.

Independently of Theorem 2, we also provide a family of Monte Carlo randomized algorithms for k-Sum that use linear space.

Theorem 3. *There exists a Monte Carlo algorithm that solves* k-Sum *on* n *numbers in* $\tilde{O}(n^{k-f(k)-1})$ *time and* $\tilde{O}(n)$ *space, where* $f(k)$ *is the largest integer* p *such that the* $(p+1)$*-th triangular number is at most one less than* k.

Theorem 3 provides algorithms for: 4-Sum in $\tilde{O}(n^2)$ time, 5-Sum in $\tilde{O}(n^3)$ time, 6-Sum in $\tilde{O}(n^4)$ time, 7-Sum in $\tilde{O}(n^4)$ time, and so on, *all in linear space.* Note that the 4-Sum time matches the Schroeppel-Shamir [12] 4-Sum result.

The actual savings over $O(n^k)$ time in Theorem 3 are subtle, and studied in detail later in the paper (for now, we note that $f(k) \geq \sqrt{2k} - 2$). Theorem 3 strengthens the time-space tradeoff results of Dinur et al. and Austrin et al. in the following sense. Applying the original Schroeppel-Shamir reduction from SubsetSum to k-Sum, and running the algorithm of Theorem 3, we can recover the endpoints of the time-space tradeoff curve previously obtained. In particular, this occurs when k is one more than a triangular number.

Combining Theorem 2 and Theorem 3, we obtain the following time-space tradeoff curve for k-Sum in the "sub-linear" space setting:

Corollary 1. *Let* $\delta \in [0,1]$ *be an arbitrary constant. There is a Monte Carlo algorithm that solves* k-Sum *on* n *numbers in* $\tilde{O}(n^{k-\delta(k-1)} + n^{k-1-\delta f(k)})$ *time and* $\tilde{O}(n^\delta)$ *space, where* $f(k)$ *is the largest integer* p *such that the* $(p+1)$*-th triangular number is at most one less than* k.

1.2 Intuition

To illustrate some of the ideas in this paper, let us consider 3-Sum. The naive algorithm checks all triples of numbers to see if they sum to the target, in $O(n^3)$

[2] Recall that algorithms are Las Vegas randomized if they always give correct results, but may take additional running time depending on the random numbers generated (but not depending on the choice of input).

[3] We consider a model of computation where the input is given in read-only memory while the machine works in read/write memory, measuring the space usage by the *working memory* size.

time. Note that choosing two numbers in the solution determines the third. A more careful algorithm will store the third set in a data structure so that after choosing the first two numbers of the solution, the third can be quickly checked. Because *the last number is determined*, this only requires $O(n^2)$ time.

Now consider algorithms that only use $\tilde{O}(\sqrt{n})$ space. One naive approach is to partition each set into $\sqrt{n}$ buckets of $\sqrt{n}$ numbers and solve 3-Sum on all triples of buckets. There are $n^{1.5}$ such triples, and since solving 3-Sum on these smaller instances will take $O(n)$ time, the running time of this naive algorithm is $O(n^{2.5})$. However, this algorithm does redundant work checking buckets, for the same reason that the $O(n^3)$ algorithm does redundant work checking numbers: both check all possible triples.

We avoid this work via hashing. We apply a particular hash family H^{lin} of Dietzfelbinger [4] to create our buckets. We show that with this hash family, choosing the first two buckets *fixes the third bucket*. Hence we now only check $O(n)$ triples of buckets, and the running time drops to $O(n^2)$. With H^{lin}, we can ensure that the sizes of the buckets also remain $O(\sqrt{n})$, to avoid increasing the running time for solving a subproblem.

We can generalize this technique in two different ways. Firstly, this technique works for general k: guessing the first $(k-1)$ buckets *fixes the last bucket* in a general k-Sum solution. Secondly, it works for sizes other than $O(\sqrt{n})$, although additional work is necessary to guarantee the bucket size. In fact, this generalization yields a self-reduction for k-Sum problems, since the resulting subproblems are smaller instances of k-Sum.

The running time and space usage of the final k-Sum algorithm depend on how the subproblems from the self-reduction are solved. The space usage in the self-reduction is optimized when a linear-space k-Sum algorithm is applied to the resulting subproblems. Hence we take particular interest in linear-space k-Sum algorithms and derive faster linear-space k-Sum algorithms, by adapting SubsetSum techniques to the k-Sum setting.

1.3 Organization

In Section 2, we discuss some basic notation and several standard k-Sum algorithms. In Section 3, we study a family of hash functions, introduced by Dietzfelbinger [4], and analyze its properties. In particular, we show the family is "almost-affine". In Section 4, we use this hash family to develop a self-reduction on k-Sum problems to reduce memory usage. In Section 5, we analyze the k-Sum time-space tradeoff curves produced by this reduction.

In Appendix A, we present the proof for our general k-Sum self-reduction theorem. In Appendix B, we use the previously mentioned hash family to also derive linear-space Monte Carlo algorithms for k-Sum.

2 Preliminaries

2.1 Randomized Algorithms and Running Time

This paper describes both Las Vegas and Monte Carlo randomized algorithms. Las Vegas algorithms always give correct results, but their running times hold in expectation over internal randomness (the input is still worst-case). Monte Carlo algorithms may give incorrect results with some (small) probability, but their running time is deterministic and worst-case. It is worth noting that a Las Vegas algorithm can be converted into a Monte Carlo algorithm with the same running time up to a constant factor, via a Markov bound.

We use $\tilde{O}$ to indicate suppression of polylog factors, and O^* to indicate suppression of polynomial factors.

When determining running time, we will use the standard word RAM model, assuming that operations on integers take constant time. Note that polylog time operations would still fold into the $\tilde{O}$ notation and do not affect the polynomial exponent, which is the primary focus of this paper.

2.2 Sets and Triangular Numbers

Definition 1. $[m]$ *denotes the set* $\{0, 1, \ldots, m-1\}$.

Definition 2. *Given sets S and T, the Minkowski sum of S and T, denoted $S+T$, is defined as the set $\{s+t \mid s \in S, t \in T\}$.*

Definition 3. *Given a set S and a function f that can operate on the elements of S, the image of S under f, denoted $f(S)$, is defined as the set $\{f(s) \mid s \in S\}$.*

Definition 4. *The n^{th} triangular number, T_n, is given by $\sum_{i=1}^{n} i = \binom{n+1}{2}$.*

2.3 Basic k-Sum Algorithms

We review standard algorithms for k-Sum on n numbers where $k \in [2, 4]$.

Theorem 4. 2-Sum *on n numbers can be solved in $O(n \log n)$ time and $\tilde{O}(n)$ space.*

The key idea is to sort one set in nondecreasing order and the other in nonincreasing order. Starting at the beginning of each set, advance the element in the first set if the current sum is too small and advance the element in the second set if the current sum is too large.

Theorem 5. 3-Sum *on n numbers can be solved in $O(n^2)$ time and $\tilde{O}(n)$ space.*

3-Sum proceeds similarly, sorting the first two sets as above and then brute-force guessing the element from the third set to use.

Theorem 6 (Schroeppel Shamir '79). 4-Sum *on n numbers can be solved in $O(n^2 \log n)$ time and $\tilde{O}(n)$ space.*

4-Sum is solved by reducing to the 2-Sum case, treating $S_1 + S_2$ as one set and $S_3 + S_4$ as another. In order to avoid $\tilde{O}(n^2)$ space usage, Schroeppel and Shamir [12] use a priority queue for each set sum, each holding at most a linear number of elements.

2.4 Hash Functions

Here are some definitions concerning hash functions:

Definition 5. *A family of hash functions $H = \{h : U \to [m]\}$ is said to be universal if for every $x, y \in U$, if $x \neq y$ then $Pr_{h\in H}[h(x) = h(y)] \leq \frac{1}{m}$.*

Definition 6. *Given a family of hash functions $H = \{h : U \to [m]\}$, and a set $S \subseteq U$, let the bucket of h with value v be $h^{-1}(\{v\}) \cap S$ (i.e. all elements in S with hash value v). Also, define $\mathcal{B}_h(x) := h^{-1}(\{h(x)\}) \cap S$ (the bucket of h with value $h(x)$).*

We will be hashing the elements in our k-Sum instance. We note that we can assume elements (and hence $|U|$) are at most $\tilde{O}(n^k)$, since we can take all numbers modulo a random prime on the order of $\tilde{O}(n^k)$; there are only $O(n^k)$ sums to consider, each with at most $O(\log n)$ prime factors (the prime number theorem guarantees there are enough primes to choose from). Note that we can verify solutions to guard against collisions, so our algorithms are Las Vegas randomized.

3 Almost Affine Hashing

Hashing has long been useful in k-Sum algorithms and reductions. Baran, Demaine, and Pătraşcu [2] proved a key lemma about the load balancing property of universal families of hash functions. Baran et al. use this to show an upper-bound for 3-Sum, Pătraşcu [10] uses it to reduce 3-Sum to 3-SumConvolution, and Abboud and Lewi show a more general reduction from k-Sum to k-SumConvolution for $k \geq 2$.

Lemma 1 (Baran Demaine Pătraşcu '05). *Given any universal family of hash functions $H = \{h : [u] \to [m]\}$, some set $S \subset [u]$ of size n, and an integer $t > 2n/m - 2$, the expected number of elements $x \in S$ with $|\mathcal{B}_h(x)| \geq t$ is at most $\frac{2n}{t-2n/m+2}$.*

However, Jafargholi and Viola [8] recently pointed out that it appears that the family of hash functions used by Baran et al. with this lemma is not known to be universal. They do suggest that similar hash functions studied by Dietzfelbinger [4] might work, but do not explore the issue further. We show that this is indeed the case; we can use the following family of hash functions:

Definition 7. *Let u, m, and k positive integers be given. For $a, b \in [km]$, let the hash function $h_{a,b} : [u] \to [m]$ be defined as $h_{a,b}(x) = ((ax + b) \mod km) \operatorname{div} k$, where div is integer division.*

Let the family of hash functions $H^{lin}_{u,m,k}$ be defined as $\{h_{a,b} \mid a, b \in [km]\}$.

Theorem 7 (Dietzfelbinger '96). *If m, u, and k are all powers of 2, and $k \geq u/2$, then $H^{lin}_{u,m,k}$ is universal. In fact, it is two-wise independent, i.e. $Pr_{h \in H^{lin}_{u,m,k}}[h(x_1) = i_1 \wedge h(x_2) = i_2] = 1/m^2$ for arbitrary $i_1, i_2 \in [m]$ and distinct $x_1, x_2 \in [u]$.*

This family of hash functions is particularly interesting with the constraint that all sizes are powers of two, since it can be implemented with bit shift operations, does not require a large prime, and uses relatively few operations, all of which were noted by Dietzfelbinger [4].

With this bound on k in mind, denote $H^{lin}_{u,m,\lceil u/2 \rceil}$ as $H^{lin}_{u,m}$.

Baran, Demaine, and Pătraşcu [2] also relied the fact that the hash function they chose was "almost-linear". We prove a similar property for $H^{lin}_{u,m}$. Call a family of hash functions H that map from $[u]$ to $[m]$ *almost-affine* if for all $h \in H$ and $x, y \in [u]$, $h(x+y) \in \{h(x) + h(y) - h(0) + z \pmod{m} \mid z \in \{-1, 0, 1\}\}$.

Lemma 2. *The family of hash functions $H^{lin}_{u,m}$ is almost-affine.*

Proof. The main idea is that dividing by k before addition can only influence the result by at most 1 due to losing a carry. Suppose we have some integers a, b. Then we can write a as $ka_1 + a_2$ and b as $kb_1 + b_2$, where $a_2, b_2 \in [k]$. Notice that:

$$\begin{aligned}(a \operatorname{div} k) + (b \operatorname{div} k) &= a_1 + b_1 \\ &= \begin{cases} ((ka_1 + kb_1 + a_2 + b_2) \operatorname{div} k) & \text{if } a_2 + b_2 < k \\ ((ka_1 + kb_1 + a_2 + b_2) \operatorname{div} k) - 1 & \text{if } a_2 + b_2 \geq k \end{cases} \\ &\in \{((a+b) \operatorname{div} k) + z \mid z \in \{-1, 0\}\}.\end{aligned}$$

Hence, we can observe that:

$$\begin{aligned} h(x+y) + h(0) \pmod{m} &= (((ax + ay + b) \mod km) \operatorname{div} k) \\ &\quad + ((b \mod km) \operatorname{div} k) \pmod{m} \\ &\in \{(((ax + ay + 2b) \mod km) \operatorname{div} k) \\ &\quad + z \pmod{m} \mid z \in \{-1, 0\}\} \\ h(x) + h(y) \pmod{m} &= (((ax + b) \mod km) \operatorname{div} k) \\ &\quad + ((ay + b \mod km) \operatorname{div} k) \pmod{m} \\ &\in \{(((ax + ay + 2b) \mod km) \operatorname{div} k) \\ &\quad + z \pmod{m} \mid z \in \{-1, 0\}\}. \end{aligned}$$

Hence, $h(x+y) \in \{h(x) + h(y) - h(0)\} + \{-1, 0, 1\} \pmod{m}$, as desired. □

Lemma 2 guarantees that if $(k-1)$ sets have their hash buckets fixed, any solution that uses elements from those buckets could only have its last element in one of $2k - 1$ buckets of the last set. Hence, hashing can be used to shrink the problem size with some limited growth in the number of cases. It is worth noting that this hash works best on 3-Sum, since for larger values of k, applying the hash tends to increase the running time of the algorithm.

It can be seen that for large enough m, large buckets can be completely avoided by simply inspecting a constant number of hashes (in expectation).

Corollary 2. *Consider a universal family of hash functions $H = \{h : [u] \to [m]\}$, a set $S \subset [u]$ of size n, where $m \leq \sqrt{n}$, and an arbitrary constant $c \geq 1$. Then:*

$$Pr_{h \in H}\left[\forall x \in S : |\mathcal{B}_h(x)| \leq (c+2)\frac{n}{m}\right] \geq 1 - \frac{2}{c^2}$$

Proof. Let $t = (c+2)\frac{n}{m}$. Let $b(h)$ be the number of elements $x \in S$ with $|\mathcal{B}_h(x)| \geq t$. Applying Lemma 1 yields that $E[b(h)] \leq \frac{2n}{c(n/m)+2} \leq \frac{2m}{c}$. Applying a Markov bound yields $Pr_h[b(h) \geq cm] \leq \frac{2}{c^2}$. However, if $b(h) < cm$ then in fact $b(h) = 0$, since $b(h)$ counts the number of elements in buckets of h with at least $(c+2)\frac{n}{m}$ elements ($m \leq \sqrt{n}$ implies $\frac{n}{m} \geq m$). Hence,

$$Pr_h\left[\forall x \in S : |\mathcal{B}_h(x)| \leq (c+2)\frac{n}{m}\right] \geq 1 - \frac{2}{c^2}.$$

This completes the proof. □

The next lemma is analogous to a result proved by Baran, Demaine, and Pătraşcu [2] for 3-Sum and their hash family, but it holds for general k and our almost-affine family. It will be used to limit the number of false positives after hashing.

Lemma 3. *Given a constant k and integers $a_1, a_2, \dots, a_k$ and $b_1, b_2, \dots, b_k$ where $\sum_{i=1}^k a_i \neq \sum_{i=1}^k b_i$, the probability that $\sum_{i=1}^k h(a_i) = \sum_{i=1}^k h(b_i)$ after picking a random $h \in H_{u,m}^{lin}$ is upper-bounded by $\frac{O(1)}{m}$.*

Proof. By repeated application of Lemma 2, for all $h \in H_{u,m}^{lin}$:

$$h(\sum_{i=1}^k a_i) \in \left\{\sum_{i=1}^k h(a_i) - (k-1)h(0) + z \pmod m \mid z \in \{-k+1, \dots, k-1\}\right\}$$
$$h(\sum_{i=1}^k b_i) \in \left\{\sum_{i=1}^k h(b_i) - (k-1)h(0) + z \pmod m \mid z \in \{-k+1, \dots, k-1\}\right\}$$

Suppose that $\sum_{i=1}^k h(a_i) = \sum_{i=1}^k h(b_i)$. This could only occur if $h\left(\sum_{i=1}^k a_i\right)$ and $h\left(\sum_{i=1}^k b_i\right)$ are within $2k - 2$ of each other.

However, $H^{lin}_{u,m}$ is two-wise independent by Theorem 7. There are only $m(4k-3)$ ways to assign the values of $h\left(\sum_{i=1}^{k} a_i\right)$ and $h\left(\sum_{i=1}^{k} b_i\right)$ because they are within $2k-2$ of each other. This leads to an upper bound on the probability of $\sum_{i=1}^{k} h(a_i) = \sum_{i=1}^{k} h(b_i)$ of $\frac{4k-3}{m}$. But k is constant, completing the proof. □

4 A *k*-Sum Self-reduction

This section uses the hashing results to derive space-efficient randomized algorithms for k-Sum. Specifically, we demonstrate how to reduce the space usage of k-Sum algorithms using Corollary 2. These reductions are Las Vegas randomized. It is worth noting that without Corollary 2, the same running times could be attained, but via Monte Carlo algorithms and the universality of $H^{lin}_{u,m}$.

We begin by illustrating the idea with the 3-Sum problem, and then present a theorem for general k. This family of hash functions does particularly well when applied to 3-Sum, since it does not increase the running-time cost.

Reminder of Theorem 1. *The* 3-Sum *problem on n numbers can be solved by a Las Vegas algorithm in time $O(n^2)$ and space $\tilde{O}(\sqrt{n})$.*

Proof. Algorithm 1 is the desired algorithm.

Algorithm 1. SpaceEfficient3Sum(S_1, S_2, S_3, t)

```
1: Set $m \leftarrow \sqrt{n}$.
2: Randomly choose a hash function $H^{lin}_{u,m}$.
3: for $v \in [m]$ and $i \in \{1,2,3\}$ do
4:    Count the $a_i \in S_i$ where $h(a_i) = v$, and store the result in $c$.
5:    if $c > 5\sqrt{n}$ then
6:       Restart the algorithm.
7:    end if
8: end for
9: for $sum \in \{h(t) - 2h(0) + z \pmod{m} \mid z \in \{-2,\dots,2\}\}$ do
10:    for $v_1 \in [m]$ do
11:       for $v_2 \in [m]$ do
12:          Set $v_3 \leftarrow sum - v_1 - v_2 \pmod{m}$.
13:          Let $S'_i = \{a_i \in S_i \mid h(a_i) = v_i\}$ for $i = 1,2,3$.
14:          Run the basic 3-Sum algorithm on $S'_1, S'_2, S'_3, t$. Return any found solution.
15:       end for
16:    end for
17: end for
18: Report no solution.
```

Correctness: Since $H^{lin}_{u,m}$ is almost-affine (Lemma 2), for any solution (a_1,a_2,a_3):

$$h(a_1) + h(a_2) + h(a_3) \in \{h(t) - 2h(0) + z \pmod{m} \mid z \in \{-2,\dots,2\}\}.$$

Hence at some point $v_i = h(a_i)$ for $i = 1, 2, 3$ and the algorithm will find this solution. When no solutions exist, the algorithm cannot find one.

Running Time: Since $m = \sqrt{n}$, applying Corollary 2 with $c = 3$ yields that there is at least a $\frac{7}{9}$ chance that all buckets for a specific set S_i are at most $5\sqrt{n}$ elements in any bucket. By a union bound, there is at least a $\frac{1}{3}$ chance that all sets S_i have at most $5\sqrt{n}$ elements in any bucket. Hence in expectation the algorithm picks at most three hashes before it gets past the bucket size check.

Checking bucket sizes takes $O(n^{1.5})$ time, since a linear scan is done for each of $\sqrt{n}$ values of v. There are $O(n)$ choices for v_1, v_2, v_3, and each iteration runs the basic 3-Sum algorithm on instances of size $O(\sqrt{n})$, taking $O(n)$ time. The total running time is hence $O(n^2)$. Note that Baran, Demaine, Pătraşcu [2] could be used for subproblems to shave off additional log factors.

Memory Usage: Since bucket sizes are guaranteed not to be too large, storing S'_i only requires $\tilde{O}(\sqrt{n})$ space. Running the basic 3-Sum algorithm on instances of $O(\sqrt{n})$ elements also uses $\tilde{O}(\sqrt{n})$ space.

This completes the proof. □

This technique holds for general k as well as sizes other than $\tilde{O}(\sqrt{n})$. Theorem 1 is actually an application of the following general theorem:

Reminder of Theorem 2. *Let A be a Las Vegas algorithm that solves k-Sum ($k \geq 3$) on n numbers in $T(n)$ time and $S(n)$ space where $T(n), S(n) \in poly(n)$, and let $\delta \in [0, 1]$ be an arbitrary constant. Then there is a Las Vegas algorithm A' that solves k-Sum on n numbers in $O(n^{k-\delta(k-1)} + n^{k-\delta(k-1)-1}T(n^\delta))$ time and $O(n^\delta + S(n^\delta))$ space.*

The proof of this theorem is more complex and nuanced, and can be found in Appendix A. Notice when $\delta = 0$, we recover the brute force algorithm's running time and space usage. When $\delta = 1$, we recover the input algorithm's (given that the input algorithm uses at least linear space).

As mentioned previously, there is a naive self-reduction that shrinks space usage. It splits each set up into buckets of size $O(n^\delta)$, and runs another algorithm on each possible combination of buckets. This would result in an algorithm that runs in $\tilde{O}(n^{k-\delta k}T(n^\delta))$ time and $\tilde{O}(S(n^\delta))$ space. This naive reduction also recovers brute-force for $\delta = 0$ and the input algorithm for $\delta = 1$, and in fact interpolates the exponents of the two algorithms for values of δ in between. Our reduction via hashing beats this naive reduction for all $\delta \in [0, 1]$, with equality only at the endpoints (given that the input algorithm uses at least linear space).

5 Time-Space Tradeoffs for *k*-Sum

This section explores the results we get by applying Theorem 2. The following theorem provides a family of k-Sum algorithms to use on subproblems:

Reminder of Theorem 3. *There exists a Monte Carlo algorithm that solves k-Sum on n numbers in $\tilde{O}(n^{k-f(k)-1})$ time and $\tilde{O}(n)$ space, where $f(k)$ is the largest integer p such that $T_{p+1} + 1 \leq k$.*

The proof of Theorem 3 is in Appendix B. This theorem yields:

Reminder of Corollary 1. *Let $\delta \in [0,1]$ be an arbitary constant. There is a Monte Carlo algorithm that solves k-*Sum *on n numbers in $\tilde{O}(n^{k-\delta(k-1)} + n^{k-1-\delta f(k)})$ time and $\tilde{O}(n^{\delta})$ space, where $f(k)$ is the largest integer p such that the $(p+1)$-th triangular number is at most one less than k.*

Proof. This follows directly from applying the reduction from Theorem 2 to the algorithm guaranteed by Theorem 3. □

Corollary 1 gives a tradeoff curve that consists of two linear pieces. Suppose we want a k-Sum algorithm that runs in $\tilde{O}(n^{\delta})$ space. For the region $\delta \in [0, \frac{1}{(k-1)-f(k)}]$, we have an algorithm that runs in $\tilde{O}(n^{k-\delta(k-1)})$ time. In the region $\delta \in [\frac{1}{(k-1)-f(k)}, 1]$, we have one that runs in $\tilde{O}(n^{k-1-\delta(f(k))})$ time. At the shared point in these two intervals, the running time is $\tilde{O}(n^{k-(k-1)/(k-1-f(k))})$.

We also note that $f(k)$ has the following (coarse) lower bound:

Lemma 4. $f(k) \geq \sqrt{2k} - 2$

Proof. Notice that $T_{\lceil\sqrt{2k}-1\rceil} \leq \frac{\sqrt{2k}(\sqrt{2k}-1)}{2} \leq k$. But $f(k)$ is the largest integer p for which $T_{p+1} + 1 \leq k$, so it must be at least $\lceil\sqrt{2k}-2\rceil$. □

This immediately gives upper-bounds for Theorem 3 and Corollary 1:

Corollary 3. *There exists a Monte Carlo algorithm that solves k-*Sum *on n numbers in $\tilde{O}(n^{k-\sqrt{2k}+1})$ time and $O(n)$ space.*

Corollary 4. *Let $\delta \in [0,1]$ be an arbitary constant. There is a Monte Carlo algorithm that solves k-*Sum *on n numbers in $\tilde{O}(n^{k-\delta(k-1)} + n^{k-1-\delta f(k)})$ time and $\tilde{O}(n^{\delta})$ space, where $f(k)$ is the largest integer p such that the $(p+1)$-th triangular number is at most one less than k.*

6 Conclusion

Our results also extend to the k-Xor problem [8], which is identical except that the elements are vectors from $\mathbb{F}_2^n$ instead of integers. For this variant there is a simple linear universal family of hash functions (let M be a random $k \times n$ matrix over $\mathbb{F}_2$, and define $h_M(x) = Mx$). Hence the hashing properties we need easily hold in this case, and the same techniques work.

One open problem is whether our family of linear-space Monte Carlo algorithms for k-Sum can be derandomized or be made to work for real inputs. The Schroeppel-Shamir algorithm for 4-Sum matches the $\tilde{O}$ running-time, so it seems plausible that this might hold for larger values of k.

An especially interesting open problem is whether the algorithms presented by Howgrave-Graham and Joux as well as Becker, Coron, and Joux can be moved from random-instances to worst-case instances and randomized algorithms. Giving better worst-case bounds for SubsetSum has been an open problem for more than 30 years.

Acknowledgements. Many thanks to Ryan Williams for providing helpful pointers to existing literature, insightful discussions, and proofreading.

References

1. Austrin, P., Kaski, P., Koivisto, M., Määttä, J.: Space–time tradeoffs for subset sum: An improved worst case algorithm (2013)
2. Baran, I., Demaine, E.D., Pătraşcu, M.: Subquadratic algorithms for 3SUM. In: Dehne, F., López-Ortiz, A., Sack, J.-R. (eds.) WADS 2005. LNCS, vol. 3608, pp. 409–421. Springer, Heidelberg (2005)
3. Becker, A., Coron, J.-S., Joux, A.: Improved Generic Algorithms for Hard Knapsacks. In: Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 364–385. Springer, Heidelberg (2011)
4. Dietzfelbinger, M.: Universal hashing and k-wise independent random variables via integer arithmetic without primes. In: Puech, C., Reischuk, R. (eds.) STACS 1996. LNCS, vol. 1046, pp. 569–580. Springer, Heidelberg (1996)
5. Dinur, I., Dunkelman, O., Keller, N., Shamir, A.: Efficient Dissection of Composite Problems, with Applications to Cryptanalysis, Knapsacks, and Combinatorial Search Problems. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 719–740. Springer, Heidelberg (2012)
6. Gajentaan, A., Overmars, M.H.: On a class of $O(n^2)$ problems in computational geometry. Comput. Geom. Theory Appl. 45(4), 140–152 (2012)
7. Howgrave-Graham, N., Joux, A.: New Generic Algorithms for Hard Knapsacks. In: Gilbert, H. (ed.) EUROCRYPT 2010. LNCS, vol. 6110, pp. 235–256. Springer, Heidelberg (2010)
8. Jafargholi, Z., Viola, E.: 3sum, 3xor, triangles. CoRR, abs/1305.3827 (2013)
9. Jørgensen, A.G., Pettie, S.: Threesomes, degenerates, and love triangles. CoRR, abs/1404.0799 (2014)
10. Patrascu, M.: Towards polynomial lower bounds for dynamic problems. In: STOC, pp. 603–610 (2010)
11. Pătraşcu, M., Williams, R.: On the possibility of faster sat algorithms. In: Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, pp. 1065–1075. Society for Industrial and Applied Mathematics, Philadelphia (2010)
12. Schroeppel, R., Shamir, A.: A $T \cdot S^2 = O(2^n)$ Time/Space Tradeoff for Certain NP-Complete Problems. In: Proceedings of the 20th Annual Symposium on Foundations of Computer Science, SFCS 1979, Washington, DC, USA, pp. 328–336. IEEE Computer Society Press, Los Alamitos (1979), `http://dx.doi.org/10.1109/SFCS.1979.3`, doi:10.1109/SFCS.1979.3
13. Woeginger, G.J.: Space and time complexity of exact algorithms: Some open problems. In: Downey, R.G., Fellows, M.R., Dehne, F. (eds.) IWPEC 2004. LNCS, vol. 3162, pp. 281–290. Springer, Heidelberg (2004)

A Appendix: Proof of Theorem 2

Reminder of Theorem 2. *Let A be a Las Vegas algorithm that solves k-Sum ($k \geq 3$) on n numbers in $T(n)$ time and $S(n)$ space where $T(n), S(n) \in poly(n)$,* *and let $\delta \in [0,1]$ be an arbitrary constant. Then there is a Las Vegas algorithm A' that solves k-*SUM *on n numbers in $O(n^{k-\delta(k-1)} + n^{k-\delta(k-1)-1}T(n^\delta))$ time and $O(n^\delta + S(n^\delta))$ space.*

Proof. The key idea is to use hashing to reduce the size of each set by a square root factor at each step. However, storing any of the intermediate sets of this computation defeats the purpose of hashing any further. To avoid this, the algorithm first determines all hash functions and values to shrink each set to the desired size, and then computes the final sets in one step.

A' will recursively construct a list L whose elements are of the form $(h, v_1, v_2, \dots, v_k)$, i.e. a hash function followed by k hash values (one for each S_i). At any step, define the active set of S_i to be $\tilde{S}_i = \{s \in S_i \mid h(s) = v_i \forall (h, v_1, v_2, \dots, v_k) \in L\}$. Each element appended to L reduces the size of all active sets, so elements can be repeatedly appended until the active sets are only $O(n^\delta)$ in size, at which point it is safe to invoke A. To handle the possibility that δ is not a perfect power of $\frac{1}{2}$, define the function $s(x) := \max((\frac{1}{2})^x, \delta)$. Step i of the algorithm will reduce the size of all active sets from $O(n^{s(i)})$ to $O(n^{s(i+1)})$.

The recursive helper function R will construct L and then invoke A. It has access to all sets S_i and is given a partially constructed L. Algorithm A' simply calls R with $L = \emptyset$.

Algorithm 2. $R(L, S_1, \dots, S_k)$

```
Require: The active sets $\tilde{S}_1, \dots, \tilde{S}_k$ each contain at most $(k+2)^2 n^{s(\ell)}$ elements.
 1: Let $\ell \leftarrow |L|$.
 2: if $s(\ell) = \delta$ then
 3:     Call $A(\tilde{S}_1, \dots, \tilde{S}_k)$.
 4:     return
 5: end if
 6: Set $m_\ell \leftarrow (k+2)n^{s(\ell)-s(\ell+1)}$.
 7: Pick a random hash function $h \in H^{lin}_{u,m_\ell}$.
 8: for $v \in [m_\ell]$ and $i \in \{1, \dots, k\}$ do
 9:     Count the $a_i \in \tilde{S}_i$ where $h(a_i) = v$, and store the result in $c$.
10:     if $c > (k+2)^2 n^{s(\ell+1)}$ then
11:         Pick another hash and try again.
12:     end if
13: end for
14: for $sum \in \{h(t) - (k-1)h(0) + z \pmod{k_\ell} \mid z \in \{-k+1, \dots, k-1\}\}$ do
15:     for $v_1, \dots, v_{k-1} \in [m_\ell]$ do
16:         Set $v_k \leftarrow sum - \sum_{i=1}^{k-1} v_i \pmod{m_\ell}$.
17:         Let $L'$ be $L$ appended with $(h, v_1, \dots, v_k)$.
18:         Call $R(L', S_1, \dots, S_k)$.
19:     end for
20: end for
```

Correctness: We first prove the size guarantee made when calling R. A' initially calls R with $\ell = 0$ and sets of size $n \leq (k+2)^2 n^{s(0)}$. R ensures that the hash it has chosen creates buckets that are no larger than $(k+2)^2 n^{s(\ell+1)}$ in size, so it may safely append an additional element to L before recursing.

We also want to show that if a solution exists, the algorithm will find it. Since $H^{lin}_{u,m}$ is almost affine, a call to R where each element of the solution is active will in turn make some recursive call where the solution elements are still active. Since the top-level call to R is made with all elements active, all elements of a solution will be found by the algorithm.

Running Time: Checking that the buckets of a randomly-selected hash function are not too large takes $O(n^{1+s(\ell)-s(\ell+1)})$ time since the algorithm needs to perform a linear scan for each hash value $v \in [m_\ell]$. Applying Corollary 2 with $c = k$, the chance of a hash failing over a specific S_i is at most $\frac{2}{k^2}$; the chance of it failing over any S_i, by a union bound, is at most $\frac{2}{k}$. Since $k \geq 3$, the expected number of hashes the algorithm needs to pick and check is at most three. Hence our expected time checking for hashes during a single call to R, not including recursive subcalls, is $O(n^{1+s(\ell)-s(\ell+1)})$.

There is a single call where $\ell = 0$. Each recursive level of R makes $O(n^{(k-1)(s(\ell)-s(\ell+1))})$ calls to the level below it. Hence, there are $O(n^{(k-1)(1-s(\ell))})$ calls to R for a fixed ℓ. The total expected time checking for hashes during all calls with a given ℓ is therefore $O(n^{(k-1)(1-s(\ell))+(1+s(\ell)-s(\ell+1))})$. But notice that:

$$\begin{aligned}(k-1)(1-s(\ell)) + (1+s(\ell)-s(\ell+1)) &= k - s(\ell)(k-2) - s(\ell+1)\\ &\leq k - s(\ell+1)(k-1)\\ &\leq k - \delta(k-1).\end{aligned}$$

Hence, the total expected time checking for hashes during all calls with a given ℓ is also $O(n^{k-\delta(k-1)})$. Since the algorithm only searches for hash functions for at levels up to $\lceil \log_2 \frac{1}{\delta} \rceil - 1$, the total expected running time checking for hashes overall is $O(n^{k-\delta(k-1)})$.

When $s(\ell) = \delta$, the algorithm needs to compute all $\tilde{S}_i$. From the previously-derived formula, there are only $O(n^{(k-1)(1-\delta)})$ calls where this occurs. Computing all $\tilde{S}_i$ only requires a linear scan of each S_i, so this takes time $O(n^{k-\delta(k-1)})$.

Finally, the algorithm invokes A $O(n^{(k-1)(1-\delta)})$ times on sets of size at most $(k+2)^2 n^\delta$, so in total the algorithm uses $O(n^{(k-1)(1-\delta)} T(n^\delta))$ time making calls to A.

The total time taken is $O(n^{k-\delta(k-1)} + n^{k-\delta(k-1)-1} T(n^\delta))$.

Memory Usage: Notice that L contains at most $\lceil \log_2 \frac{1}{\delta} \rceil$ elements of size $(k+1)$ each, so it takes $O(1)$ space. The space needed to verify there are no large buckets is also $O(1)$, since the algorithm only computes a count for only a single hash value at a time.

Invoking A on sets of size at most $(k+2)^2 n^\delta$ requires only $O(n^\delta + S(n^\delta))$ space (to store the inputs along with the space needed by A).

This completes the proof. □

B Appendix: Linear Space Algorithms for k-Sum

The two factors in the space usage of the algorithm derived from applying Theorem 2 are balanced when the original algorithm requires only linear space. In this appendix, we utilize almost affine hashing to produce a family of linear-space algorithms for k-Sum.

Theorem 8, Theorem 9, and Corollary 5 are based on results produced by Austrin, Kaski, Koivisto, and Määttä [1]. We shift from the SubsetSum problem to the k-Sum problem and use almost affine hashing in place of carefully chosen moduli. The switch from chosen moduli to this family of hash functions is justified by the fact that, as mentioned before, this hashing can be done with bit shift operations, without the availability of large primes, and with relatively few operations. As mentioned before, if the Schroeppel-Shamir reduction from SubsetSum to k-Sum is used on this set of algorithms, every endpoint of their piecewise-linear time-space tradeoff curve for SubsetSum is recovered, so this adaptation is lossless.

The technique requires that there are only a constant number of solutions to the k-Sum instance, which can be ensured by some standard preprocessing:

Theorem 8. *There is a $O(n \log n)$ time Monte Carlo algorithm to process instances of k-Sum which takes as input an instance $(S_1, \dots, S_k, t)$ of size n and outputs $O(\log n)$ k-Sum instances of the same size. If the original instance has a solution, then at least one of the output instances will have at least one solution and at most $O(1)$ solutions. Otherwise, none of the instances will have any solutions.*

Proof. Consider a k-Sum instance $S_1, \dots, S_k$ with target t. Without loss of generality, all sets contain only nonnegative elements (it is safe to add a positive constant to all elements in any particular set S_i and to the target t at the same time).

Let S be the set of all solutions. The algorithm will guess that the size of S is in the range $[2^s, 2^{s+1})$ for $s = 0, 1, \dots, k \log n$ (try them all, one will be correct). Let $m = 2^s$, and for each S_i choose uniformly at random a function $f_i : S_i \to [m]$. Also, randomly choose a $u \in [m]$.

For a fixed solution $(a_1, \dots, a_k) \in S$, there is a $\frac{1}{m}$ probability that:

$$\sum_{i=1}^{k} f_i(a_i) \equiv u \pmod{m} \tag{1}$$

Also, any two distinct solutions both satisfy (1) with probability $\frac{1}{m^2}$. When s is a correct guess,
$1 \leq \frac{|S|}{m} < 2$. Let X be a random variable denoting the number of solutions that satisfy (1). Then:

$$\begin{aligned}\mathbb{E}[X] &= \frac{|S|}{m} \\ \mathbb{E}[X^2] &= \mathbb{E}[X] + \frac{|S|(|S|-1)}{m^2} < \frac{|S|}{m} + \frac{|S|^2}{m^2}.\end{aligned}$$

The first and second moment methods give:

$$\begin{aligned}Pr(X > 10) &< \frac{\mathbb{E}[X]}{10} < \frac{1}{5} \\ Pr(X > 0) &> \frac{\mathbb{E}[X]^2}{\mathbb{E}[X^2]} > \frac{1}{1 + m/|S|} > \frac{1}{2}.\end{aligned}$$

By a union bound, there is at least one correct solution and at most $O(1)$ solutions that satisfy (1) with constant probability. If a solution satisfies (1), then in fact $\sum_{i=1}^{k} f_i(a_i) = u + jm$ for some $j \in [k]$ (there are at most $k-1$ carries). Guess this j by iterating over all possibilities.

Let A be the largest element in any S_i. For all S_i, let
$S_i' = \{a_i + (kA+1)f_i(a_i) \mid a_i \in S_i\}$ and let $t' = t + (kA+1)(u+jm)$. Notice that this maps invalid solutions to invalid solutions and correct solutions that satisfy (1) to correct solutions provided that j was guessed correctly. The k-Sum instances $S_1', \ldots, S_k'$ with target t' are output, over all choices of s and j, for $k \log n = O(\log n)$ total instances.

If the original instance has a solution, then at least one guess of s is correct, and there is a constant probabability that there will be an output instance has at least one solution and at most $O(1)$ solutions. Otherwise, none of the instances will have solutions, as desired.

The algorithm takes $O(n \log n)$ time since there are $\log n$ values of s to guess and modifying every element takes linear time.

This completes the proof. □

We now inductively construct algorithms to solve k-Sum for increasing k, assuming that at least one and at most $O(1)$ solutions exist. For this proof, k will take on values one more than a triangular number.

Theorem 9. *For every integer $p \geq 0$, there exists a Monte Carlo algorithm that solves $(T_{p+1}+1)$-Sum on n numbers in $\tilde{O}(n^{T_p+1})$ time and $\tilde{O}(n)$ space, assuming that at least one solution and at most $O(1)$ solutions exist.*

Proof. It will be convenient to define a recursive function HashReduction that takes k sets $S_1, \dots, S_k$ and a modulus m and finds up to *num* solutions (stopping with fewer if not that many solutions exist) to k-Sum in the modular setting.

We wish to show that Algorithm 3 meets the desired requirements when run with $k = T_{p+1} + 1$, $num = 1$, and m large enough to avoid wrap-around (begin with arithmetic over the integers).

Note that HashReduction makes calls to other functions, passing the images of S_i under some hash function h. It is assumed that these sets are implemented as vectors, and that the results can be returned as indices, so that the original elements can be recovered.

Algorithm 3. HashReduction$(k, S_1, \dots, S_k, t, num, m)$

```
Require: $k = T_j + 1$ for some $j \geq 1$.
 1: if $k = 2$ then
 2:     Run the basic 2-Sum algorithm on $S_1$ and $S_2$, stopping at $num$ solutions.
 3:     return
 4: end if
 5: Let $m' \leftarrow \Theta(n^{j-1})$.
 6: Randomly choose a hash function $h \in H^{lin}_{m,m'}$.
 7: for $v_\ell \in [m']$ do
 8:     Initialize an empty lookup table $T$.
 9:     Sort $h(S_1)$.
10:     for $a_2 \in S_2, \dots, a_j \in S_j$ do
11:         Do a binary search for $v_\ell - \sum_{i=2}^{j} h(a_i) \pmod{m'}$ in $h(S_1)$.
12:         If a solution $(a_1, \dots, a_j)$ is found, store it in $T$ as $(\sum_{i=1}^{j} a_i \pmod{m}) \rightarrow$
            $(a_1, \dots, a_j)$, but store $\Theta(n)$ entries at most.
13:     end for
14:     for $sum \in \{h(t) - (k-1)h(0) + z \pmod{m'} \mid z \in \{-k+1, \dots, k-1\}\}$ do
15:         Set $v_r \leftarrow sum - v_\ell \pmod{m'}$.
16:         Call HashReduction$(T_{j-1} + 1, h(S_{j+1}), \dots, h(S_k), v_r, \Theta(n^{T_{j-2}+1}), m')$.
17:         For each solution $(a_{j+1}, \dots, a_k)$, lookup $t - \sum_{i=j+1}^{k} a_i \pmod{m}$ in $T$.
18:         for entry $t - \sum_{i=j+1}^{k} a_i \pmod{m} \rightarrow (a_1, \dots, a_j)$ in $T$ do
19:             Record $(a_1, \dots, a_j, a_{j+1}, \dots, a_k)$ as a solution.
20:             if $num$ solutions have been found then
21:                 return
22:             end if
23:         end for
24:     end for
25: end for
```

Correctness: We begin by proving that HashReduction would run correctly if each call actually returned all solutions, not just the requested *num* at each recursive call. We will later show that it suffices to only return the requested number of solutions.

HashReduction divides the sets into two groups, left and right, and guesses the sum of a solution's hash values for each group. It relies on the almost affine-property of H^{lin} in order to reduce the number of cases it needs to guess.

Suppose there is a solution to the current HashReduction call, $a_1, \dots, a_k$. If $k = 2$, then basic 2-Sum algorithm is called, which is a correct algorithm by Theorem 4. Otherwise, HashReduction chooses a hash and then runs its main for-loop.

In some iteration, v_ℓ is a correct guess for $\sum_{i=1}^{j} h(a_i)$. In this same iteration, $(\sum_{i=1}^{j} a_i) \to (a_1, \dots, a_j)$ will be stored in T. By Lemma 2, $\sum_{i=1}^{k} h(a_i) \pmod{m'}$ must equal some value in the set
$\{h(t) - (k-1)h(0) + z \pmod{m'} \mid z \in \{k-1, \dots, k-1\}\}$. The algorithm guesses all possible values for this sum, and then computes what v_r must be to get this sum. Since $\sum_{i=1}^{k} a_i = t$, this solution will be correctly recorded by the algorithm and returned.

Next, we will show that it suffices to return only the requested number of solutions. Here, we will use the assumption that at most $O(1)$ solutions exist in the top level call. Fix some solution to the top level call, $a_1, a_2, \dots, a_k$. Consider only the recursive branch where all v_ℓ and v_r are guessed correctly for this solution.

We claim that with probability arbitrarily close to 1, the value of *num* for every call along this branch is large enough to have all solutions returned. Since there are $O(1)$ function calls in this branch (this number depends only on the original value of k), it suffices to show this holds for any particular call.

At any recursive call, the current sets under consideration are a contiguous group of the original sets, $S_\ell, \dots, S_r$, transformed by randomly chosen hash functions $h_1, h_2, \dots, h_s$. We want to bound the probability of a false positive, i.e. some $b_\ell, \dots, b_r$ such that $b_i \in S_i$ and

$$\sum_{i=\ell}^{r} (h_s \circ \cdots \circ h_1)(a_i) = \sum_{i=\ell}^{r} (h_s \circ \cdots \circ h_1)(b_i).$$

By Lemma 3, this probability has an upper bound of $\frac{O(1)}{\#\text{ values of } h_s}$ plus the probability of the event:

$$\sum_{i=\ell}^{r} (h_{s-1} \circ \cdots \circ h_1)(a_i) = \sum_{i=\ell}^{r} (h_{s-1} \circ \cdots \circ h_1)(b_i).$$

Repeating this s times gives an upper bound of $\sum_{i=1}^{s} \frac{O(1)}{\#\text{ values of } h_i}$ plus the probability of the event that:

$$\sum_{i=\ell}^{r} a_i = \sum_{i=\ell}^{r} b_i.$$

But any $b_\ell, \ldots, b_r$ for which this holds can be combined with the other a_i to make a different solution to the original top-level call: $(a_1, \ldots, a_{\ell-1}, b_\ell, \ldots, b_r, a_{r+1}, \ldots, a_k)$. Hence there are only $O(1)$ many $b_\ell, \ldots, b_r$ for which this event can occur. Ignoring these $O(1)$ solutions, the probability that any other $b_\ell, \ldots, b_r$ is a false positive is just $\frac{O(1)}{\#\text{ values of } h_s}$, since the number of possible hash values drops by a factor of roughly n in each recursive call. But the number of values of h_s was chosen to be some $m' = \Theta(n^{j-1})$. The algorithm should pick m' large enough to guarantee that non-solutions only have less than a $\frac{1}{n^{j-1}}$ probability of a false positive. It is then possible to use a Markov bound to pick a large enough value for $\Theta(n)$ and $\Theta(n^{T_{j-2}+1})$ to guarantee that with probability arbitrarily close to 1, the chosen value of *num* for any particular call along this solution branch is large enough.

Hence this fixed solution will eventually be found by the top level call, and the algorithm correctly finds some solution.

Running Time: We will inductively prove that for all $p \geq 0$, HashReduction takes $\tilde{O}(n^{T_p+1} + num)$ time when run with $k = T_{p+1} + 1$.

For the base case $p = 0$, $k = 2$ and the algorithm simply runs the basic 2-Sum algorithm. Theorem 4 guarantees it has the desired running time.

Assume that the inductive hypothesis holds for $p = q$. Consider when HashReduction is run with $k = T_{q+2} + 1$, $j = q + 2$. The algorithm chooses a hash function ($O(1)$ time) and then runs through $m' = O(n^{q+1})$ iterations of its main for-loop. Finding solutions to store in T takes $O(n^{T_q+1})$ time in all cases, since for all $q \geq 0$, $q + 1 \leq T_q + 1$. By the inductive hypothesis, calling HashReduction with $k = T_{q+1} + 1$ takes $\tilde{O}(n^{T_q+1})$ time.

Count the time taken to find and return solutions separately. Since only *num* solutions are requested, this requires $O(num)$ time. The total time for all iterations is hence $\tilde{O}(n^{T_{q+1}+1} + num)$, as desired. By induction, the hypothesis is true for all $p \geq 0$.

Memory Usage: Every recursive call uses $\tilde{O}(n)$ space for the lookup table T. The number of recursive calls depends only on k, not n, so the total memory usage is $\tilde{O}(n)$, as desired. $\square$

Instances of the general k-Sum problem can be solved by preprocessing and then running this algorithm.

Corollary 5. *For a constant integer $p \geq 0$, there exists a Monte Carlo algorithm that solves* $(T_{p+1} + 1)$-Sum *on n numbers in $\tilde{O}(n^{T_p+1})$ time and $\tilde{O}(n)$ space.*

Algorithm 4. CompleteKSum($k, S_1, \ldots, S_k, t$)

```
Proof.  1: Preprocess $(S_1, \ldots, S_k, t)$ via the algorithm in Theorem 8.
 2: **for** Resulting instances $(S'_1, \ldots, S'_k, t')$ **do**
 3:     Run HashReduction on the instance to find a solution.
 4: **end for**
```

By Theorem 8 and Theorem 9, Algorithm 4 has the desired properties. Notice that it has to run on $O(\log n)$ instances, but this is absorbed by the $\tilde{O}$ notation. $\square$

The following lemma produces algorithms for the remaining values of k:

Lemma 5. *Let A be an algorithm that solves k-*Sum *($k \geq 3$) on n numbers in $\tilde{O}(n^d)$ time and $\tilde{O}(n)$ space for some constant d. Then there is an algorithm A' that solves* $(k+1)$-Sum *on n numbers in $\tilde{O}(n^{d+1})$ time and $\tilde{O}(n)$ space.*

Proof. The algorithm A' is to guess one element $s \in S_{k+1}$ of the solution and then to run A on $S_1, \ldots, S_k$ for the remaining elements, which now need to sum to $t - s$. $\square$

Hence, for general k, we get the following Monte Carlo algorithm for k-Sum:

Reminder of Theorem 3. *There exists a Monte Carlo algorithm that solves k-*Sum *on n numbers in $\tilde{O}(n^{k-f(k)-1})$ time and $\tilde{O}(n)$ space, where $f(k)$ is the largest integer p such that $T_{p+1} + 1 \leq k$.*

Proof. This follows directly from Corollary 5 and Lemma 5. $\square$

Equivalence between Priority Queues and Sorting in External Memory

Zhewei Wei[1,2] and Ke Yi[3,⋆]

[1] School of Information, Renmin University of China
wzskytop@gmail.com
[2] MADALGO[⋆⋆], Department of Computer Science, Aarhus University, Denmark
zhewei@cs.au.dk
[3] The Hong Kong University of Science and Technology, Hong Kong
yike@cse.ust.hk

Abstract. A priority queue is a fundamental data structure that maintains a dynamic ordered set of keys and supports the followig basic operations: insertion of a key, deletion of a key, and finding the smallest key. The complexity of the priority queue is closely related to that of sorting: A priority queue can be used to implement a sorting algorithm trivially. Thorup [11] proved that the converse is also true in the RAM model. In particular, he designed a priority queue that uses the sorting algorithm as a black box, such that the per-operation cost of the priority queue is asymptotically the same as the per-key cost of sorting. In this paper, we prove an analogous result in the external memory model, showing that priority queues are computationally equivalent to sorting in external memory, under some mild assumptions. The reduction provides a possibility for proving lower bounds for external sorting via showing a lower bound for priority queues.

1 Introduction

The priority queue is an abstract data structure of fundamental importance. A priority queue maintains a set of keys and supports the following operations: insertion of a key, deletion of a key, and findmin, which returns the current minimum key in the priority queue. It is well known that a priority queue can be used to implement a sorting algorithm: we simply insert all keys to be sorted into the priority queue, and then repeatedly delete the minimum key to extract the keys in sorted order. Thorup [11] showed that the converse is also true in the RAM model. In particular, he showed that given a sorting algorithm that sorts N keys in $NS(N)$ time, there is a priority queue that uses the sorting algorithm as a black box, and supports insertion and deletion in $O(S(N))$ time, and findmin in constant time. The reduction uses linear space. The main implication of this reduction is that we can regard the complexity of internal priority queues

⋆ Supported by HKRGC under grant GRF-621413.
⋆⋆ Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation.

as settled, and just focus on establishing the complexity of sorting. Algorithmically, it also gives new priority queue constructions by using the fastest (integer) sorting algorithms currently known: an $O(N \log\log N)$ deterministic algorithm by Han [7] and an $O(N\sqrt{\log\log N})$ randomized one by Han and Thorup [8].

In this paper, we prove an analogous result in the external memory model (the I/O model), showing that priority queues are almost computationally equivalent to sorting in external memory. We design a priority queue that uses the sorting algorithm as a black box, such that the cost of an insertion or deletion (given the key of the element to be deleted) in the priority queue is essentially the same as the per-key I/O cost of the sorting algorithm. The priority queue always has the current minimum key in memory so findmin can be handled without I/O cost. Our priority queue is a non-trivial generalization of Thorup's.

1.1 Our Results

Let us first recall the standard I/O model [1]: The machine consists of an internal memory of size M and an infinitely large external memory. Computation can only be carried out in internal memory. The external memory is divided into blocks of size B, and with one I/O, a block of B keys can be together moved from internal to external memory or vice versa. We measure the complexity of an algorithm by counting the number of I/Os it performs, while internal memory computation is free.

Let $\log^{()}$ denote the nested logarithmic function, i.e., $\log^{(0)} x = x$ and $\log^{(i)} = \log(\log^{(i-1)} x)$. Our main result is stated in the following theorem:

Theorem 1. *Suppose we can sort up to N keys in $NS(N)/B$ I/Os in external memory, where S is a non-decreasing function. Then there exists an external priority queue that uses linear space and supports a sequence of N insertion and deletion operations in $O(\frac{1}{B}\sum_{i\geq 0} S(B\log^{(i)}\frac{N}{B}))$ amortized I/Os per operation. Findmin can be supported without I/O cost. The reduction uses $O(B)$ internal memory and is deterministic.*

The first implication of Theorem 1 is that if the main memory has size $\Omega(B\log^{(c)}\frac{N}{B})$ for any constant c, then our priority queue supports insertion and deletion with $O(S(N)/B)$ amortized I/O cost. This is because $S(N) = 0$ when $N \leq M$. Even if $M = O(B)$, the reduction is still tight as long as the function S grows not too slowly. More precisely, we have the following corollary:

Corollary 1. *For $S(N) = \Omega(2^{\log^* \frac{N}{B}})$, the priority queue supports updates with $O(S(N)/B)$ amortized I/O cost; for $S(N) = o(2^{\log^* \frac{N}{B}})$, the priority queue supports updates with $O(S(N)\log^* \frac{N}{B}/B)$ amortized I/O cost.*

The first part can be verified by plugging $S(N) = 2^{\log^* \frac{N}{B}}$ into Theorem 1 and showing that the $S(B\log^{(i)}\frac{N}{B})$'s decrease exponentially with i. For the second part, we simply relax all the $S(B\log^{(i)}\frac{N}{B})$'s to $S(N)$. Note that $2^{\log^* \frac{N}{B}} = o(\log^{(c)}\frac{N}{B})$ for any constant c, so it is very unlikely that a sorting algorithm could achieve $S(N) = o(2^{\log^* \frac{N}{B}})$. No such algorithm is known, even in the RAM model. Therefore, we can essentially consider our reduction to be tight.

1.2 Related Work

Sorting and priority queues have been well studied in the comparison-based I/O model, in which the keys can only be accessed via comparisons. Aggarwal and Vitter [1] showed that $\Theta(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os are sufficient and necessary to sort N keys in the comparison-based I/O model. This bound is often referred to as the *sorting bound*. If the comparison constraint is replaced by the weaker indivisibility constraint, there is an $\Omega(\min\{\frac{N}{B} \log_{M/B} \frac{N}{B}, N\})$ lower bound, known as the *permuting bound*. The two bounds are the same when $\frac{N}{B} \log_{M/B} \frac{N}{B} < N$; it is conjectured that for this parameter range, $\Omega(\frac{N}{B} \log_{M/B} \frac{N}{B})$ is still the sorting lower bound even without the indivisibility constraint. For $\frac{N}{B} \log_{M/B} \frac{N}{B} > N$, the current situation in the I/O model is the same as that in the RAM model, that is, the best upper bound is just to use the best RAM algorithm (which has $O(N \log\log N)$ time deterministically or $O(N\sqrt{\log\log N})$ time randomized) naively in external memory with one I/O only accessing one key in external memory, completely wasting the parallelism of the block accesses. There is no lower bound without the indivisibility assumption or when $\frac{N}{B} \log_{M/B} \frac{N}{B} > N$. It has been observed that when the block size is not too small, none of the RAM sorting algorithms works better than the comparison-based one, which makes the situation "cleaner". Thus, a sorting lower bound (without any restrictions) has been considered to be more hopeful in the I/O model (with B not too small) than in the RAM model, and it was posed as a major open problem in [1]. Thus, our result provides a way to approach a sorting lower bound via that of priority queues, while data structure lower bounds have been considered (relatively) easier to obtain than (concrete) algorithm lower bounds (except in restricted computation models), as witnessed by the many recent strong cell probe lower bounds for data structures, such as [10,9] among many others. However, our result does not offer any new bounds for priority queues because we do not know of a better sorting algorithm than the comparison-based ones in the I/O model.

Since a priority queue can be used to sort N keys with N insertion and N deletemin operations, it follows that $\Omega(\frac{1}{B} \log_{M/B} \frac{N}{B})$ is also a lower bound for the amortized I/O cost per operation for any external priority queue, in the comparison-based I/O model. There are many priority queue constructions that achieve this lower bound, such as the buffer tree [2], M/B-ary heaps [5], and array heaps [4]. See the survey [12] for more details. However, they do not use sorting as just a black box, and cannot be improved even if we have a faster external sorting algorithm. Thus they do not give a priority queue-to-sorting reduction. The extra $O(\log_{M/B} \frac{N}{B})$ factor comes from a tree structure with fanout $O(M/B)$ within the priority queue construction. and a key must be moved $\Omega(\log_{M/B} \frac{N}{B})$ times to "bubble up" or "bubble down".

Arge et al. [3] developed a cache-oblivious priority queue that achieves the sorting bound with the tall cache assumption, that is, M is assumed to be of size at least B^2. We note that their structure can serve as a priority queue-to-sorting reduction in the I/O model, by replacing the cache-oblivious sort with a sorting black box. The resulting priority queue supports all operations in $O(\frac{1}{B}\sum_{i\geq 0} S(N^{(2/3)^i})$ amortized I/Os if the sorting algorithm sorts N keys in $NS(N)/B$ I/Os. However, this reduction is not tight for $S(N) = O(\log\log\frac{N}{B})$, and there seems to be no easy way to get rid of the tall cache assumption, even if the algorithm has the knowledge of M and B.

1.3 Reduction in Internal Memory

In this section we describe some high level ideas of Thorup's priority queue in internal memory, which will provide some intuition for our reduction in external memory. Given N keys sorted in ascending order, the priority queue will divide the keys into exponentially increasing subsets called *levels*. The number of levels is $\Theta(\log N)$. The minimum key is contained in the smallest level called the *head*. We employ an atomic heap [6] of size $\Theta(\log N)$ to accomodate insertions before distributing them to corresponding levels. An atomic heap can support updates and predecessor queries in sets of $O(\log^2 N)$ size in constant time, so insertions can be distributed to the corresponding levels in constant time. However, the $O(\log N)$-level structure implies that a key may be moved $O(\log N)$ times in its lifetime. To overcome this, we group the keys in a level into base sets, each of size $\Theta(\log N)$. By moving the pointers to the base sets rather than the base sets themselves, we can move $\Omega(\log N)$ keys in constant time. Base sets are rebalanced by merging and splitting whenever their sizes change by a constant factor, which only adds $O(1)$ to the amortized update cost. We also associate each level with a buffer to accomodate keys before distributing them to the base sets. Finally, the head consists of a single base set, which contains $\Theta(\log N)$ keys. We employ an atomic heap on top of it so that the minimum key can be returned in constant time.

2 Structure

In this section, we describe the structure of our priority queue. In the next section, we show how this structure supports various operations. Finally we analyze the I/O costs of these operations.

Some intuitions. Before diving into technical details, we first give some intuitions on our structure and why it works in the I/O model. We follow Thorup's general framework in our reduction. However, in order to achieve I/O-efficiency, there are several challenges to be addressed. An obvious problem is how to integrate the block size B into the structure. The choice of the parameters appears to be rather important, as we don't know any external structures that are as powerful as atomic heaps, and therefore have to use a delicate recursive structure to get near optimal performance. The second challenge is that the buffer flush and rebalance operations of Thorup's priority queue are not designed to be I/O-efficient. We need to come up with new schemes of maintaining and flushing the buffers in order to achieve I/O efficiency. A third challenge concerns the different ways of handling deletions in internal and external memory. In an internal priority queue, each key is associated with a pointer, so given a deletion we can simply follow the pointer to the key and perform the deletion immediately. In external memory, however, this will incur an extra $\Omega(1)$ I/O cost for a deletion. Deletions are usually supported in a lazy fashion in external memory: we insert a "deleting signal" to the structure, and perform the actual deletions afterwards. However, combining the deleting signal techniques with Thorup's idea turns out to be a non-trivial task; Since we do not have direct access to the base sets, we cannot address the deleting signals when they hit the "lowest level" like buffer tree or any other external priority queues do. As we shall see later, this introduces some more subtle complications, and we have to carefully schedule the operations in order to maintain the shape of the priority queue.

The priority queue consists of multiple layers whose sizes vary from N to cB, where c is some constant to be determined later. The i'th layer from above has size $\Theta(B \log^{(i)} \frac{N}{B})$, for $i \geq 0$, and the priority queue has $O(\log^* N)$ layers. For the sake of simplicity we will refer to a layer by its size. Thus the layers from the largest to the smallest are layer N, layer $B \log \frac{N}{B}$, …, layer cB. Layer cB is also called the head, and is stored in main memory. Given a layer X, its *upper layer* and *lower layer* are layer $B2^{\frac{X}{B}}$ and layer $B \log \frac{X}{B}$, respectively. We use Ψ_X to denote $B2^{\frac{X}{B}}$ and Φ_X to denote $B \log \frac{X}{B}$. The priority queue maintains the invariant that the keys in layer Φ_X are smaller than the keys in layer X. In particular, the minimum key is always stored in the head and can be accessed without I/O cost.

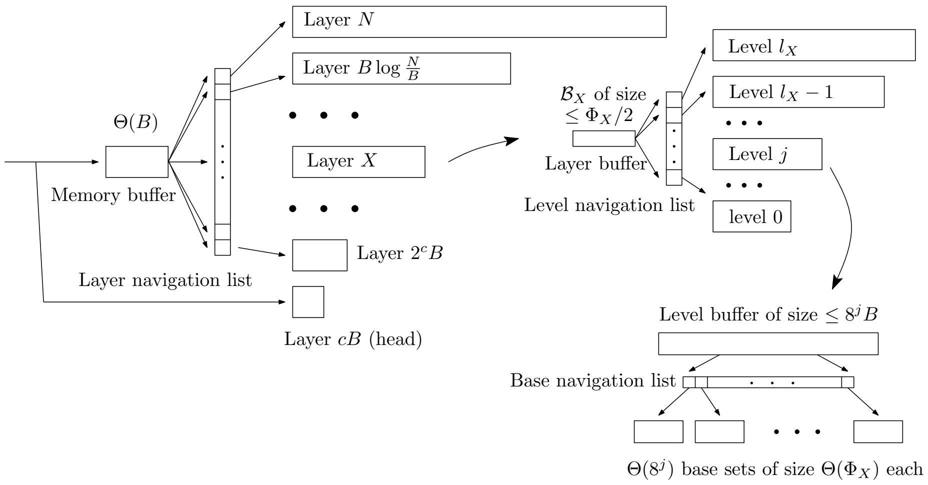


Fig. 1. The components of the priority queue

We maintain a main memory buffer of size $O(B)$ to accommodate incoming insertion and deletion operations. In order to distribute keys in the memory buffer to different layers I/O-efficiently, we maintain a structure called *layer navigation list*. Since this structure will also be used in other components of the priority queue, we define it in a unified way. Suppose we want to distribute the keys in a buffer $\mathcal{B}$ to t sub-structures $S_1, S_2, \dots S_t$. The keys in different sub-structures are sorted relative to each other, that is, the keys in S_i are less or equal to the keys in S_{i+1}. Each sub-structure S_i is associated with a buffer $\mathcal{B}_i$, which accommodates keys transferred from $\mathcal{B}$. The goal is to distribute the keys in $\mathcal{B}$ to each $\mathcal{B}_i$ I/O-efficiently, such that the keys that go to $\mathcal{B}_i$ have values between the minimum keys of S_i and S_{i+1}. A navigation list stores a set of t *representatives*, each representing a sub-structure. The representative of S_i, denoted r_i, is a triple that stores the minimum key of S_i, the number of keys stored in $\mathcal{B}_i$, and a pointer to the last non-full block of the buffer $\mathcal{B}_i$. The representatives are stored consecutively on the disk, and are sorted on the minimum keys. The layer navigation list is built for the $O(\log^* N)$ layers, so it has size $O(\log^* N)$. Please see Figure 1.

Now we will describe the structures inside a layer X except layer cB, which is always in the main memory. First we maintain a *layer buffer* of size $\Phi_X/2$ to store keys flushed from the memory buffer. The main structure of layer X consists of $O(\log \frac{X}{\Phi_X})$ levels with exponentially increasing sizes. The j'th level from the bottom, denoted level j, has size $\Theta(8^j \Phi_X)$. We also keep the invariant that the keys in level j are less or equal to the keys in level $j+1$. We maintain a *level navigation list* of size $\Theta(\log \frac{X}{\Phi_X})$, which represents the $\log \frac{X}{\Phi_X}$ levels. Most keys in level j are stored in $\Theta(8^j)$ disjoint *base sets*, each of size $\Theta(\Phi_X)$. The base sets, from left to right, are sorted relative to each other, but they are not internally sorted. Other than the base sets, there is a *level buffer* of size $8^j B$, which is used to temporarily accommodate keys before distributing them to the base set. We also maintain a *base navigation list* of size $\Theta(8^j)$ for the base sets. Note that we do not impose the level structures on layer cB since it can fit in the main memory. The components of the priority queue are illustrated in Figure 1.

Let l_X denote the top level of layer X. We use $\mathcal{B}_X$ to denote the layer buffer of layer X and $\mathcal{B}_j$ to denote the level buffer of level j when the layer is specified. Our priority queue maintains the following invariants for layer X:

Invariant 1. *The layer buffer $\mathcal{B}_X$ contains at most $\frac{1}{2}\Phi_X$ keys; the level buffer $\mathcal{B}_j$ at layer X contains at most $8^j B$ keys.*

Invariant 2. *The layer buffer $\mathcal{B}_X$ only contains keys between the minimum keys of layer X and its upper layer. The level buffer $\mathcal{B}_j$ only contains keys between the minimum keys of level j and its upper level.*

Invariant 3. *A base set in layer X has size between $\frac{1}{2}\Phi_X$ and $2\Phi_X$; level j of layer X, for $j = 0, 1, \dots, l_X - 1$, has size between $2 \cdot 8^j \Phi_X$ and $6 \cdot 8^j \Phi_X$, and level l_X has size between $2 \cdot 8^{l_X} \Phi_X$ and $40 \cdot 8^{l_X} \Phi_X$.*

Invariant 4. *The head contains at most $2cB$ keys and no delete signal.*

Note that when we talk about the size of a level, we only count the keys in its base sets and exclude the level buffer. The top level has a slightly different size range so that the construction works for any value of X.

We say a layer buffer, a level buffer, a base set, a level or the head overflows if its size exceeds its upper bound in Invariant 1, 3 or 4; we say a base set, a level underflows if its size gets below the lower bound in Invariant 3.

3 Operations

Recall that the priority queue supports three operations: insertion, deletion, and findmin. Since we always maintain the minimum key in the main memory (it is always in the head), the cost of a findmin operation is free. We process deletions in a lazy fashion, that is, when a deletion comes we generate a *delete signal* with the corresponding key and a time stamp, and insert the delete signal to the priority queue. In most cases we treat the delete signals as normal insertions. We only perform the actually delete in the head or during global rebuilding so that the current minimum key is always valid. To ensure linear space usage we perform a global rebuild after every $N/8$ updates.

Our priority queue is implemented by three general operations: *global rebuild*, *flush*, and *rebalance*. A global rebuild operation sorts all keys and processes all delete signals to maintain linear size. A flush operation distributes all keys in a buffer to the buffers of corresponding sub-structures to maintain Invariant 1. A rebalance operation moves keys between two adjacent sub-structures to maintain Invariant 3.

3.1 Global Rebuild

We conduct the first global rebuild when the internal memory buffer is full. Then, after each global rebuild, we set N to be the number of keys in the priority queue, and keep it fixed until the next global rebuild. A global rebuild is triggered whenever layer N (in fact, its top level) becomes unbalanced or the priority queue has received $N/8$ new updates since the last global rebuild. We show that it takes $O(NS(N)/B)$ I/Os to rebuild our priority queue. We first sort all keys in the priority queue and process the delete signals. Then we scan through the remaining keys and divide them into base sets of size Φ_N, except the last base set which may be smaller. This base set is merged to its predecessor if its size is less or equal to $\frac{1}{2}\Phi_N$. The first base set is used to construct the lower layers, and the rest are used to construct layer N. To rebuild the $O(\log \frac{N}{\Phi_N})$ levels of layer N, we scan through the base sets, and take the next $4 \cdot 8^j$ base sets to build level j, for $j = 0, 1, 2, \ldots$. Note that the base navigation list of these $4 \cdot 8^j$ base sets can be constructed when we scan through the keys in the base sets. The level rebuild process stops when we encounter an integer l_N such that the number of remaining base sets is more than $4 \cdot 8^{l_N}$, but less or equal to $4 \cdot (8^{l_N} + 8^{l_N+1}) = 36 \cdot 8^{l_N}$. Then we take these base sets to form the top level of layer N. After the global rebuild, level j has size $4^j\Phi_N$, and the top level l_N has size between $(4 \cdot 8^{l_N} - \frac{1}{2})\Phi_N$ and $(36 \cdot 8^{l_N} + \frac{1}{2})\Phi_N$. For $X = B \log \frac{N}{B}, B \log^{(2)} \frac{N}{B}, \ldots, cB$, layer X are constructed recursively using the same algorithm. All buffers are left empty.

Based on the global rebuild algorithm, the priority queue maintains the following invariant between two global rebuilds:

Invariant 5. *The top level l_X in layer X is determined by the maximum l_X such that*

$$1 + \sum_{j=0}^{l_X} 4 \cdot 8^j \leq \frac{X}{\Phi_X}.$$

The number of layers and the number of levels in each layer will not change between two global rebuilds.

As a result of Invariant 5, we have the following lemma:

Lemma 1. *Suppose the top level in layer X is level l_X. Then l_X is an integer that satisfies the following inequality:*

$$4 \cdot 8^{l_X} \Phi_X \leq X \leq 40 \cdot 8^{l_X} \Phi_X.$$

3.2 Flush

We define the flush operation in a unified way. Suppose we have a buffer $\mathcal{B}$ and k sub-structures $S_1, S_2, \ldots, S_k$. Each S_i is associated with a buffer $\mathcal{B}_i$, and a navigation list L of size k is maintained for the k sub-structures. To flush the buffer $\mathcal{B}$ we first sort the keys in it. Then we scan through the navigation list, and for each representative r_i in L, we read the last non-full block of $\mathcal{B}_i$ to the memory, and fill it with keys in $\mathcal{B}$. When the block is full, we write it back to disk, and allocate a new block. We do so until we encounter a key that is larger than the key in r_{i+1}. Then we update r_i, and advance to r_{i+1}. The I/O cost for a flush is the cost of sorting a buffer of size $|\mathcal{B}|$ plus one I/O for each sub-structure, so we have the following lemma:

Lemma 2. *The I/O cost for flushing keys in buffer $\mathcal{B}$ to k sub-structures is bounded by $O(\frac{|\mathcal{B}|S(|\mathcal{B}|)}{B} + k)$.*

There are three individual flush operations. A *memory flush* distributes keys in the internal memory buffer to $O(\log^* N)$ layer buffers; a *layer flush* on layer X distributes keys in the layer buffer to $O(\log \frac{X}{\Phi})$ level buffers in the layer; and a *level flush* on level j at layer X distributes keys in the level buffer to $\Theta(8^j)$ base sets in the level.

3.3 Rebalance

Rebalancing the base sets. Base rebalance is performed only after a level flush, since this is the only operation that causes a base set to be unbalanced. Consider a level flush in level j of layer X. Suppose the base set A overflows after the flush.

To rebalance A we sort and scan through the keys in it, and split it into base sets of size Φ_X. If the last base set has less than $\frac{1}{2}\Phi_X$ keys we merge it into its predecessor. Note that any base set coming out of a split has between $\frac{3}{2}\Phi_X$ and $\frac{1}{2}\Phi_X$ keys, so it takes at least $\frac{1}{2}\Phi_X$ new updates to any of them before it initiates a new split. Note that after the split we should update the representatives in the base navigation list. This can be done without additional I/Os asymptotically to the level flush operation: We store all new representatives in a temporary list and rebuild the navigation list after all overflowed base sets are rebalanced in level j. A base set never underflows so we do not have a join operation.

Rebalancing the levels. We define two level rebalance operations: *level push* and *level pull.* Consider level j at layer X. When the number of keys in level j (except the top level) gets to more than $6 \cdot 8^j \Phi_X$, a level push operation is performed to move some of its base sets to the upper level. More precisely, we scan through the navigation list of level j to find the first representative r_k such that the number of keys before r_k is larger than $4 \cdot 8^j \Phi_X$. Then we split the navigation list of level j around r_k and attach the second half to the navigation list of level $j+1$. Note that by moving the representatives we also move their corresponding base sets to level $j+1$. By Invariant 3, the number of keys in a base set is at most $2\Phi_X$, so the new level j has size between $4 \cdot 8^j \Phi_X$ and $(4 \cdot 8^j + 2)\Phi_X$. Finally, to maintain Invariant 2 we sort level buffer $\mathcal{B}_j$ and move keys larger than the r_k to the level buffer $\mathcal{B}_{j+1}$.

Conversely, if the number of keys in level j gets below $2 \cdot 8^j \Phi_X$ (except the top level), a level pull operation is performed. We cut a proportion of the navigation list of level $j+1$ and attach it to the navigation list of level j, such that the number of keys in level i becomes between $4 \cdot 8^j \Phi_X$ and $(4 \cdot 8^j - 2)\Phi_X$. We also sort $\mathcal{B}_{j+1}$, the buffer of level $j+1$, and move the corresponding keys to level buffer $\mathcal{B}_j$.

Observe that after a level push/pull, the number of keys in level j is between $(4 \cdot 8^j - 2)\Phi_X$ and $(4 \cdot 8^j + 2)\Phi_X$, so it takes at least $\Omega(8^j \Phi_X)$ new updates before the level needs to be rebalanced again. The main reason that we adopt this level rebalance strategy is that it does not touch all keys in the level; the rebalance only takes place on the base navigation lists and the keys in the level buffers.

Rebalancing the layers. When the top level l_X of layer X becomes unbalanced, we can no longer rebalance it only using the navigation list. Recall that its upper level is level 0 in layer Ψ_X. For simplicity we will refer to the two levels as level l_X and level 0, without specifying their layers. We also define two operations for rebalancing a layer: *layer push* and *layer pull.* A layer push is performed when the layer overflows, that is, the number of keys in level l_X gets more than $40 \cdot 8^{l_X} \Phi_X$. In this case we sort all keys in level l_X and level 0 together, then use the first $4 \cdot 8^{l_X} \Phi_X$ keys to rebuild level l_X and the rest to rebuild level 0. Recall that to rebuild a level we scan through the keys and divide them into base sets of size Φ_X, except the last one which has size between $\frac{1}{2}\Phi_X$ and $\frac{3}{2}\Phi_X$, and then we scan through the keys again to build the base navigation list. Note that the rebuild operation will change the minimum key in layer Ψ_X, so we update the layer navigation list accordingly. Finally we sort the keys in the layer buffer $\mathcal{B}_X$ and the level buffer $\mathcal{B}_{l_X}$, and move the keys larger than the new minimum key of layer Ψ_X to the level buffer $\mathcal{B}_0$.

A layer pull operation is performed when the layer underflows, that is, there are less than $2 \cdot 8^{l_X} \Phi_X$ keys in level l_X. A layer pull proceeds in the same way as a layer push does, except for the last step. Here we sort the layer buffer $\mathcal{B}_{\Psi_X}$ and the level buffer $\mathcal{B}_0$ and move the keys smaller than the new minimum key to the level buffer $\mathcal{B}_{l_X}$. After a layer push or pull, the number of keys in level l_X is $4 \cdot 8^{l_X} \Phi_X$. By lemma 1, we have $40 \cdot 8^{l_X} \Phi_X \geq X$, so it takes at least $2 \cdot 8^{l_X} \Phi_X = \Omega(X)$ new updates to layer X before we initiate a new push or a pull again.

Note that since we do not impose the level structure on the head layer cB, we need to design the layer push and layer pull operations specifically for it. A layer push is performed when the number of keys in the head gets to more than $2cB$. We sort all keys in it and level 0 of layer Ψ_{cB}, and use the first cB keys to rebuild the head and the rest to rebuild level 0. A layer pull is performed when the head becomes empty. The operation processes in the same way as a layer push does, except that after rebuilding both levels, we sort the layer buffer $\mathcal{B}_{\Psi_{cB}}$ and the level buffer $\mathcal{B}_0$ together, and move the keys smaller than the new minimum key of layer $\mathcal{B}_{\Psi_{cB}}$ to the head.

3.4 Scheduling Flush and Rebalance Operations

A key component in our priority queue construction is a schedule of the flush and rebalance operations. This is mainly due to the introduction of the deleting signals. Since we are only able to process deleting signals in the head, an update may cause the priority queue to shrink and expand multiple times. In order to maintain the shape of the priority queue, we need to schedule the operations delicately. When a new update comes, we insert it to the head if it is smaller than the maximum key in the head, and to the memory buffer if otherwise.If a delete signal is inserted to the head we process it so that invariant 4 is maintained. Whenever the memory buffer overflows or the head becomes empty or overflowed we start to update the priority queue. This process is divided into three stages: the flush stage, the push stage, and the pull stage. In the flush stage we flush all overflowed buffers and rebalance all unbalanced base sets; in the push stage we use push operations to rebalance all overflowed layers and levels. We treat delete signals as insertions in the flush stage and the push stage. In the pull stage we deal with delete signals and use pull operations to rebalance all underflowed layers and levels.

In the flush stage, we initialize a queue Q_o to keep track of all overflowed buffers and a doubly linked list L_o to keep track of all overflowed levels. The buffers are flushed in a BFS fashion. First we flush the memory buffer into $O(\log^* N)$ layer buffers. After flushing the memory buffer, we insert the representatives of the overflowed layer buffers into Q_o, from bottom to top. We also check whether the head overflows after the memory flush. If so, we insert its representatives to the beginning of L_o. Then we start to flush the layer buffers in Q_o. Again, when flushing a layer buffer we insert the representatives of the overflowed level buffers to Q_o from bottom to top. After all layer buffers are flushed, we begin to flush level buffers in Q_o. After each level flush, we rebalance all unbalanced base sets in this level, and if the level overflows we add the representative of this level to the end of L_o. Note that the representatives in L_o are sorted on the minimum keys of the levels.

After all overflowed level buffers are flushed, we enter the push stage and start to rebalance levels in L_o in a bottom-up fashion. In each step, we take out the first level in L_o (which is also the current lowest overflowed level) and rebalance it. Suppose this level is level j of layer X. If it is not the top level or the head layer we perform a level push; otherwise we perform a layer push. Then we delete the representative of this level from L_o. A level push may cause the level buffer of level $j+1$ to be overflowed, in which case we flush it and rebalance the overflowed base sets. Then we check whether level $j+1$ overflows. If so, we insert the representative of level $j+1$ to the head of L_o (unless it is already at the beginning of L_o) and perform a level push on level $j+1$. Otherwise we take out a new level in L_o and continue the process. When the top level of layer N becomes unbalanced we simply perform a global rebuild.

After rebalancing all levels, we enter the pull stage and start to process the delete signals. This is done as follows. We first process all delete signals in the head. If the head becomes empty we perform a layer pull to get more keys into the head. This may cause higher levels or layers to underflow, and we keep performing level pulls and layer pulls until all levels and layers are balanced. Consider a level pull or layer pull on level j of layer X. After the level pull or layer pull the level buffer $\mathcal{B}_j$ may overflow. If so, we flush it and rebalance the base sets when necessary. Note that this may cause the size of level j to grow, but it will not overflow, as we will show later, so that we do not need push operations in the pull stage. After all levels and layers are balanced, we process the delete signals in the head again. We repeat the pull process until there are no delete signals left in the head and the head is non-empty.

4 Correctness and I/O Cost Analysis

We note that with inappropriate scheduling, it is possible that the pull stage fails due to lack of keys or overflows in some levels. The following two lemmas guarantee that in our scheduling, the pull stage will succeed.

Lemma 3. *When we perform a level pull on level j, there are enough keys in level $j+1$ to rebalance level j; When we perform a layer pull on layer X, there are enough keys in level 0 of layer Ψ_X to rebalance level l_X.*

Lemma 4. *A level or a layer never overflows in the pull stage.*

The following lemma states that the I/O cost for each update is very close to $S(N)/B$, which directly implies Theorem 1.

Lemma 5. *The amortized I/O cost per update for the priority queue is bounded by $O(\frac{1}{B}\sum_{i=0} S(B\log^{(i)}\frac{N}{B}))$.*

Due to space limitations, we omit the proofs of above lemmas from this extended abstract; all missing proofs can be found in the full version of the paper [13].

References

1. Aggarwal, A., Vitter, J.S.: The input/output complexity of sorting and related problems. Communications of the ACM 31(9), 1116–1127 (1988)
2. Arge, L.: The buffer tree: A technique for designing batched external data structures. Algorithmica 37(1), 1–24 (2003)
3. Arge, L., Bender, M., Demaine, E., Holland-Minkley, B., Munro, J.: Cache-oblivious priority queue and graph algorithm applications. In: Proc. ACM Symposium on Theory of Computing, pp. 268–276. ACM (2002)
4. Brodal, G., Katajainen, J.: Worst-case efficient external-memory priority queues. In: Proc. Scandinavian Workshop on Algorithms Theory, pp. 107–118 (1998)
5. Fadel, R., Jakobsen, K., Katajainen, J., Teuhola, J.: Heaps and heapsort on secondary storage. Theoretical Computer Science 220(2), 345–362 (1999)
6. Fredman, F.W., Willard, D.E.: Trans-dichotomous algorithms for minimum spanning trees and shortest paths. In: Proc. 31st Annu. IEEE Sympos. Found. Comput. Sci., pp. 719–725 (1990)
7. Han, Y.: Deterministic sorting in $o(n \log \log n)$ time and linear space. Journal of Algorithms 50(1), 96–105 (2004)
8. Han, Y., Thorup, M.: Integer sorting in $o(n\sqrt{\log \log n})$ expected time and linear space. In: Proc. IEEE Symposium on Foundations of Computer Science, pp. 135–144. IEEE (2002)
9. Larsen, K.G.: The cell probe complexity of dynamic range counting. In: Proc. ACM Symposium on Theory of Computing (2012)
10. Pătraşcu, M.: Unifying the landscape of cell-probe lower bounds. SIAM Journal on Computing 40(3) (2011)
11. Thorup, M.: Equivalence between priority queues and sorting. Journal of the ACM 54(6), 28 (2007)
12. Vitter, J.: External memory algorithms and data structures: Dealing with massive data. ACM Computing Surveys 33(2), 209–271 (2001)
13. Wei, Z., Yi, K.: Equivalence between Priority Queues and Sorting in External Memory. arXiv: 1207.4383 (2014)

Amortized Bounds for Dynamic Orthogonal Range Reporting

Bryan T. Wilkinson[*]

MADALGO, Aarhus University, Aarhus, Denmark
btw@cs.au.dk

Abstract. We consider the fundamental problem of 2-D dynamic orthogonal range reporting for 2- and 3-sided queries in the standard word RAM model. While many previous dynamic data structures use $O(\log n/\log\log n)$ update time, we achieve faster $O(\log^{1/2+\epsilon} n)$ and $O(\log^{2/3+\epsilon} n)$ update times for 2- and 3-sided queries, respectively. Our data structures have optimal $O(\log n/\log\log n)$ query time. Only Mortensen [14] had previously lowered the update time convincingly below $O(\log n)$, with 3- and 4-sided data structures supporting updates in $O(\log^{5/6+\epsilon} n)$ and $O(\log^{7/8+\epsilon} n)$ time, respectively. In practice, fast updates are often as important as fast queries, so we make a step forward for an important problem that has not seen any progress in recent years.

We also obtain new results for the special case of 3-sided insertion-only emptiness, showing that the difference in complexity between fully dynamic and partially dynamic 2-D orthogonal range reporting can be significant (i.e., $\Omega(\text{polylog}\, n)$ factor differences). In particular, we achieve $O((\log n \log\log n)^{2/3})$ update time and $O((\log n \log\log n)^{1/3})$ query time. At the other end of our update/query trade-off curve, we achieve $O(\log n/\log\log n)$ update time and $O(\log\log n)$ query time. In contrast, in the pointer machine model, there are only $O(\log\log n)$ factor differences between the complexities of fully dynamic and partially dynamic 2-D orthogonal range reporting.

1 Introduction

We consider various special cases of 2-D dynamic orthogonal range reporting, a fundamental problem in computational geometry. Range reporting has many applications in, for example, databases and information retrieval, since it is often useful to model objects with d attributes as points in $\mathbb{R}^d$, where each dimension represents an attribute. Performing an orthogonal range reporting query then corresponds to filtering objects with inequality filters on attributes. The 2-D orthogonal range reporting problem has been studied for over 30 years, but optimal bounds are still not known. We give the first improved bounds since the work of Mortensen [14] in 2006. In particular, we give data structures with faster

[*] Work supported in part by the Danish National Research Foundation grant DNRF84 through the Center for Massive Data Algorithmics (MADALGO) and in part by the Natural Sciences and Engineering Research Council of Canada (NSERC).

updates, maintaining optimal query times. In practice, fast updates are often as important as fast queries, since fast updates allow the efficient maintenance of a larger number of specialized indices, which can then support a more diverse set of queries efficiently. We also give partially dynamic data structures with faster query times than can be achieved by fully dynamic data structures.

Problems. The *range reporting* problem involves maintaining a set P of n points from $\mathbb{R}^d$ so that given an online sequence of queries of the form $Q \subseteq \mathbb{R}^d$, we can efficiently compute $P \cap Q$. The output size, k, of a range reporting query Q is $|P \cap Q|$. The *range emptiness* problem requires only that the data structure decides whether or not $P \cap Q$ is empty. In *dynamic* range reporting, insertions and deletions of points to and from P may be interspersed with the online sequence of queries. In *incremental* range reporting, the set P is initially empty and there are no deletions (i.e., there are only insertions and queries). In *orthogonal* range reporting, queries are restricted to the set of axis-aligned hypercubes. That is, a query must be of the form $[\ell_1, h_1] \times [\ell_2, h_2] \times \cdots \times [\ell_d, h_d]$. An *s-sided* query range for $d \leq s \leq 2d$, has finite boundaries in both directions along $s - d$ axes and only one finite boundary along $2d - s$ axes. For example, the range $[\ell, r] \times (-\infty, t]$ is a 3-sided 2-D query range.

In the *range minimum query (RMQ)* problem, we maintain an array A of n elements from $\mathbb{R}$ so that given an online sequence of queries of the form $(i, j) \in [n] \times [n]$ such that $i \leq j$, we can efficiently compute $\min\{A[k] \mid i \leq k \leq j\}$. That is, we compute the minimum element in a given query subarray. RMQ can be applied to solve the lowest common ancestor problem in trees and it is often used in solutions to textual pattern matching problems. In the dynamic variant of this problem, an update $(i, v) \in [n] \times \mathbb{R}$ sets $A[i]$ to v. In the *decremental* variant of the problem, the update may not increase $A[i]$. That is, if $A[i]$ contains u prior to the update, then it must be that $v \leq u$.

We consider 2- and 3-sided 2-D orthogonal range reporting. In particular, we consider dynamic reporting as well as incremental emptiness. We also consider RMQ due to its close connection to 3-sided emptiness.

Model. Our model of computation is the standard w-bit word RAM model. We assume that our points have integer coordinates that each fit in a single word, so $P \subseteq [U]^d$, where $w \geq \log U$. Similarly, the bounds of each query range are from $[U]$. We make the standard assumption that $w \geq \log n$ so that an index into our input array fits in a single word.

Previous Results. In the pointer machine model, the best known bounds for dynamic orthogonal range reporting are $O(n \log n)$ space, and $O(\log n \log\log n)$ update time, and $O(\log n \log\log n + k)$ query time. These bounds are achieved by augmenting range trees [5] with dynamic fractional cascading [13]. The doubly logarithmic factors can be eliminated for the partially dynamic variants of the problem. The priority search tree [12] solves 3-sided dynamic orthogonal range reporting optimally in the pointer machine model with only linear space, $O(\log n)$ update time, and $O(\log n + k)$ query time.

In the word RAM model, sublogarithmic time operations were first obtained for 3-sided queries. Willard [15] reduces both the update time and the query time of the priority search tree to $O(\log n/\log\log n)$. Let T_u be the time for an update operation and let T_q be the time for a query operation. Alstrup et al. [1] give a cell probe lower bound for 2-D orthogonal range emptiness, showing that $T_q = \Omega(\log n/\log(T_u \log n))$. This lower bound implies a lower bound of $\Omega(\log n/\log\log n)$ on the time per operation (i.e., $\max\{T_u, T_q\}$). These lower bounds hold even when restricted to 2-sided queries. Crucially, they hold for the amortized bounds of fully dynamic data structures as well as for the worst-case bounds of partially dynamic data structures, but not for the amortized bounds of partially dynamic data structures. For some intuition, imagine an adversary forces an extremely expensive insertion. In the fully dynamic case, it can then undo the insertion with a deletion and repeat the expensive operation over and over again. However, in the incremental case, the adversary cannot repeat the operation and the rest of the insertions might require very little work, resulting in low amortized time per insertion.

Despite mounting evidence that the true update and query times for 2-D orthogonal range reporting might be $\Theta(\log n/\log\log n)$, Mortensen [14] showed that updates could in fact be made faster. For 3-sided queries, his data structure supports updates in $O(\log^{5/6+\epsilon} n)$ time. He gives a worst-case deterministic fully dynamic data structure for 4-sided queries that requires $O(n\log^{7/8+\epsilon} n)$ space, $O(\log^{7/8+\epsilon} n)$ update time, and optimal $O(\log n/\log\log n)$ query time. The speed up in update time is achieved by reducing the number of bits required to specify points and packing multiple points into a word for efficient parallel processing. Importantly, Mortensen shows that update time can be reduced convincingly (i.e., by an $\Omega(\text{polylog}\, n)$ factor) below $O(\log n)$ while maintaining optimal $O(\log n/\log\log n)$ query time.

Our Results. We make an initial effort to determine the complexity of updates by first considering the 2- and 3-sided special cases and allowing amortization and randomization. All of our update and query bounds are amortized. The only source of randomization in our data structures originates from our use of randomized dynamic predecessor search data structures [7]. It is possible to eliminate all randomization from our data structures by instead using the deterministic predecessor search data structure of Andersson and Thorup [2], at the expense of doubly logarithmic factors. All of our data structures require only linear space.

Our results are summarized in Table 1. Note that $\epsilon > 0$ is an arbitrarily small constant and $k = |P \cap Q|$ is the output size of a reporting query. Each of our reporting data structures with a query time of the form $O(t(n)+k)$ can be easily adapted to decide emptiness in $O(t(n))$ time. Also, pred is the cost of an update or query to a linear-space dynamic predecessor search data structure containing at most n elements from a universe of size U. These updates and queries can be performed in $O(\log\log U)$ [7] time, $O(\log_w n)$ time [8], or $O(\sqrt{\log n/\log\log n})$ time [2]. For each 3-sided emptiness result there is a corresponding RMQ result due to the close relationship between these two problems.

Table 1. Our results

Problem	Update Time	Query Time
2-sided reporting	$\log^{1/2+\epsilon} n$	$\log n / \log\log n + k$
2-sided reporting	$(\log n \log\log n)^{1/2}$	$\log n + k$
3-sided reporting	$\log^{2/3+\epsilon} n$	$\log n / \log\log n + k$
3-sided reporting	$(\log n \log\log n)^{2/3}$	$\log n + k$
RMQ	$\log^{2/3+\epsilon} n$	$\log n / \log\log n$
RMQ	$(\log n \log\log n)^{2/3}$	$\log n$
2-sided incremental emptiness	$\text{pred} + \log\log n$	$\text{pred} + \log\log n$
3-sided incremental emptiness	$\text{pred} + (\log n \log\log n)^{2/3}$	$\text{pred} + (\log n \log\log n)^{1/3}$
3-sided incremental emptiness	$\text{pred} + \log n / \log\log n$	$\text{pred} + \log\log n$
decremental RMQ	$(\log n \log\log n)^{2/3}$	$(\log n \log\log n)^{1/3}$
decremental RMQ	$\log n / \log\log n$	$\log\log n$

For 2- and 3-sided reporting, we obtain data structures with optimal query time and update times of the form $O(\log^{\gamma} n)$ for $\gamma < 1$. Mortensen [14] previously gave a 3-sided data structure with $\gamma = 5/6 + \epsilon$. We improve the exponent to $\gamma = 2/3 + \epsilon$ for 3-sided queries and even further to $\gamma = 1/2 + \epsilon$ for 2-sided queries. It is plausible that an exponent of $\gamma = 1/2$ is optimal.

We circumvent the lower bound of Alstrup et al. [1] by considering amortized solutions to partially dynamic problems. We give an optimal 2-sided data structure with operations that run in only $O(\text{pred} + \log\log n)$ time. We finally give 3-sided incremental emptiness data structures. At one end of our update/query trade-off curve, we obtain update and query times that are both of the form $O(\log^{\gamma} n)$ for $\gamma < 1$, showing that there can be $\Omega(\text{polylog}\, n)$ factor differences between the complexities of fully dynamic and partially dynamic reporting problems. In contrast, in the pointer machine model, there are only $O(\log\log n)$ factor differences between the complexities of the fully dynamic and partially dynamic problems. At the other end of our trade-off curve, we obtain queries that run in only $O(\text{pred} + \log\log n)$ time. These latter bounds could plausibly be optimal.

Our Approach. Mortensen [14] develops a framework which essentially uses a high-fanout priority search tree to reduce the problem of 3-sided range reporting to the same problem, but with fewer points, of which multiple can be packed into a single word. Once points are packed into words, we encounter a situation that can be modeled very closely to the external memory model. An important divergence from the work of Mortensen [14] is that we start by explicitly designing external memory data structures. This both simplifies our presentation and allows reuse of previous external memory results. We both reuse the framework of Mortensen [14] and extend it to consider the more special case of 2-sided queries and incremental emptiness. In the latter case, it turns out that using a high-fanout range tree yields better results than the high-fanout priority search tree of Mortensen [14].

2 Preliminaries

Dynamic and Static Axes. An *axis* is a set of coordinates from which points and query ranges obtain their coordinates along some dimension. Axes may also specify restrictions on how points are assigned coordinates. A *standard axis* is an axis with coordinates from $[U]$ that accommodates at most n points with distinct coordinates. Our geometric problems have two standard axes.

A standard technique in static orthogonal range reporting is rank space reduction. Instead of using the full coordinates of points, we use their x- and y-ranks. Thus, each point is specified by ranks from $[n]$ instead of coordinates from $[U]$. To handle a query, we must perform a predecessor search for each side of the query range in order to reduce the query range to rank space. In the dynamic case, rank space reduction as such does not work since inserting a new point might require updating the ranks of all of the other points. To encapsulate this problem, Mortensen [14] introduces the concept of a *dynamic axis.* A dynamic axis has coordinates from the set of nodes of a linked list such that one node u is less than another node v if u occurs prior to v in the linked list. There is a bijection between points and linked list nodes. Initially, a new point has no node, so an update specifies its position along the axis with a pointer to its predecessor node. A new node is inserted to the linked list after the predecessor node. Query coordinates are specified by pointers to linked list nodes. A dynamic axis of size u can accommodate at most u points/nodes.

It is easy to reduce a standard axis to a dynamic axis via predecessor search. This is why some of our results include pred terms. In some cases, in choosing $\text{pred} = O(\sqrt{\log n/ \log\log n})$, the pred terms are dominated by the other terms of the update/query times. In these cases, we have omitted the pred terms.

In solving the problem encapsulated by dynamic axes, we will end up with some form of reduction of coordinates to a smaller set of integers (though not to the set of ranks as in rank space reduction). A *static axis* of size u has coordinates from $[u]$ and accommodates at most u points with distinct coordinates.

3-*Sided Emptiness and RMQ.* For any set of points P, a 3-sided range of the form $[\ell, r] \times (-\infty, t]$ is empty if and only if the lowest point of P in the slab $[\ell, r] \times (-\infty, +\infty)$ has y-coordinate less than or equal to t. Assume we have an array A of n elements from $[U]$. We construct a specific set P of points such that for each $i \in [n]$, there is a point $(i, A[i]) \in P$. Then, the result of an RMQ (i, j) on A is the y-coordinate of the lowest point of P in the slab $[i, j] \times (-\infty, +\infty)$. So, both types of queries can be solved by finding the lowest point in a query vertical slab. This is precisely how our emptiness data structures operate. Consider an update (i, v) which sets $A[i]$, initially containing u, to v. We update P by deleting (i, u) and inserting (i, v). So, an RMQ update is at most twice as expensive as a 3-sided emptiness update. In the case of decremental RMQ, we are guaranteed that $v \leq u$. Assuming we have a 3-sided incremental emptiness data structure that can handle multiple points with the same x-coordinate, it is sufficient to update P by only inserting (i, v) without first deleting (i, u). It is easy to modify our incremental emptiness data structures to handle multiple points with the same x-coordinate by, in this case, implicitly deleting (i, u). Thus, (decremental) RMQ can thus be solved by a 3-sided (incremental) emptiness data structure with a static x-axis of size n and a standard y-axis. In order to reduce the standard y-axis to a dynamic axis, we require predecessor search along the y-axis only for updates, since an RMQ is specified with only two x-coordinates and no y-coordinates.

Notation. We borrow and extend the notation of Mortensen [14] to specify various different 2-D range reporting problems. A dynamic problem is specified by $T^s(t_x : u_x, t_y : u_y)$ where $T \in \{\mathrm{R}, \mathrm{R_I}, \mathrm{E}, \mathrm{E_I}\}$; $d \leq s \leq 2d$; $t_x, t_y \in \{\mathrm{d}, \mathrm{s}\}$; and $1 \leq u_x, u_y \leq n$. Depending on T, the problem is either s-sided dynamic reporting (R), s-sided incremental reporting ($\mathrm{R_I}$), s-sided dynamic emptiness (E), or s-sided incremental emptiness ($\mathrm{E_I}$). For each axis $a \in \{x, y\}$, depending on t_a, the axis is either dynamic (d) or static (s). The axis has size u_a. We assume without loss of generality that 2-sided queries are of the form $(-\infty, r] \times (-\infty, t]$ and 3-sided queries are of the form $[\ell, r] \times (-\infty, t]$.

We say a data structure has performance $(\text{Update} : T_u, \text{Query} : T_q, \text{Space} : S)$ if it requires $O(T_u)$ update time, $O(T_q)$ query time (or $O(T_q + k)$ query time for reporting problems), and $O(S)$ words of space. We originally defined n as $|P|$, but we redefine it now to an upper bound on $|P|$ that we know in advance and for which we have the guarantee that $w \geq \log n$. We give the performances of data structures as functions of the sizes of their axes rather than of $|P|$. Our final data structures have axes of size n. We can eliminate the requirement of knowing n in advance and simultaneously ensure that performance is a function of $|P|$ rather than n by initially building a data structure with axes of constant size, rebuilding the data structure with axes twice as large whenever the data structure fills up, and rebuilding the data structure with axes half as large whenever the data structure is only a quarter full (except when the axes are already at their initial constant size).

When considering external memory data structures, we denote the input size by N and the output size by K, in keeping with the conventions of the external memory literature. A block can hold B elements and internal memory can hold M elements. In the external memory setting, we say a data structure has performance $(\text{Update} : T_u, \text{Query} : T_q, \text{Space} : S)$ if it requires $O(T_u)$ I/Os for an update, $O(T_q)$ I/Os for a query (or $O(T_q + K/B)$ I/Os for a reporting query), and $O(S)$ blocks of space.

3 Data Structures

3.1 Dynamic Reporting

An important technique we will apply involves performing some form of reduction of the coordinates of small sets of $u \leq n$ points in such a way that we can describe each point using only $O(\log u)$ bits. Thus, since $w \geq \log n$, we can pack $O(\log n / \log u)$ of these points into a single word. Using standard word operations and table lookups, we can then operate on multiple points at unit cost.

A similar situation arises in the external memory model: multiple points fit in a block and we can operate on all of the points in all of the blocks in internal memory at no cost. In fact, it is possible to simulate external memory algorithms to solve problems on packed words. For this reason, we begin with the design of an external memory reporting data structure, which we intend to simulate in the word RAM.

Lemma 1. *For any* $f \in [2, B]$*, there exists an external memory data structure for* $\mathrm{R}^3(\mathrm{s} : N, \mathrm{s} : N)$ *with performance* $(\mathrm{Update} : (f/B)\log_f N, \mathrm{Query} : \log_f N + (f/B)K, \mathrm{Space} : N/B)$*.*

Proof. The data structure is a modified I/O tournament tree (I/O-TT) [11]. The I/O-TT is an I/O-efficient priority queue data structure, but we adapt it to answer 3-sided range reporting queries. The I/O-TT stores elements, each of which consists of a key and a priority. We map a point to an element so that the point's x-coordinate is the element's key and the point's y-coordinate is the element's priority.

The I/O-TT is a static binary tree on x-coordinates where each node is associated with a set of at most B points and an update buffer of size B[1]. The sets of points are in heap order by y-coordinate and a non-root node may only contain points if its parent is full. Updates are initially sent to the root node. An update is inserted into a node's update buffer if it cannot be resolved in the node directly. When a node's update buffer is filled, the updates are flushed down to the node's children (by x-coordinate) in $O(1/B)$ amortized I/Os per update. The total cost of an update is thus $O((1/B)h)$ I/Os, where h is the height of the I/O-TT. A query to the I/O-TT involves finding the point with minimum y-coordinate. This point is in the root and can thus be found in $O(1)$ I/Os. The I/O-TT requires $O(N/B)$ blocks of space.

Our first modification is to use the I/O priority search tree [3] query algorithm to handle 3-sided reporting queries instead of priority queue queries. Ignoring the update buffers, the I/O-TT is essentially an I/O priority search tree with fanout of 2 instead of B. The query algorithm of the I/O priority search tree performs $O(1)$ I/Os in $O(h + K/B)$ nodes. However, this query algorithm is only correct if the updates in the buffers of all of these nodes, as well as all of their ancestors, have been flushed. We flush these $O(h + K/B)$ buffers as the query algorithm proceeds. These flushing operations may cascade to descendant nodes, but all I/Os performed while cascading are charged to update operations. Therefore, there are only $O(h + K/B)$ additional I/Os that cannot be charged to update operations.

Our second modification is to increase the fanout of the I/O-TT to f. To ensure that we can still flush updates from a node to its children efficiently, we must reduce the sizes of the sets of points and update buffers to B/f. In this way, we can store these sets for all of a node's children in $O(1)$ blocks and thus flush an update buffer in $O(f/B)$ amortized I/Os per update. As a side-effect,

[1] In the original description of the I/O-TT these sets have size M instead of B, but it is easy to see that such large sets are not necessary to achieve the desired bounds.

the I/O priority search tree query algorithm then visits $O(h + (f/B)K)$ nodes. Since the height of the I/O-TT is now $O(\log_f N)$, we are done. □

Let $m \geq 2$ be the smallest constant such that any of our external memory data structures can operate with $M = mB$. Let $\delta > 0$ be a sufficiently small constant to be determined later.

Lemma 2. *For any* $u \in [n^{\delta/2m}]$ *and* $f \in [2, (\delta/2m)\log n/\log u]$*, there exists a data structure for* $\mathrm{R}^3(\mathrm{s} : u, \mathrm{s} : u)$ *with performance* $(\text{Update} : 1 + f\log^2 u/\log n,$ $\text{Query} : \log_f u, \text{Space} : u)$.

Proof. We simulate the data structure of Lemma 1 in the word RAM. Since an element (a point or block pointer) requires at most $2\log N$ bits, if $N = u$, we can store $(\delta/2)\log n/\log u$ elements in $\delta\log n$ bits in a single word. We designate a single word to act as our simulated main memory containing up to $M = (\delta/2)\log n/\log u$ elements. The rest of our actual main memory acts as our simulated external memory: it is divided into blocks of $B = (\delta/2m)\log n/\log u$ elements such that $M = mB$. Since $u \leq n^{\delta/2m}$, each block can hold at least one element. A constant number of standard word operations can transfer a simulated block into or out of our simulated main memory.

The update and query algorithms of Lemma 1 may perform arbitrary manipulations of the elements in main memory between I/Os. Since the algorithms are finite and do not depend on the input size, there are only a constant number of different manipulations of the elements in main memory. Since our simulated main memory can be described in exactly $\delta\log n$ bits, we use a global lookup table containing n^δ entries of $\delta\log n$ bits to support constant-time simulations of each of these manipulations. Since our final data structures reuse these lookup tables for many instances of our intermediate data structures (such as this one), we do not include the space for global lookup tables in the space bounds for our intermediate data structures. The lookup tables can be built in time polynomial in their number of entries. We set δ sufficiently small so that they can be built in $O(n)$ time. Whenever we rebuild our final data structures to handle constant factor changes in n, we can also afford to rebuild our global lookup tables.

Substituting $N = u$ and $B = (\delta/2m)\log n/\log u$ into Lemma 1 (and converting from space consumption in blocks to space consumption in words) gives the desired bounds. We also need to add a constant term to the running times of the update and query algorithms, since they may read from and write to simulated main memory a constant number of times without performing any simulated I/Os. In the external memory model, these reads and writes to main memory are free, but they are not free in our simulation. Also, since we need to report each output point individually, the $(f/B)K$ term in the query time simply becomes k. □

We now extract and slightly generalize two techniques from the framework of Mortensen [14] and encapsulate them in the following two lemmata. The first technique involves converting a data structure with a static axis to a data structure with a dynamic axis. It can also be used as a space reduction technique.

Lemma 3 (Lemma 24 of Mortensen [14]). *Given a data structure for* T^s $(t_x : u, t_y : u)$ *for* $u \in [n]$ *with performance* $(\mathrm{Update} : T_u, \mathrm{Query} : T_q, \mathrm{Space} : u \operatorname{polylog} u)$ *where* $t_a = \mathrm{s}$*, for some* $a \in \{x, y\}$*, and queries have only one finite boundary along the other axis, then there exists a data structure for the same problem with* $t_a = \mathrm{d}$ *and performance* $(\mathrm{Update} : \log\log u + T_u, \mathrm{Query} : \log\log u + T_q, \mathrm{Space} : u)$*.*

The second technique involves converting a data structure that can only handle $u \leq n$ points to a data structure that handles n points. This is achieved using a u-ary priority search tree.

Lemma 4 (Lemma 23 of Mortensen [14]). *Given a data structure for* $T^s(\mathrm{s} : u, \mathrm{d} : u)$ *for* $u \in [n]$ *with performance* $(\mathrm{Update} : T_u, \mathrm{Query} : T_q, \mathrm{Space} : u)$ *where queries have only one finite boundary along the* y*-axis, then there exists a data structure for* $T^s(\mathrm{s} : n, \mathrm{d} : n)$ *with performance* $(\mathrm{Update} : (\log n / \log u)(\log\log n + T_u), \mathrm{Query} : (\log n / \log u)(\log\log n + T_q), \mathrm{Space} : n)$*.*

We note that $\mathrm{R}^2(\mathrm{s} : u, \mathrm{s} : u)$ is a special case of $\mathrm{R}^3(\mathrm{s} : u, \mathrm{s} : u)$, so the data structure of Lemma 2 also solves $\mathrm{R}^2(\mathrm{s} : u, \mathrm{s} : u)$. Since 2-sided queries have only one finite boundary along the x-axis, we can apply Lemma 3 to the data structure of Lemma 2 in order to obtain a dynamic y-axis, which then allows us to apply Lemma 4 (with $u = 2^{(\delta/2m)((1/f)\log n \log\log n)^{1/2}}$) to handle n points. Finally, another application of Lemma 3 converts the still static x-axis into a dynamic axis and we obtain the following theorem.

Theorem 1. *For any* $f \in [2, \log n / \log\log n]$*, there exists a data structure for* $\mathrm{R}^2(\mathrm{d} : n, \mathrm{d} : n)$ *with performance* $(\mathrm{Update} : (f \log n \log\log n)^{1/2}, \mathrm{Query} : (f \log n \log\log n)^{1/2} + \log_f n, \mathrm{Space} : n)$*.*

We obtain the 2-sided reporting data structures of Table 1 by setting $f = \log^{\epsilon'} n$ for some positive constant $\epsilon' < 2\epsilon$, or alternatively setting $f = 2$. All of the techniques we have seen so far carry through for both 2- and 3-sided queries, except for the first application of Lemma 3. Therefore, in order to obtain results for 3-sided queries we only need some way to support a dynamic y-axis for 3-sided queries.

We proceed by designing an external memory data structure for online list labelling which we will use to augment our original external memory data structure with a dynamic y-axis. In the online list labelling problem, we maintain an assignment of labels from some universe of totally ordered labels to linked list nodes so that the labels are monotonically increasing along the list. Assume the linked list has at most n nodes. For a universe of size $O(n)$, an insertion or deletion requires that $\Theta(\log^2 n)$ worst-case nodes are relabelled [4]. However, for a universe of size $O(n^2)$, an insertion or deletion can be limited to relabelling only $O(\log n)$ amortized nodes (this is a folklore modification of [10]). In the external memory setting, we consider a linked list to be a linked list of blocks, where each block contains an ordered array of elements with unique ids. A pointer to a specific element is then its id along with a pointer to its containing block.

An insertion is specified by the element to be inserted and a pointer to its intended predecessor. A deletion is specified by a pointer to the element to be deleted. A relabelling consists of a triple (i, ℓ, ℓ'), where i is the id of the element being relabelled, ℓ is its old label, and ℓ' is its new label.

Lemma 5. *There exists an external memory data structure for online list labelling of up to N elements with labels from a universe of size $O(N^2)$ that, upon each update, reports $O(\log N)$ amortized relabellings in $O(1 + (1/B)\log N)$ amortized I/Os.*

Proof. See Appendix A.

Lemma 6. *For any $f \in [2, B]$, there exists an external memory data structure for* $\mathrm{R}^3(\mathrm{s} : N, \mathrm{d} : N)$ *with performance* $(\mathrm{Update} : 1 + (f/B)\log_f N \log N,$ $\mathrm{Query} : \log_f N + (f/B)K, \mathrm{Space} : N/B)$.

Proof. We maintain the online list labelling data structure of Lemma 5 on the y-axis list of our data structure of type $\mathrm{R}^3(\mathrm{s} : N, \mathrm{d} : N)$, using x-coordinates as the unique ids. We build the data structure D of Lemma 1 on an asymmetric $N \times N'$ grid, where $N' = O(N^2)$, instead of an $N \times N$ grid. The bounds of the data structure increase by only constant factors, but we obtain the requirement that points must be a constant factor larger to store the larger y-coordinates. Queries and updates to our data structure include y-axis pointers, which we convert to y-coordinates in $[N']$ in constant I/Os using the online list labelling data structure. We then forward the operations on to D, including the given x-coordinates and our computed y-coordinates. Upon an update, the online list labelling data structure reports $O(\log N)$ relabellings in $O(1+(1/B)\log N)$ I/Os. In a scan through the relabellings, we convert each relabelling of the form (i, ℓ, ℓ') to a deletion of (i, ℓ) from D and an insertion of (i, ℓ') to D. So, our update time increases by an $O(\log N)$ factor. □

We can now simulate the data structure of Lemma 6 similarly to the simulation of Lemma 2.

Lemma 7. *For any $u \in [n^{\delta/O(m)}]$ and $f \in [2, (\delta/O(m))\log n/\log u]$, there exists a data structure for* $\mathrm{R}^3(\mathrm{s} : u, \mathrm{d} : u)$ *with performance* $(\mathrm{Update} : 1+f\log^3 u/\log n,$ $\mathrm{Query} : \log_f u, \mathrm{Space} : u)$.

Proof. See Appendix B.

Applications of Lemmata 4 and 3 to the data structure of Lemma 7 with $u = 2^{(\delta/O(m))((1/f)\log n \log\log n)^{1/3}}$ yield the following theorem.

Theorem 2. *For any $f \in [2, \log n/\log\log n]$, there exists a data structure for* $\mathrm{R}^3(\mathrm{d} : n, \mathrm{d} : n)$ *with performance* $(\mathrm{Update} : f^{1/3}(\log n \log\log n)^{2/3}, \mathrm{Query} : f^{1/3}$ $(\log n \log\log n)^{2/3} + \log_f n, \mathrm{Space} : n)$.

We obtain the 3-sided reporting and RMQ data structures of Table 1 by setting $f = \log^{\epsilon'} n$ for some positive constant $\epsilon' < 3\epsilon$, or alternatively setting $f = 2$.

3.2 Incremental Emptiness

We will require a data structure of Mortensen [14] that solves a problem called *colored predecessor search* in a linked list L with colored nodes. The data structure supports the following operations:

- insert(u, v, c): inserts node u with color c after node v
- delete(u): deletes node u
- change(u, c): changes the color of u to c
- predecessor(u, c): returns the last node of color c that is not after u

Lemma 8 (Theorem 15 of Mortensen [14]). *There exists a linear-space data structure for colored predecessor search in a linked list L that supports all operations in $O(\log\log|L|)$ time.*

We begin with an efficient data structure for 2-sided incremental emptiness.

Theorem 3. *There exists a data structure for* $\mathrm{E}^2_\mathrm{I}(\mathrm{d} : n, \mathrm{d} : n)$ *with performance* $(\mathrm{Update} : \log\log n, \mathrm{Query} : \log\log n, \mathrm{Space} : n)$.

Proof. Without loss of generality, we handle 2-sided queries of the form $(-\infty, r] \times (-\infty, t]$. It is sufficient to find the lowest point in the query range and compare its y-coordinate to t. Since y-coordinates are linked list nodes, we require the list order data structure of Dietz and Sleator [6] to compare them. The lowest point with x-coordinate at most r is the minimal point whose x-coordinate is the predecessor of r. A point $p = (p_x, p_y) \in P$ is minimal if and only if there does not exist another point $(p'_x, p'_y) \in P\backslash\{p\}$ such that $p'_x < p_x$ and $p'_y < p_y$. We maintain the colored predecessor search data structure of Lemma 8 on the x-axis list. We color nodes associated with minimal points one color and all other nodes another color. We can thus find the minimal point whose x-coordinate is the predecessor of r in $O(\log\log n)$ time. A newly inserted point may be a minimal point, which can result in many, say k, previously minimal points becoming non-minimal. For each of these k points we must execute a color change operation, which requires $O(k \log\log n)$ time. However, in the incremental setting, a point p can only transition from minimal to non-minimal at most once, so we can charge the $O(\log\log n)$ cost of the color change of p to the insertion of p. □

Given that 2-sided incremental emptiness queries can be supported very efficiently, we can use an alternative to Lemma 4 to handle n points given a data structure for only $u \leq n$ points: a u-ary range tree instead of a u-ary priority search tree. The range tree requires superlinear space; however, we can reduce space to linear once again with an application of Lemma 3.

Lemma 9. *For any* $u \in [n^{\delta/O(m)}]$ *and* $f \in [2, (\delta/O(m)) \log n/\log u]$, *there exists a data structure for* $\mathrm{E}^3_\mathrm{I}(\mathrm{s} : n, \mathrm{d} : n)$ *with performance* $(\mathrm{Update} : (\log n/\log u)(\log\log n + f \log^3 u/\log n), \mathrm{Query} : \log\log n + \log_f u, \mathrm{Space} : n \log n/\log u)$

Proof. See Appendix C.

An application of Lemma 3 to the data structure of Lemma 9 yields the following theorem.

Theorem 4. *For any* $u \in [n^{\delta/O(m)}]$ *and* $f \in [2, (\delta/O(m)) \log n/ \log u]$*, there exists a data structure for* $\mathrm{E}^3_\mathrm{I}(\mathrm{d} : n, \mathrm{d} : n)$ *with performance* $(\mathrm{Update} : (\log n/ \log u)$ $(\log \log n + f \log^3 u/ \log n), \mathrm{Query} : \log \log n + \log_f u, \mathrm{Space} : n)$.

We obtain the 3-sided incremental emptiness and decremental RMQ data structures of Table 1 by setting $u = 2^{(\log n \log \log n)^{1/3}}$ and $f = 2$, or alternatively setting $u = 2^{(\log \log n)^2}$ and $f = \log^{\epsilon'} n$ for a sufficiently small $\epsilon' > 0$.

References

1. Alstrup, S., Husfeldt, T., Rauhe, T.: Marked ancestor problems. In: Proceedings of the Annual IEEE Symposium on Foundations of Computer Science (FOCS), pp. 534–544 (1998)
2. Andersson, A., Thorup, M.: Dynamic ordered sets with exponential search trees. Journal of the ACM (JACM) 54(3) (2007)
3. Arge, L., Samoladas, V., Vitter, J.S.: On two-dimensional indexability and optimal range search indexing. In: Proceedings of the ACM Symposium on Principles of Database Systems (PODS), pp. 346–357 (1999)
4. Bender, M.A., Cole, R., Demaine, E.D., Farach-Colton, M., Zito, J.: Two simplified algorithms for maintaining order in a list. In: Möhring, R.H., Raman, R. (eds.) ESA 2002. LNCS, vol. 2461, pp. 152–164. Springer, Heidelberg (2002)
5. Bentley, J.L.: Multidimensional divide-and-conquer. Communications of the ACM (CACM) 23(4), 214–229 (1980)
6. Dietz, P.F., Sleator, D.D.: Two algorithms for maintaining order in a list. In: Proceedings of the ACM Symposium on Theory of Computing (STOC), pp. 365–372 (1987)
7. van Emde Boas, P.: Preserving order in a forest in less than logarithmic time and linear space. Information Processing Letters (IPL) 6(3), 80–82 (1977)
8. Fredman, M.L., Willard, D.E.: Blasting through the information theoretic barrier with fusion trees. In: Proceedings of the ACM Symposium on Theory of Computing (STOC), pp. 1–7 (1990)
9. Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM Journal of Computing 13(2), 338–355 (1984)
10. Itai, A., Konheim, A.G., Rodeh, M.: A sparse table implementation of priority queues. In: Even, S., Kariv, O. (eds.) ICALP 1981. LNCS, vol. 115, pp. 417–431. Springer, Heidelberg (1981)
11. Kumar, V., Schwabe, E.J.: Improved algorithms and data structures for solving graph problems in external memory. In: Proceedings of the Annual IEEE Symposium on Parallel and Distributed Processing (SPDP), pp. 169–176 (1996)
12. McCreight, E.M.: Priority search trees. SIAM Journal of Computing 14(2), 257–276 (1985)
13. Mehlhorn, K., Näher, S.: Dynamic fractional cascading. Algorithmica 5(2), 215–241 (1990)
14. Mortensen, C.W.: Fully dynamic orthogonal range reporting on RAM. SIAM Journal of Computing 35(6), 1494–1525 (2006)
15. Willard, D.E.: Examining computational geometry, van Emde Boas trees, and hashing from the perspective of the fusion tree. SIAM Journal of Computing 29(3), 1030–1049 (2000)

A Proof of Lemma 5

Lemma. *There exists an external memory data structure for online list labelling of up to N elements with labels from a universe of size $O(N^2)$ that, upon each update, reports $O(\log N)$ amortized relabellings in $O(1 + (1/B)\log N)$ amortized I/Os.*

Upon an insertion or deletion, we load the target block in a single I/O and add or remove the element. We maintain block sizes of $\Theta(B)$ elements (except when there are too few elements to fill a single block) by splitting and merging adjacent blocks whenever they become a constant factor too large or too small. Each split or merge can be charged to $\Omega(B)$ updates. Each split or merge causes an insertion to or a deletion from the linked list of blocks. We maintain an online list labelling data structure for a polynomially large universe [10] to assign the $O(N/B)$ blocks labels from a universe of size $O((N/B)^2)$ with $O(\log(N/B))$ amortized relabellings per update. If an element has rank r in the block with label ℓ, then its label is $B\ell + r - 1$, which is bounded by $O(N^2)$. For each block relabelled, we can report all of the relabellings for all of its elements in $O(1)$ I/Os.

B Proof of Lemma 7

Lemma. *For any $u \in [n^{\delta/O(m)}]$ and $f \in [2, (\delta/O(m))\log n/\log u]$, there exists a data structure for* $\mathrm{R}^3(\mathrm{s} : u, \mathrm{d} : u)$ *with performance* $(\text{Update} : 1 + f\log^3 u/\log n$, $\text{Query} : \log_f u, \text{Space} : u)$.

We simulate the data structure of Lemma 6 as in Lemma 2, except that elements now require a constant factor more bits due to the polynomially large universe used by the online list labelling data structure. In internal memory, the y-axis list is a standard linked list. To support conversions from pointers to y-axis list nodes to simulated external memory pointers, we store in each y-axis list node the external memory element's unique id (x-coordinate) and a pointer to the simulated block containing the element. Whenever there is a split or a merge, we must update $O(B)$ pointers from the y-axis list to simulated blocks. These updates can be charged to the $\Omega(B)$ updates that are required to cause a split or a merge. When a point is reported by the simulated data structure, its x-coordinate is included. To obtain the associated y-axis list node, we simply maintain (in $O(1)$ time per update) an array of size u containing, for each x-coordinate, the associated y-axis list node.

C Proof of Lemma 9

Lemma. *For any $u \in [n^{\delta/O(m)}]$ and $f \in [2, (\delta/O(m))\log n/\log u]$, there exists a data structure for* $\mathrm{E}^3_{\mathrm{I}}(\mathrm{s} : n, \mathrm{d} : n)$ *with performance* $(\text{Update} : (\log n/\log u)$ $(\log\log n + f\log^3 u/\log n), \text{Query} : \log\log n + \log_f u, \text{Space} : n\log n/\log u)$

We first consider a problem called *subsequence predecessor search* in which we maintain a primary linked list L and a set of secondary linked lists $\mathcal{S}$. For the purposes of this problem, let $n = |L| + \sum_{S \in \mathcal{S}} |S|$ be the total size of all lists. Each node u in a secondary list $S \in \mathcal{S}$ is associated with a primary node $p(u)$, such that mapping S with function p yields a subsequence of L. For any primary node v, there may be multiple nodes u from different secondary lists for which $p(u) = v$. We require the following operations:

- $\mathrm{insert}_{\mathrm{p}}(u, v)$: inserts primary node u after v in L
- $\mathrm{delete}_{\mathrm{p}}(u)$: deletes primary node u from L (only if there is no secondary node v with $p(v) = u$)
- $\mathrm{insert}_{\mathrm{s}}(u, S)$: inserts a new secondary node v with $p(v) = u$ to secondary list S (preserving the subsequence ordering of S)
- $\mathrm{delete}_{\mathrm{s}}(u)$: deletes secondary node u from its secondary list
- $\mathrm{predecessor}(u, S)$: returns the last secondary node v in secondary list S such that $p(v)$ is not after primary node u in L
- $\mathrm{secondaries}(u)$: returns all secondary nodes v such that $p(v) = u$ for primary node u
- $\mathrm{primary}(u)$: returns $p(u)$ for secondary node u

Lemma 10. *There exists a data structure for subsequence predecessor search that requires $O(n)$ space, $O(\log \log n)$ time for updates and predecessor queries, $O(1 + k)$ time to report the k secondary nodes associated with a primary node, and $O(1)$ time to find the primary node associated with a secondary node.*

Proof. We construct an aggregate doubly linked list A containing, for each primary node u in order, pointers to all secondary nodes v such that $p(v) = u$ followed by a pointer to u. We also store pointers from primary and secondary nodes to their corresponding aggregate nodes. We color each node of A with the list of the node to which it stores a pointer. For each aggregate node pointing to a secondary node u, we also store an extra pointer to $p(u)$ which does not affect the aggregate node's color. We build the colored predecessor search data structure of Lemma 8 on A, which has size n. We can then support a subsequence predecessor query in secondary list S with a colored predecessor query in A with color S. Reporting the secondary nodes of a primary node requires a walk in A. Reporting $p(u)$ for secondary node u requires following two pointers. It is a simple exercise to verify that all of the update operations can be supported using a constant number of colored predecessor search operations and a constant amount of pointer rewiring. $\square$

We build a static u-ary range tree on x-coordinates. In each internal node of the range tree, we build 2 auxiliary data structures on the points stored in the node. Each of these data structures has its own y-axis list corresponding to a subsequence of the original y-axis list. The total size of all y-axis lists is $O(nh)$, where h is the height of the range tree. We build the subsequence predecessor search data structure of Lemma 10 using the original y-axis list as the primary list and all other y-axis lists as secondary lists. This data structure requires $O(nh)$ space. Now, given a query or update, we can translate it in $O(\log\log(nh)) = O(\log\log n)$ time into the coordinate space of a specific auxiliary data structure. Also, we can convert a point in the coordinate space of a specific auxiliary data structure to our original coordinate space in $O(1)$ time.

One of our auxiliary data structures is that of Theorem 3, which handles 2-sided ranges of the form $(-\infty, r] \times (-\infty, t]$ and $[\ell, \infty) \times (-\infty, t]$. The other is a data structure which handles 3-sided ranges that are aligned to the boundaries of the x-ranges of the node's children. We call these *aligned* queries. This data structure for aligned queries is the data structure of Lemma 7 built on the lowest points in each of the node's chidren. An aligned query is empty if and only if it contains none of the lowest points in each of the node's children. Since a node has u children, the axes of our data structure for aligned queries have size u. All of these auxiliary data structures require a total of $O(nh)$ space.

Given an insertion of a point p, we insert p into the 2-sided data structures of all $O(h)$ nodes of the range tree to which p belongs at a cost of $O(h \log\log n)$. If p becomes the lowest point in the child of some node u, we must delete the old lowest point from the data structure of Lemma 7 in u and insert p. This introduces another term of $O(hf \log^3 u/\log n)$ to our update time.

Given a query of the form $[\ell, r] \times (-\infty, t]$, we find the lowest common ancestor (LCA) in the range tree of the leaf corresponding to x-coordinate ℓ and the leaf corresponding to x-coordinate r. Using a linear-space data structure, this LCA operation requires only $O(1)$ time [9]. In the resulting node u, we can decompose the query into an aligned query and two 2-sided queries in children of u. Our range is empty if and only if all 3 of these subranges are empty. The 3 emptiness queries to the auxiliary data structures require $O(\log\log n + \log_f u)$ time. Since the height of the range tree is $O(\log_u n) = O(\log n/\log u)$, we are done.

Author Index

Abboud, Amir 1
Abed, Fidaa 13
Acar, Umut A. 25
Agarwal, Pankaj K. 37
Agarwal, Rachit 49
Alewijnse, Sander P.A. 61
Attali, Dominique 74

Bekos, Michael A. 87
Ben-Avraham, Rinat 100
Bender, Michael A. 112
Bhattacharyya, Arnab 125
Bilò, Davide 137
Björklund, Andreas 149
Bläsius, Thomas 161
Bliznets, Ivan 173
Boissonnat, Jean-Daniel 185
Bouts, Quirijn W. 61
Bringmann, Karl 197
Brink, Alex P. ten 61
Brückner, Guido 161
Buchbinder, Niv 209

Chakaravarthy, Venkatesan T. 222
Chalermsook, Parinya 235
Chan, Timothy M. 247
Charguéraud, Arthur 25
Charikar, Moses 260
Chen, Jianer 642
Chen, Shahar 209
Cheng, Siu-Wing 272
Correa, José R. 13

Darwish, Omar 284
Davoodi, Pooya 296
Dean, Brian C. 309
Delling, Daniel 321
Desai, Amey 467
Devillers, Olivier 74
de Wolf, Ronald 592
Dijk, Thomas C. van 87
Dvořák, Zdeněk 334, 346

Efentakis, Alexandros 358
Efthymiou, Charilaos 371
Elmasry, Amr 284
Ene, Alina 382
Even, Guy 394

Farach-Colton, Martín 112
Farruggia, Andrea 406
Ferragina, Paolo 406
Ferreira, Rui 418
Fineman, Jeremy T. 296
Fink, Martin 87
Fiorini, Samuel 430
Fomin, Fedor V. 173, 443
Friedrich, Tobias 197

Gawrychowski, Paweł 455
Ghashami, Mina 467
Gibson, Matt 480
Glisse, Marc 74
Goldberg, Andrew V. 321
Golovach, Petr A. 492
Goswami, Mayank 112
Grohe, Martin 505
Grossi, Roberto 418
Gualà, Luciano 137
Gupta, Anupam 517
Gupta, Shalmoli 222
Gutin, Gregory 530

Har-Peled, Sariel 37, 542
He, Meng 247
Hell, Pavol 554
Henze, Matthias 100
Henzinger, Monika 260
Heydrich, Sandy 235
Hoffmann, Michael 566
Holm, Eugenia 235
Huang, Chien-Chung 13

Iacono, John 296

Jalasutram, Rommel 309
Jansen, Bart M.P. 579
Jaume, Rafel 100
Jeffery, Stacey 592
Jones, Mark 530

Kamiński, Marcin 492
Karrenbauer, Andreas 235
Kaski, Petteri 149
Kersting, Kristian 505
Keszegh, Balázs 100
Kindermann, Philipp 87
Kobourov, Stephen 87
Kociumaka, Tomasz 605
Kowalik, Łukasz 149
Krithika, R. 430
Krohmer, Anton 197
Kupec, Martin 334
Kusters, Vincent 566

Larkin, Daniel H. 618
Lazard, Sylvain 74
Leucci, Stefano 137
Lewenstein, Moshe 455, 630
Lewi, Kevin 1
Li, Wenjun 642
Liu, Jingcheng 654
Lokshtanov, Daniel 443, 666
Lu, Pinyan 654

Magniez, Frederic 592
Malchik, Caleb 677
Maniatis, Spyridon 492
Maria, Clément 185
McGregor, Andrew 689
Medina, Moti 394
Medjedovic, Dzejla 112
Mencel, Liam 272
Merz, Florian 701
Miltzow, Tillmann 566
Miyazawa, Flávio K. 713
Mladenov, Martin 505
Mnich, Matthias 346
Mohar, Bojan 554
Molinaro, Marco 517
Montes, Pablo 112
Munro, J. Ian 247, 630

Naor, Joseph (Seffi) 209
Narayanaswamy, N.S. 430
Navarro, Gonzalo 725
Newman, Alantha 737
Nguyễn, Huy L. 260, 382
Nicholson, Patrick K. 455, 630

Özkan, Özgür 296

Pagh, Rasmus 750
Pajor, Thomas 321
Panolan, Fahad 443
Pedrosa, Lehilton L.C. 713
Pfoser, Dieter 358
Phillips, Jeff M. 467
Pilipczuk, Marcin 173
Pilipczuk, Michał 173
Price, Eric 689
Proietti, Guido 137
Puglisi, Simon J. 725
Pupyrev, Sergey 87

Räcke, Harald 774
Rafiey, Arash 554
Rainey, Mike 25
Raman, Venkatesh 430, 630
Raz, Orit E. 100
Rizzi, Romeo 418, 762
Ron, Dana 394
Roy, Sambuddha 222
Roy, Subhro 542
Roy Choudhury, Anamitra 222
Rutter, Ignaz 161

Sabharwal, Yogish 222
Sacomoto, Gustavo 418
Sagot, Marie-France 418
Sanders, Peter 701
Saurabh, Saket 443, 666
Schouery, Rafael C.S. 713
Selman, Erkal 505
Shachnai, Hadas 786
Shah, Chintan 774
Sharir, Micha 100
Sheng, Bin 530
Sirén, Jouni 725
Spoerhase, Joachim 87
Starikovskaya, Tatiana 605
Stöckel, Morten 750
Suchý, Ondřej 666
Suri, Subhash 37
Sviridenko, Maxim 713

Tarjan, Robert E. 618
Thilikos, Dimitrios M. 492
Tomescu, Alexandru I. 762
Tsai, Meng-Tsung 112
Tubis, Igor 100
Tůma, Vojtěch 334

van Brink, Martijn 798
van der Zwaan, Ruben 798
Varadarajan, Kasturi 480
Venturini, Rossano 406
Vigneron, Antoine 272
Vildhøj, Hjalte Wedel 605
Vorotnikova, Sofya 689

Wakabayashi, Yoshiko 713
Wang, Jianxin 642
Wang, Joshua R. 810
Waters, Chad 309
Wei, Zhewei 830
Werneck, Renato F. 321
Wilkinson, Bryan T. 842
Williams, Ryan 1
Winslow, Andrew 677
Wolff, Alexander 87
Wu, Xiaodong 480

Yi, Ke 830
Yıldız, Hakan 37

Zehavi, Meirav 786
Zhang, Chihao 654
Zhang, Wuzhou 37
Zhou, Gelin 247